AYI NEDJIMI Consultants

Windows Scheduler Internals : Architecture, Performance Tracing & Optimisation

Catégorie : Articles Techniques Lecture : 90 min Publié le : 29/03/2026 Auteur : Ayi NEDJIMI

Whitepaper expert sur le scheduler Windows NT : architecture interne, algorithmes, CPU hybrides P/E-core, Thread Director, ETW tracing, WinDbg, optimisations NUMA et sécurité.

Introduction

Le scheduler — ou ordonnanceur — constitue le cœur battant de tout système d'exploitation moderne. Dans le noyau Windows NT, ce composant décide, plusieurs milliers de fois par seconde sur chaque cœur logique, quel thread obtient du temps processeur. Cette décision, prise en quelques microsecondes, détermine la réactivité d'une interface utilisateur, le débit d'un serveur de base de données, la latence d'un moteur de jeu, et la posture de sécurité d'un hyperviseur. Comprendre le scheduler en profondeur n'est pas un exercice académique : c'est une compétence opérationnelle indispensable pour tout ingénieur système, développeur kernel, ou consultant en performance et sécurité.

Ce document est un whitepaper technique de référence. Il ne s'agit pas d'une introduction vulgarisée ni d'un résumé marketing des nouveautés de Windows 11. Nous allons plonger dans les structures de données internes du noyau (`KTHREAD`, `KPRCB`, `EPROCESS`), dans les algorithmes de sélection de thread, dans les mécanismes de boost de priorité et d'anti-starvation, dans le fonctionnement du Thread Director Intel pour les architectures hybrides, et dans l'interaction entre le scheduler et l'hyperviseur Hyper-V. Chaque affirmation technique est vérifiable par reverse engineering ou par instrumentation ETW.

Le scheduler Windows a subi des transformations majeures depuis Windows NT 3.1 (1993) jusqu'à Windows 11 24H2 et Server 2025. L'architecture matérielle a évolué d'un monde mono-processeur vers des topologies NUMA multi-socket avec des centaines de cœurs logiques, des architectures hybrides P-core/E-core, et des couches de virtualisation omniprésentes. Le scheduler a dû s'adapter à chacune de ces évolutions tout en maintenant la compatibilité binaire avec des applications Win32 écrites il y a trente ans. Cette tension entre modernité et héritage est un fil conducteur de ce document.

Objectifs du document

Ce whitepaper poursuit plusieurs objectifs complémentaires. Premièrement, fournir une cartographie complète des mécanismes internes du scheduler Windows NT tel qu'il existe dans Windows 11 (versions 23H2 et 24H2) et Windows Server 2025. Deuxièmement, expliquer les différences de comportement entre le profil client (orienté réactivité) et le profil serveur (orienté débit), en identifiant les paramètres de configuration qui les distinguent. Troisièmement, documenter les interactions entre le scheduler et les technologies matérielles récentes : architectures hybrides Intel (Alder Lake, Raptor Lake, Meteor Lake), topologies NUMA, et

extensions de virtualisation. Quatrièmement, fournir une méthodologie reproductible de traçage et de diagnostic des problèmes de scheduling, à l'aide d'ETW, WinDbg et des compteurs de performance. Enfin, offrir des recommandations d'optimisation fondées sur une compréhension réelle des mécanismes internes, et non sur des recettes empiriques.

Ce document s'adresse aux ingénieurs systèmes seniors, aux développeurs kernel et driver, aux architectes infrastructure, aux consultants en performance, et aux professionnels de la cybersécurité qui ont besoin de comprendre comment le scheduler affecte l'isolation entre processus, la surface d'attaque des side-channels, et les mécanismes de mitigation comme le Core Scheduler de Hyper-V. Pour un consultant en cybersécurité, la connaissance du scheduler est essentielle : les attaques par side-channel (Spectre, MDS, L1TF) exploitent directement le partage de ressources matérielles entre threads co-schedulés, et les mécanismes de défense (Core Scheduler, KVAS) sont implémentés dans le dispatcher lui-même.

Périmètre

Le périmètre de ce document couvre deux versions majeures de Windows. Côté client, Windows 11 dans ses déclinaisons 23H2 (Sun Valley 3, build 22631) et 24H2 (Germanium, build 26100). Côté serveur, Windows Server 2025 (build 26100, partageant le même noyau que Windows 11 24H2). Cette convergence de build entre client et serveur est significative : le code du scheduler est identique, mais les paramètres de configuration diffèrent. Le noyau (`ntoskrnl.exe`) est le même binaire ; ce sont les valeurs par défaut du registre, les stratégies de boost, et la configuration du quantum qui créent les différences de comportement observées.

Côté client, le scheduler est optimisé pour la réactivité de l'interface utilisateur. Le thread de la fenêtre au premier plan reçoit un boost de priorité et un quantum étendu. Le Multimedia Class Scheduler Service (MMCSS) garantit un temps CPU minimum aux threads audio et vidéo. Le mode EcoQoS permet de reléguer les tâches d'arrière-plan sur les E-cores avec une fréquence réduite. Côté serveur, le scheduler privilégie le débit global et l'équité entre services. Les quantums sont plus longs (12 intervalles d'horloge contre 2 par défaut sur le client), le boost de premier plan est désactivé, et les politiques NUMA sont plus agressives pour maximiser la localité mémoire dans les configurations multi-socket.

Ce document ne couvre pas le scheduling en mode utilisateur (UMS, abandonné), le scheduling des fibres (coopératif, géré entièrement en user-mode), ni le scheduling temps-réel dur (Windows n'est pas un RTOS, même avec les priorités 16-31). Le scheduling GPU (WDDM) et le scheduling I/O (I/O priority) sont mentionnés uniquement dans leurs interactions avec le scheduler CPU.

Hypothèses techniques

Ce document suppose que le lecteur possède une connaissance solide du langage C, une familiarité avec l'assemblage x86-64 (au minimum la capacité de lire du code désassemblé), et une compréhension des concepts fondamentaux des systèmes d'exploitation : espace noyau vs espace utilisateur, interruptions, appels système, mémoire virtuelle, et synchronisation. La connaissance des IRQL (Interrupt Request Levels) de Windows est un prérequis important : le scheduler opère au niveau DISPATCH_LEVEL (IRQL 2), ce qui a des implications directes sur les verrous utilisables et les opérations autorisées dans le code de dispatch. Le lecteur devrait être

familier avec les concepts de base des processeurs modernes : pipeline, caches L1/L2/L3, TLB, et au minimum une compréhension superficielle de la prédiction de branchement et de l'exécution spéculative (pour comprendre les mitigations side-channel).

Méthodologie d'analyse

Les informations présentées dans ce document proviennent de quatre sources complémentaires. La première est le reverse engineering statique du noyau Windows à l'aide d'IDA Pro et Ghidra. Le fichier `ntoskrnl.exe` de Windows 11 24H2 (build 26100) a été désassemblé et les fonctions clés du scheduler (`KiDispatcherReadyThread`, `KiSearchForNewThread`, `KiQuantumEnd`, `KiSwapContext`, `KiSelectProcessor`) ont été analysées instruction par instruction. Les symboles publics de Microsoft (PDB) fournissent les noms de fonctions et de structures, ce qui facilite considérablement l'analyse. La deuxième source est le débogage kernel en temps réel avec WinDbg, connecté à une machine virtuelle Hyper-V via un canal série virtuel. Cela permet d'inspecter les structures de données en mémoire, de poser des breakpoints sur les fonctions du scheduler, et d'observer le comportement en conditions réelles. La troisième source est le traçage ETW (Event Tracing for Windows), en particulier les providers `Microsoft-Windows-Kernel-Process` et `Microsoft-Windows-Kernel-Processor-Power`, ainsi que les événements du kernel logger (context switches, ready thread, dispatcher). Windows Performance Recorder (WPR) et Windows Performance Analyzer (WPA) sont les outils principaux de capture et d'analyse. La quatrième source est la littérature publiée, en particulier Windows Internals, 7th Edition (Russinovich, Solomon, Ionescu), qui reste la référence incontournable malgré son antériorité par rapport aux dernières versions de Windows.

Fondations du Scheduler Windows

Avant de plonger dans l'architecture interne du scheduler, il est indispensable de poser les fondations conceptuelles. Le modèle de scheduling Windows repose sur des choix architecturaux faits lors de la conception initiale de Windows NT au début des années 1990, sous la direction de Dave Cutler. Ces choix — scheduling préemptif basé sur les priorités, thread comme unité de scheduling, quantum de temps fixe — restent les piliers du scheduler moderne, même si les implémentations ont été profondément remaniées au fil des versions.

Niveaux de priorité Windows (0-31) — classes dynamiques et temps réel

Les 32 niveaux de priorité du scheduler NT : zone dynamique (1-15) avec boosting, zone temps réel (16-31) à priorité fixe

Process vs Thread : EPROCESS/KPROCESS vs ETHREAD/KTHREAD

Dans le noyau Windows, un processus est représenté par deux structures imbriquées : `EPROCESS` (Executive Process) et `KPROCESS` (Kernel Process). La structure `KPROCESS` est le premier membre de `EPROCESS` et contient les informations utilisées directement par le scheduler : la priorité de base du processus (`BasePriority`), le quantum par défaut (`QuantumReset`), le masque d'affinité (`Affinity`), et la liste des threads appartenant au processus (`ThreadListHead`). La structure `EPROCESS` encapsule `KPROCESS` et ajoute les informations de niveau exécutif : l'espace

d'adressage (pointeur vers le répertoire de pages), la table des handles, les informations de sécurité (token primaire), les quotas, les statistiques de comptabilité, et les données du sous-système Win32 (via `PEB`).

Un thread est représenté de manière analogue par `ETHREAD` (Executive Thread) et `KTHREAD` (Kernel Thread). `KTHREAD` est le premier membre de `ETHREAD` et contient toutes les informations nécessaires au scheduler : l'état du thread (`State`), la priorité courante (`Priority`), la priorité de base (`BasePriority`), le quantum restant (`Quantum`), le masque d'affinité (`Affinity`), le processeur idéal (`IdealProcessor`), le pointeur de pile noyau (`KernelStack`), la frame de trap (`TrapFrame`), et les blocs d'attente (`WaitBlockList`). `ETHREAD` ajoute les informations de niveau exécutif : l'identifiant de thread (`Cid`), le token d'impersonation, les compteurs I/O, et les données LPC/ALPC.

La distinction fondamentale est la suivante : un processus est un conteneur — il possède un espace d'adressage, des ressources, et un contexte de sécurité. Un thread est une unité d'exécution — il possède un contexte processeur (registres, pile) et un état de scheduling. Le scheduler ne prend jamais de décision au niveau du processus. Il ne « schedule » pas un processus ; il schedule des threads individuels. Quand on dit qu'un processus « tourne », on veut dire qu'au moins un de ses threads est dans l'état Running sur un processeur logique.

On peut inspecter ces structures dans WinDbg avec les commandes suivantes :

```
kd> dt nt!_KPROCESS
   +0x000 Header           : _DISPATCHER_HEADER
   +0x018 ProfileListHead  : _LIST_ENTRY
   +0x028 DirectoryTableBase : Uint8B
   +0x030 ThreadListHead   : _LIST_ENTRY
   +0x040 ProcessLock      : Uint4B
   +0x044 ProcessTimerDelay : Uint4B
   +0x048 DeepFreezeStartTime : Uint8B
   +0x050 Affinity         : _KAFFINITY_EX
   +0x0f8 ReadyListHead    : _LIST_ENTRY
   +0x108 BasePriority     : Char
   +0x109 QuantumReset     : Char
   ...

kd> dt nt!_KTHREAD
   +0x000 Header           : _DISPATCHER_HEADER
   +0x018 SListFaultAddress : Ptr64 Void
   +0x020 QuantumTarget    : Uint8B
   +0x028 InitialStack     : Ptr64 Void
   +0x030 StackLimit       : Ptr64 Void
   +0x038 StackBase        : Ptr64 Void
   +0x040 ThreadLock       : Uint8B
   +0x048 CycleTime        : Uint8B
   +0x050 CurrentRunTime   : Uint4B
   +0x058 TrapFrame        : Ptr64 _KTRAP_FRAME
   +0x098 ApcState         : _KAPC_STATE
   +0x0dc Priority         : Char
   +0x120 State            : UChar
   +0x164 BasePriority     : Char
   +0x165 PriorityDecrement : UChar
   +0x1c8 Affinity         : _GROUP_AFFINITY
   +0x1d8 IdealProcessor   : Uint4B
   ...
```

Modèle NT : le thread comme unité de scheduling

Le choix du thread comme unité de scheduling est un héritage direct de la conception de Windows NT, qui s'est délibérément écarté du modèle UNIX classique. Dans les systèmes UNIX traditionnels (avant POSIX threads), le processus était à la fois le conteneur de ressources et l'unité de scheduling. Le `fork()` créait un nouveau processus avec son propre espace d'adressage, et le scheduler manipulait des processus. L'introduction des threads POSIX (pthreads) dans les années 1990 a ajouté le concept de threads au-dessus du modèle processus existant, souvent avec des implémentations hybrides (threads N:M, LinuxThreads, puis NPTL). Linux, jusqu'à aujourd'hui, utilise `task_struct` comme unité unique, un thread étant simplement un `task_struct` qui partage son espace d'adressage avec d'autres via `clone()`.

Dave Cutler, en concevant Windows NT, a opté pour une séparation nette dès le départ. Un processus (`EPROCESS`) ne possède jamais de contexte processeur propre — il n'a pas de registres, pas de pile, pas d'état de scheduling. Tout processus a au minimum un thread (le thread initial), et c'est ce thread qui s'exécute. Cette séparation a des conséquences architecturales profondes : le code du scheduler ne manipule que des structures `KTHREAD`, jamais des `EPROCESS`. Les fonctions `KiDispatcherReadyThread`, `KiSearchForNewThread`, et `KiSwapContext` opèrent exclusivement sur des pointeurs `KTHREAD`. Les informations du processus parent ne sont consultées que pour obtenir la priorité de base (`KPROCESS.BasePriority`) et le quantum par défaut (`KPROCESS.QuantumReset`). Ce design simplifie considérablement le scheduler et permet une granularité fine du contrôle de l'exécution.

Cette architecture a un impact direct sur la sécurité et l'escalade de privilèges. Le token de sécurité d'un processus s'applique à tous ses threads par défaut, mais un thread individuel peut assumer un token d'impersonation différent. Le scheduler ne prend pas en compte le contexte de sécurité dans ses décisions — un thread tournant sous SYSTEM et un thread tournant sous un utilisateur non-privilégié, à la même priorité, reçoivent exactement le même traitement. C'est le sous-système de sécurité qui vérifie les droits d'accès lors des appels système, pas le scheduler.

Scheduling préemptif

Windows NT utilise un scheduling préemptif basé sur les priorités. Le terme « préemptif » signifie que le système d'exploitation peut interrompre un thread en cours d'exécution à tout moment pour donner le processeur à un thread de priorité supérieure, sans que le thread interrompu n'ait besoin de coopérer ou de céder volontairement le contrôle. C'est une rupture fondamentale avec le modèle coopératif de Windows 3.x et des premières versions de MacOS, où une application mal écrite (boucle infinie sans appel à `PeekMessage`) pouvait geler l'intégralité du système.

Le mécanisme de préemption repose sur l'interruption d'horloge (clock interrupt). Sur les systèmes x86-64 modernes, cette interruption est typiquement générée par le LAPIC (Local Advanced Programmable Interrupt Controller) à une fréquence configurable. Par défaut, Windows utilise une résolution d'horloge de 15.625 ms (64 ticks par seconde), mais cette valeur peut être modifiée par les applications via `timeBeginPeriod()` jusqu'à 0.5 ms, ou par le système via le Dynamic Tick (tickless kernel) introduit dans Windows 8. À chaque interruption d'horloge, le handler d'interruption incrémente le compteur de ticks système et décrémente le quantum du

thread en cours d'exécution. Quand le quantum atteint zéro, le handler enqueue un DPC (Deferred Procedure Call) qui invoquera `KiQuantumEnd` au retour de l'interruption, au niveau IRQL DISPATCH_LEVEL.

La préemption peut également être déclenchée par un événement asynchrone : un thread de priorité supérieure devient prêt (par exemple, une I/O se termine et réveille un thread en attente avec un boost de priorité), un appel à `SetThreadPriority` élève la priorité d'un thread au-dessus de celle du thread courant, ou un IPI (Inter-Processor Interrupt) signale qu'un thread de haute priorité doit être dispatché sur un autre processeur. Dans tous ces cas, la fonction `KiDispatcherReadyThread` est appelée, et si le thread nouvellement prêt a une priorité supérieure au thread courant sur le processeur cible, une préemption est programmée.

Classes de priorité (0-31)

Le scheduler Windows utilise 32 niveaux de priorité, numérotés de 0 à 31. Ces niveaux sont divisés en deux classes distinctes ayant des comportements fondamentalement différents.

Plage	Classe	Comportement	Exemples
0	Système	Réservé au Zero Page Thread	Thread de mise à zéro des pages libres
1-15	Dynamique	Priorité ajustable par boost/decay	Applications utilisateur, services, processus système
16-31	Temps réel	Priorité fixe, pas de boost/decay	Drivers audio, processus critiques SYSTEM

Le niveau 0 est réservé exclusivement au Zero Page Thread, un thread système spécial qui met à zéro les pages de mémoire physique en arrière-plan pour les préparer à l'allocation. Aucun thread utilisateur ne peut atteindre le niveau 0.

Les niveaux 1 à 15 constituent la classe dynamique. La priorité effective (courante) d'un thread dans cette plage peut varier entre sa priorité de base et 15. Le système peut temporairement augmenter la priorité d'un thread (boost) en réponse à certains événements (complétion d'I/O, activation de fenêtre, fin d'attente), puis la laisser décroître (decay) progressivement vers la priorité de base. Ce mécanisme de boost/decay est central pour la réactivité du système.

Les niveaux 16 à 31 constituent la classe temps réel. Malgré leur nom, ces niveaux ne fournissent pas de garanties temps-réel dur — Windows n'est pas un RTOS. Cependant, un thread à priorité 16+ ne voit jamais sa priorité modifiée automatiquement par le système (pas de boost, pas de decay). Un thread à priorité 31 préemptera toujours un thread à priorité 30 ou moins. L'utilisation des priorités temps réel est réservée aux composants système critiques et aux drivers. Un processus utilisateur a besoin du privilège `SeIncreaseBasePriorityPrivilege` pour se placer dans la classe temps réel.

La priorité effective d'un thread résulte de la combinaison de deux réglages : la classe de priorité du processus (`SetPriorityClass`) et le niveau de priorité du thread au sein de cette classe (`SetThreadPriority`). Voici la correspondance :

```
// Priorité de base = f(classe processus, niveau thread)
// Exemples pour la classe NORMAL_PRIORITY_CLASS (base = 8) :
//   THREAD_PRIORITY_LOWEST       = base - 2 = 6
//   THREAD_PRIORITY_BELOW_NORMAL = base - 1 = 7
//   THREAD_PRIORITY_NORMAL       = base     = 8
//   THREAD_PRIORITY_ABOVE_NORMAL = base + 1 = 9
//   THREAD_PRIORITY_HIGHEST      = base + 2 = 10
//
// Pour REALTIME_PRIORITY_CLASS (base = 24) :
//   THREAD_PRIORITY_LOWEST       = 16
//   THREAD_PRIORITY_NORMAL       = 24
//   THREAD_PRIORITY_HIGHEST      = 31
//   THREAD_PRIORITY_TIME_CRITICAL= 31

// Classes de processus et leur priorité de base :
// IDLE_PRIORITY_CLASS            = 4
// BELOW_NORMAL_PRIORITY_CLASS    = 6
// NORMAL_PRIORITY_CLASS          = 8
// ABOVE_NORMAL_PRIORITY_CLASS    = 10
// HIGH_PRIORITY_CLASS            = 13
// REALTIME_PRIORITY_CLASS        = 24
```

Quantum et time slicing

Le quantum représente la durée maximale pendant laquelle un thread peut s'exécuter de manière continue avant que le scheduler ne réévalue l'allocation du processeur. Contrairement à une idée reçue, le quantum n'est pas mesuré directement en millisecondes mais en « quantum units » internes. Historiquement, un quantum unit correspondait à un tiers d'un tick d'horloge. Avec le tick d'horloge par défaut de 15.625 ms, un quantum de 6 units correspondait à environ 2 ticks soit 31.25 ms, et un quantum de 36 units à 12 ticks soit 187.5 ms.

Sur Windows 11 (profil client), le quantum par défaut est court et variable. Le thread de premier plan (foreground) reçoit un quantum étendu, typiquement 6 units de base plus un bonus configuré via `PspForegroundQuantum`. Le tableau `PspForegroundQuantum` est indexé par la valeur de `Win32PrioritySeparation` et contient trois entrées correspondant aux niveaux de boost du premier plan (0, 1, 2). Par défaut sur un client, la séparation est configurée pour le boost maximal, ce qui triple le quantum du thread de premier plan.

Sur Windows Server 2025, le quantum par défaut est plus long et fixe : 36 units (12 ticks, environ 187.5 ms). Il n'y a pas de boost de quantum pour le premier plan, car sur un serveur il n'y a typiquement pas d'interface graphique interactive. Ce quantum long favorise le débit : un thread serveur peut effectuer plus de travail avant d'être interrompu, ce qui réduit le surcoût des context switches.

Le paramètre de registre `HKLM\SYSTEM\CurrentControlSet\Control\PriorityControl\Win32PrioritySeparation` contrôle trois aspects du quantum : la longueur (courte vs longue), la variabilité (fixe vs variable), et le boost de premier plan (0, 1, 2). La valeur est un champ de bits de 6 bits :

```
// Win32PrioritySeparation (6 bits) :
// Bits 5-4 : Longueur du quantum
//   00 = défaut système (court pour client, long pour serveur)
//   01 = long (36 units base)
//   10 = court (6 units base)
//
// Bits 3-2 : Variabilité
//   00 = défaut système (variable pour client, fixe pour serveur)
//   01 = variable (quantum dépend du statut foreground/background)
//   10 = fixe (même quantum pour tous)
//
// Bits 1-0 : Séparation foreground/background
//   00 = pas de boost
//   01 = boost moyen
//   10 = boost maximum

// Valeur par défaut Windows 11 client : 0x26 (100110b)
//   = quantum court, variable, boost max
// Valeur par défaut Server 2025 : 0x18 (011000b)
//   = quantum long, fixe, pas de boost
```

Starvation, boosting et aging

Le mécanisme de boost de priorité est un élément central de la réactivité du système. Windows applique des boosts temporaires dans plusieurs situations spécifiques. Quand une I/O se termine, le thread qui attendait cette I/O reçoit un boost dont l'amplitude dépend du type de périphérique : +1 pour un disque, +2 pour un réseau, +6 pour un clavier ou une souris, +8 pour un événement sonore. Ce boost s'ajoute à la priorité de base du thread (pas à sa priorité courante) et est plafonné à 15 — le boost ne peut jamais faire passer un thread dans la classe temps réel.

Après un boost, la priorité du thread décroît d'un niveau à chaque expiration de quantum, jusqu'à revenir à la priorité de base. Par exemple, un thread avec une priorité de base 8 qui reçoit un boost clavier (+6) se retrouve à priorité 14. À l'expiration de son premier quantum, il passe à 13, puis 12, puis 11, et ainsi de suite jusqu'à retrouver sa priorité de base 8 au bout de 6 quantums. Ce mécanisme de decay assure que les boosts sont transitoires et ne créent pas de situation où un thread monopolise indéfiniment une priorité élevée.

Le boost de premier plan (foreground boost) est spécifique au profil client. Quand une fenêtre passe au premier plan, tous les threads de son processus reçoivent un boost de priorité (typiquement +2) et un quantum étendu. Ce mécanisme est responsable de la sensation de réactivité lorsqu'on clique sur une fenêtre : les threads de l'application au premier plan obtiennent temporairement un avantage de scheduling par rapport aux tâches d'arrière-plan.

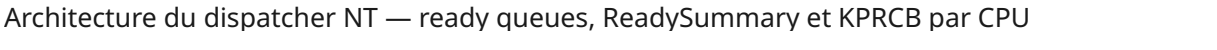
L'anti-starvation est géré par le Balance Set Manager, un thread système qui s'exécute périodiquement (toutes les secondes environ). Il scanne les ready queues à la recherche de threads qui n'ont pas été exécutés depuis un temps anormalement long (typiquement 4 secondes). Un tel thread est considéré comme victime de starvation — il est prêt à s'exécuter mais n'obtient jamais de temps CPU parce que des threads de priorité supérieure monopolisent le processeur. Le Balance Set Manager booste temporairement ces threads affamés à la priorité

15, leur accorde un quantum, puis les laisse décroître normalement. Ce mécanisme empêche les inversions de priorité pathologiques et assure qu'aucun thread prêt n'est indéfiniment privé de CPU, même en situation de forte charge.

> **À retenir :** Le scheduler Windows utilise 32 niveaux de priorité. Les niveaux 1-15 (classe dynamique) sont sujets au boost et au decay automatiques. Les niveaux 16-31 (classe temps réel) ont une priorité fixe. Le mécanisme d'anti-starvation du Balance Set Manager garantit qu'aucun thread prêt n'est indéfiniment privé de CPU, en boostant temporairement les threads affamés à la priorité 15.

Architecture interne du Scheduler NT

L'architecture interne du scheduler Windows NT est le résultat de trente ans d'évolution incrémentale. Le code du dispatcher, situé dans `ntoskrnl.exe`, est parmi le code le plus optimisé du noyau — chaque instruction compte, car ces fonctions sont invoquées des millions de fois par seconde sur un système chargé. Cette section décortique les structures de données centrales, les fonctions clés, et l'organisation des ready queues qui permettent au scheduler de prendre des décisions en temps constant O(1).



Architecture du dispatcher NT — ready queues, ReadySummary et KPRCB par CPU

Vue d'ensemble du dispatcher : threads → 32 ready queues avec lookup O(1) via BitScanReverse → distribution sur les KPRCB par CPU

Vue d'ensemble du dispatcher

Le dispatcher est le composant du noyau responsable du context switching et de la sélection du prochain thread à exécuter. Il opère au niveau IRQL DISPATCH_LEVEL (2), ce qui signifie que pendant l'exécution du code du dispatcher, les interruptions de priorité inférieure (APC_LEVEL, PASSIVE_LEVEL) sont masquées, mais les interruptions matérielles de priorité supérieure (device interrupts, clock interrupt, IPI) restent actives. Ce choix d'IRQL a des implications directes : le code du dispatcher ne peut pas accéder à la mémoire paginée (risque de page fault à un IRQL trop élevé), ne peut pas acquérir de mutex (seulement des spinlocks), et ne peut pas appeler des fonctions qui pourraient attendre.

Les fonctions centrales du dispatcher forment un ensemble cohérent :

- `KiDispatcherReadyThread` — appelée quand un thread devient prêt (fin d'attente, création, boost). Elle sélectionne un processeur cible et insère le thread dans la ready queue appropriée, ou préempte le thread courant si le nouveau thread est plus prioritaire.
- `KiSearchForNewThread` — appelée quand le thread courant cède le processeur (quantum expiré, entrée en attente, terminaison). Elle cherche le thread de plus haute priorité dans la ready queue locale du processeur courant.
- `KiSelectProcessor` — détermine le meilleur processeur logique pour exécuter un thread donné, en tenant compte de l'affinité, de la topologie NUMA, du cache, et de l'état idle des processeurs.

- `KiSwapContext` — effectue le context switch proprement dit : sauvegarde le contexte du thread sortant, charge le contexte du thread entrant, et bascule la pile noyau.
- `KiQuantumEnd` — appelée via DPC quand le quantum d'un thread expire. Décrémente la priorité si nécessaire et cherche un autre thread à exécuter.
- `KiDeferredReadyThread` — variante de `KiDispatcherReadyThread` utilisée pour les threads qui doivent être insérés dans la deferred ready list (traitée plus tard pour réduire la contention de verrou).

Le flux typique lors d'une interruption d'horloge est le suivant : le handler d'interruption d'horloge (`KeUpdateRunTime`) décrémente le quantum du thread courant. Si le quantum atteint zéro, un DPC est enqueué. Quand l'IRQL redescend de l'IRQL de l'interruption d'horloge à DISPATCH_LEVEL, les DPCs en attente sont drainés, et `KiQuantumEnd` est invoquée. Cette fonction vérifie s'il y a un thread de priorité égale ou supérieure dans la ready queue. Si oui, elle insère le thread courant en fin de queue (round-robin) et effectue un context switch vers le nouveau thread. Si non, le thread courant obtient un nouveau quantum et continue son exécution.

La structure KTHREAD en détail

La structure `KTHREAD` est le cœur du scheduler. Sur Windows 11 24H2, elle fait environ 0x480 octets (la taille exacte varie entre les builds). Voici les champs les plus significatifs pour le scheduling, avec leur rôle :

```
kd> dt nt!_KTHREAD -y State
   +0x120 State : UChar
   // 0=Initialized, 1=Ready, 2=Running, 3=Standby,
   // 4=Terminated, 5=Waiting, 6=Transition, 7=DeferredReady

kd> dt nt!_KTHREAD -y Priority
   +0x0dc Priority : Char         // Priorité courante (0-31)

kd> dt nt!_KTHREAD -y BasePriority
   +0x164 BasePriority : Char     // Priorité de base

kd> dt nt!_KTHREAD -y PriorityDecrement
   +0x165 PriorityDecrement : UChar // Boost restant à décroître

kd> dt nt!_KTHREAD -y Quantum
   +0x0de Quantum : Char          // Quantum restant (signé)

kd> dt nt!_KTHREAD -y QuantumTarget
   +0x020 QuantumTarget : Uint8B  // Cycles CPU cible pour ce quantum

kd> dt nt!_KTHREAD -y Affinity
   +0x1c8 Affinity : _GROUP_AFFINITY  // Affinité hard (masque)

kd> dt nt!_KTHREAD -y IdealProcessor
   +0x1d8 IdealProcessor : Uint4B  // Processeur idéal (soft affinity)

kd> dt nt!_KTHREAD -y TrapFrame
   +0x058 TrapFrame : Ptr64 _KTRAP_FRAME  // Frame d'interruption

kd> dt nt!_KTHREAD -y InitialStack
   +0x028 InitialStack : Ptr64 Void  // Sommet de la pile noyau

kd> dt nt!_KTHREAD -y StackLimit
   +0x030 StackLimit : Ptr64 Void    // Limite basse de pile

kd> dt nt!_KTHREAD -y KernelStack
   +0x038 KernelStack : Ptr64 Void   // Pointeur de pile actuel (sauvé lors du switch)

kd> dt nt!_KTHREAD -y WaitBlockList
   +0x170 WaitBlockList : Ptr64 _KWAIT_BLOCK  // Liste de blocs d'attente

kd> dt nt!_KTHREAD -y ApcState
   +0x098 ApcState : _KAPC_STATE     // État APC (process attaché, listes APC)

kd> dt nt!_KTHREAD -y ThreadListEntry
   +0x2e8 ThreadListEntry : _LIST_ENTRY  // Chaînage dans la liste de threads du processus
```

Quelques champs méritent une explication approfondie. Le champ `InitialStack` pointe vers le sommet de la pile noyau allouée au thread lors de sa création (typiquement 24 Ko sur x64). `KernelStack` contient le pointeur de pile sauvegardé lors du dernier context switch — c'est cette valeur qui est restaurée dans RSP quand le thread reprend l'exécution. La différence entre `InitialStack` et `KernelStack` indique la profondeur de pile utilisée au moment de la suspension.

Le champ `TrapFrame` pointe vers la structure `KTRAP_FRAME` qui contient les registres utilisateur sauvegardés lors de la transition vers le mode noyau (syscall ou interruption). C'est cette structure qui est restaurée par `iretq` ou `sysret` pour retourner en mode utilisateur. Elle contient les registres généraux (RAX-R15), le pointeur d'instruction (RIP), le pointeur de pile utilisateur (RSP), les flags (RFLAGS), et les sélecteurs de segment.

Le champ `ApcState` est crucial pour comprendre le concept d'attachement de processus. Normalement, un thread s'exécute dans l'espace d'adressage de son processus parent. Mais Windows permet à un thread de s'attacher temporairement à un autre processus (`KeStackAttachProcess`), ce qui change le répertoire de pages courant. L'`ApcState` garde une trace du processus auquel le thread est actuellement attaché, ce qui est essentiel pour le scheduler lors du context switch : si le thread entrant est attaché à un processus différent de celui du thread sortant, un changement d'espace d'adressage (rechargement de CR3) est nécessaire.

KPROCESS et EPROCESS

La structure `KPROCESS` est comparativement plus simple que `KTHREAD` dans sa contribution au scheduling. Ses champs clés sont :

```
kd> dt nt!_KPROCESS
   +0x000 Header           : _DISPATCHER_HEADER   // Pour WaitForSingleObject sur le
process
   +0x028 DirectoryTableBase : Uint8B              // CR3 — base du répertoire de pages
   +0x030 ThreadListHead   : _LIST_ENTRY          // Tête de la liste doublement chaînée
des KTHREAD
   +0x050 Affinity         : _KAFFINITY_EX        // Affinité par défaut pour les
nouveaux threads
   +0x108 BasePriority     : Char                 // Priorité de base du processus
   +0x109 QuantumReset     : Char                 // Quantum attribué aux threads de ce
processus
   +0x10c ActiveProcessors : _KAFFINITY_EX        // Bitmask des CPUs exécutant un thread
de ce processus
   +0x1b8 ProcessListEntry : _LIST_ENTRY          // Chaînage dans la liste globale des
processus
```

Le champ `DirectoryTableBase` contient la valeur chargée dans le registre CR3 lors d'un context switch vers un thread de ce processus. C'est le mécanisme fondamental de l'isolation des espaces d'adressage. Avec les mitigations Meltdown (KVAS — Kernel Virtual Address Shadow), il y a en réalité deux valeurs de CR3 : une pour l'espace noyau (avec le noyau mappé) et une pour l'espace utilisateur (avec seulement une page de trampoline noyau). Le scheduler doit gérer cette dualité lors du context switch.

Le champ `ActiveProcessors` est un bitmask mis à jour atomiquement à chaque context switch. Quand un thread d'un processus commence à s'exécuter sur un CPU, le bit correspondant est activé. Quand le dernier thread du processus quitte ce CPU, le bit est désactivé. Ce bitmask est utilisé pour les TLB shootdowns : quand l'espace d'adressage d'un processus est modifié (changement de mapping), un IPI est envoyé uniquement aux processeurs listés dans `ActiveProcessors` pour invalider leurs TLB.

KPRCB : la structure per-CPU

La structure `KPRCB` (Kernel Processor Control Block) est l'épicentre du scheduling par processeur. Chaque processeur logique possède sa propre instance de `KPRCB`, accessible via le registre GS en mode noyau (sur x64, GS:0x20 pointe vers le `KPCR` qui contient le `KPRCB`). Cette structure est volumineuse (plus de 0x10000 octets) et contient les files d'attente locales, les compteurs de performance, et les pointeurs vers les threads critiques.

```
kd> !prcb 0
PRCB for Processor 0 at fffff8034a080180:
Current IRQL -- 0
Threads--  Current ffffbd0843ae7080 Next 0000000000000000 Idle fffff8034a0dc840
Number 0 SetMember 1
Interrupt Count -- 0018e2f4
Times -- Dpc    00001a2e Interrupt 00001056
         Kernel 0005f1c8 User     0003a24b

kd> dt nt!_KPRCB fffff8034a080180
   +0x008 CurrentThread    : 0xffffbd08`43ae7080 _KTHREAD  // Thread en cours d'exécution
   +0x010 NextThread       : (null)                        // Prochain thread (standby)
   +0x018 IdleThread       : 0xfffff803`4a0dc840 _KTHREAD  // Thread idle de ce CPU
   +0x02c ReadySummary     : 0x100                         // Bitmask: quelles priorités
ont des threads prêts
   +0x040 DispatcherReadyListHead : [32] _LIST_ENTRY       // 32 listes de threads prêts
(une par priorité)
   +0x280 DpcData          : [2] _KDPC_DATA               // Files DPC (normale et
threaded)
   +0x5fc ContextSwitches  : 0x5a3b21                      // Compteur de context switches
```

Le triplet `CurrentThread` / `NextThread` / `IdleThread` résume l'état du scheduling sur un processeur donné. `CurrentThread` est le thread actuellement en exécution. `NextThread` est non-null quand un thread de priorité supérieure a été sélectionné pour préempter le thread courant mais que le context switch n'a pas encore eu lieu (il se produira au retour de l'interruption ou du DPC courant). `IdleThread` est le thread idle permanent de ce processeur — quand aucun thread prêt n'est disponible, c'est ce thread qui s'exécute et qui peut mettre le processeur en état d'économie d'énergie (C-state).

Le champ `ReadySummary` est un bitmask de 32 bits où chaque bit correspond à un niveau de priorité. Si le bit *n* est activé, cela signifie que `DispatcherReadyListHead[n]` contient au moins un thread prêt. Ce bitmask permet une recherche en O(1) du thread de plus haute priorité : une instruction `BSR` (Bit Scan Reverse) sur `ReadySummary` retourne immédiatement l'index du bit le plus significatif, donc la plus haute priorité avec un thread prêt.

Ready queues : 32 niveaux, recherche O(1)

L'organisation des ready queues est un modèle d'élégance algorithmique. Chaque processeur logique maintient un tableau de 32 listes doublement chaînées (`DispatcherReadyListHead[0..31]`), une par niveau de priorité. Chaque liste contient les threads prêts à ce niveau de priorité, chaînés via leur champ `WaitListEntry` dans `KTHREAD`. Les threads sont insérés en fin de liste (FIFO) lors de l'insertion dans la ready queue, et retirés en tête de liste lors de la sélection — ce qui implémente le round-robin intra-priorité.

Le bitmask `ReadySummary` est maintenu de manière cohérente avec les listes :

```
// Pseudo-code simplifié de l'insertion dans la ready queue
void KiInsertIntoReadyQueue(KPRCB *Prcb, KTHREAD *Thread) {
    UCHAR Priority = Thread->Priority;

    // Insérer le thread en fin de liste à sa priorité
    InsertTailList(&Prcb->DispatcherReadyListHead[Priority],
                   &Thread->WaitListEntry);

    // Activer le bit correspondant dans ReadySummary
    Prcb->ReadySummary |= (1UL << Priority);

    Thread->State = Ready;
}

// Pseudo-code simplifié de la sélection du thread de plus haute priorité
KTHREAD *KiSelectReadyThread(KPRCB *Prcb) {
    if (Prcb->ReadySummary == 0)
        return Prcb->IdleThread;  // Aucun thread prêt

    // BitScanReverse trouve le bit le plus haut
    ULONG HighestPriority;
    BitScanReverse(&HighestPriority, Prcb->ReadySummary);

    // Retirer le premier thread de la liste à cette priorité
    PLIST_ENTRY Entry = RemoveHeadList(
        &Prcb->DispatcherReadyListHead[HighestPriority]);

    KTHREAD *Thread = CONTAINING_RECORD(Entry, KTHREAD, WaitListEntry);

    // Si la liste est maintenant vide, désactiver le bit
    if (IsListEmpty(&Prcb->DispatcherReadyListHead[HighestPriority]))
        Prcb->ReadySummary &= ~(1UL << HighestPriority);

    Thread->State = Standby;
    return Thread;
}
```

La complexité de cette sélection est O(1) — elle ne dépend ni du nombre de threads dans le système, ni du nombre de threads prêts. L'instruction `BSR` (ou son équivalent intrinsèque `_BitScanReverse`) s'exécute en un cycle sur les processeurs modernes. C'est un facteur clé de la scalabilité du scheduler Windows.

En complément des ready queues locales (per-CPU), le scheduler maintient une deferred ready list. Quand un thread devient prêt en contexte d'un autre CPU (par exemple, un DPC sur le CPU 0 complète une I/O et réveille un thread dont le processeur idéal est le CPU 3), le thread n'est pas inséré directement dans la ready queue du CPU 3 (ce qui nécessiterait d'acquérir le spinlock du CPU 3). Au lieu de cela, il est placé dans la deferred ready list du CPU courant, et un IPI est envoyé au CPU cible pour le traiter. Ce mécanisme réduit considérablement la contention de verrou dans les configurations multi-cœur.

Évolution historique du verouillage

L'histoire du scheduler Windows est en grande partie l'histoire de l'élimination des verrous globaux au profit de structures per-CPU. Windows NT 3.1 utilisait un unique spinlock global pour le dispatcher (`KiDispatcherLock`). Tous les processeurs devaient acquérir ce verrou pour toute opération de scheduling — insertion dans la ready queue, sélection de thread, context switch.

Sur un système mono-processeur, cela ne posait pas de problème. Sur un multiprocesseur SMP, la contention sur ce verrou unique devenait le goulot d'étranglement principal au-delà de 4-8 processeurs.

Windows NT 4.0 a introduit des améliorations en ajoutant des spinlocks per-processeur pour certaines opérations, mais le `KiDispatcherLock` global restait nécessaire pour les opérations impliquant des threads sur plusieurs processeurs. Windows 2000 et XP ont apporté des optimisations incrémentales, notamment l'utilisation de queued spinlocks pour réduire le trafic de cache inter-CPU.

La révolution est venue avec Windows Vista/Server 2008, qui a fondamentalement restructuré le dispatcher en éliminant le `KiDispatcherLock` global au profit de verrous per-thread (`KTHREAD.ThreadLock`) et per-processeur. Chaque CPU possède désormais ses propres ready queues protégées par le spinlock du PRCB, et les opérations impliquant plusieurs CPUs utilisent des IPI et des listes déferred plutôt que des verrous partagés.

Windows 10 et 11 ont continué cette évolution avec le modèle hybride actuel : les opérations locales (sélection du prochain thread sur le CPU courant) n'acquièrent aucun verrou cross-CPU, tandis que les opérations globales (placement d'un thread sur un CPU distant, load balancing) utilisent des mécanismes lockless (interlocked operations) et des IPI ciblés. Le résultat est un scheduler qui scale linéairement jusqu'à des centaines de cœurs logiques, un prérequis pour les configurations Server 2025 avec des processeurs comme l'AMD EPYC 9654 (96 cœurs, 192 threads logiques).

> **À retenir :** Le scheduler Windows a évolué d'un modèle à verrou global unique (NT 3.1) vers un modèle per-CPU quasi-lockless (Windows 10+). Les ready queues sont locales à chaque processeur, avec un bitmask `ReadySummary` permettant une sélection O(1) via l'instruction `BSR`. La deferred ready list et les IPI remplacent les verrous partagés pour les opérations cross-CPU.

Algorithmes de Scheduling

Le scheduler Windows implémente un algorithme de scheduling à priorité fixe préemptif avec round-robin intra-priorité. Cette description simple masque une complexité considérable dans les détails d'implémentation, en particulier pour le load balancing multi-cœur, la gestion de l'affinité, et l'optimisation de la localité cache. Cette section décompose chaque aspect de l'algorithme.

Sélection du thread : la priorité l'emporte toujours

La règle fondamentale du scheduler Windows est absolue : le thread de plus haute priorité prêt à s'exécuter obtient le processeur. Il n'y a pas de notion de « fairness » pondérée comme dans le CFS (Completely Fair Scheduler) de Linux, pas de poids proportionnels, pas de virtual runtime. Si un thread à priorité 15 est constamment prêt, il monopolisera le processeur indéfiniment au détriment de tous les threads à priorité 14 et inférieure — seul le mécanisme d'anti-starvation du Balance Set Manager interviendra, et seulement après plusieurs secondes.

Ce choix de design est délibéré et a des conséquences profondes. Il rend le comportement du scheduler hautement prévisible pour les applications critiques : un thread audio à priorité 15 (via MMCSS) sait qu'il ne sera jamais préempté par un thread de compilation à priorité 8. Cette prévisibilité est essentielle pour les applications temps-réel « mou » (soft real-time) comme le rendu audio, la capture vidéo, ou les systèmes SCADA.

L'algorithme de sélection proprement dit est trivial grâce à la structure de données des ready queues :

```
// KiSearchForNewThread — version simplifiée
// Appelée quand le CPU courant a besoin d'un nouveau thread
KTHREAD *KiSearchForNewThread(KPRCB *CurrentPrcb) {
    ULONG Summary = CurrentPrcb->ReadySummary;

    if (Summary == 0) {
        // Pas de thread prêt sur ce CPU
        // Tenter de voler un thread d'un autre CPU (work stealing)
        KTHREAD *Stolen = KiStealReadyThread(CurrentPrcb);
        if (Stolen != NULL)
            return Stolen;
        return CurrentPrcb->IdleThread;
    }

    ULONG HighPri;
    _BitScanReverse(&HighPri, Summary);

    KTHREAD *Selected = DequeueHead(
        &CurrentPrcb->DispatcherReadyListHead[HighPri]);

    if (IsListEmpty(&CurrentPrcb->DispatcherReadyListHead[HighPri]))
        CurrentPrcb->ReadySummary &= ~(1UL << HighPri);

    return Selected;
}
```

Round-robin intra-priorité

Quand plusieurs threads de même priorité sont prêts à s'exécuter sur un même processeur, le scheduler les traite en round-robin : chaque thread reçoit un quantum complet, puis est replacé en fin de la file d'attente de sa priorité, et le thread suivant dans la file obtient son quantum. Ce comportement est une conséquence naturelle de l'organisation FIFO des ready queues : les threads sont insérés en fin de liste (`InsertTailList`) et retirés en tête (`RemoveHeadList`).

Le round-robin n'est pas configurable — il n'y a pas de notion de poids ou de parts CPU au sein d'un même niveau de priorité. Tous les threads à priorité 8 reçoivent exactement le même quantum (sauf le boost de premier plan). Si une application a besoin de plus de CPU qu'une autre au même niveau de priorité, elle doit soit créer plus de threads (ce qui lui donne proportionnellement plus de temps CPU total), soit augmenter sa priorité.

Cette limitation a motivé l'introduction de Job Objects avec des limites CPU (CPU rate control) et de la Quality of Service basée sur les CPU Sets dans les versions récentes de Windows. Ces mécanismes offrent un contrôle plus fin de l'allocation CPU sans manipuler les priorités, qui restent l'outil principal du scheduler.

Load balancing multi-core

Dans un système multi-cœur, le scheduler doit distribuer les threads prêts entre les processeurs disponibles. Windows utilise une approche mixte : placement proactif lors de la mise en ready queue, et vol de travail (work stealing) quand un processeur devient idle.

Quand un thread devient prêt (via `KiDispatcherReadyThread`), le scheduler doit choisir un processeur cible. L'algorithme `KiSelectProcessor` évalue les candidats dans l'ordre de préférence suivant :

1. **Processeur idéal** — le processeur vers lequel le thread a une affinité « douce ». Si ce processeur est idle, il est choisi immédiatement.
2. **Dernier processeur** — le processeur sur lequel le thread a couru en dernier (champ `LastProcessor` dans `KTHREAD`). Favorise la réutilisation du cache L1/L2.
3. **Processeur courant** — le processeur qui exécute le code de dispatch. Évite l'envoi d'un IPI.
4. **Processeur idle dans le même nœud NUMA** — préserve la localité mémoire.
5. **N'importe quel processeur idle** — dernier recours parmi les processeurs disponibles.
6. **Processeur exécutant le thread de plus basse priorité** — si aucun processeur n'est idle et que le nouveau thread a une priorité supérieure, il préempte le thread de plus basse priorité parmi les candidats autorisés par l'affinité.

Le work stealing intervient quand un processeur termine son thread courant et que sa ready queue locale est vide. Plutôt que de basculer immédiatement vers le thread idle, le scheduler tente de voler un thread prêt de la ready queue d'un autre processeur. La sélection du processeur victime suit la topologie : on vole en priorité aux processeurs partageant le même L2, puis le même L3, puis le même nœud NUMA, et enfin aux processeurs distants. Ce respect de la hiérarchie mémoire est crucial pour la performance des applications data-intensive.

```
// Pseudo-code simplifié du work stealing
KTHREAD *KiStealReadyThread(KPRCB *IdlePrcb) {
    // Parcourir les processeurs par proximité topologique
    for (each TargetPrcb in TopologicalOrder(IdlePrcb)) {
        if (TargetPrcb == IdlePrcb) continue;
        if (TargetPrcb->ReadySummary == 0) continue;

        // Acquérir le verrou de la ready queue cible
        AcquireSpinLock(&TargetPrcb->ReadyQueueLock);

        ULONG HighPri;
        if (_BitScanReverse(&HighPri, TargetPrcb->ReadySummary)) {
            // Vérifier que le thread volé accepte d'être exécuté sur IdlePrcb
            KTHREAD *Candidate = PeekHead(
                &TargetPrcb->DispatcherReadyListHead[HighPri]);

            if (Candidate->Affinity matches IdlePrcb->Number) {
                DequeueThread(TargetPrcb, Candidate);
                ReleaseSpinLock(&TargetPrcb->ReadyQueueLock);
                return Candidate;
            }
        }

        ReleaseSpinLock(&TargetPrcb->ReadyQueueLock);
    }
    return NULL;
}
```

Affinité : soft vs hard

L'affinité détermine sur quels processeurs logiques un thread est autorisé à s'exécuter. Windows distingue deux types d'affinité. L'affinité hard (dure) est un masque binaire stocké dans `KTHREAD.Affinity` — le thread ne peut jamais s'exécuter sur un processeur dont le bit correspondant est à zéro. Par défaut, l'affinité hard couvre tous les processeurs du système. L'API `SetThreadAffinityMask` permet de la restreindre. L'affinité soft (douce) est représentée par `KTHREAD.IdealProcessor` — c'est le processeur préféré du thread, celui que le scheduler essaiera de choisir en premier, mais sans garantie. Le thread peut s'exécuter sur n'importe quel processeur autorisé par l'affinité hard, l'ideal processor n'est qu'une préférence.

Sur les systèmes à plus de 64 processeurs logiques, l'affinité est exprimée via `GROUP_AFFINITY`, qui combine un numéro de groupe (0-19) et un masque de 64 bits au sein de ce groupe. Les API legacy (`SetThreadAffinityMask`) ne peuvent manipuler que le groupe courant ; les API modernes (`SetThreadGroupAffinity`) sont nécessaires pour le contrôle cross-groupe.

```
// Exemple : restreindre un thread aux cœurs 0-3 du groupe 0
GROUP_AFFINITY ga = {0};
ga.Group = 0;
ga.Mask = 0x0F;  // bits 0,1,2,3
SetThreadGroupAffinity(hThread, &ga, NULL);

// Exemple WinDbg : afficher l'affinité d'un thread
kd> !thread ffffbd0843ae7080
THREAD ffffbd0843ae7080  Cid 0004.0008  Teb: 0000000000000000 Win32Thread:
0000000000000000 RUNNING on processor 0
...
Affinity : Group 0, Mask = 0xffffffffffffffff
IdealProcessor : 2
```

Localité cache et sélection du processeur idéal

Le choix du processeur idéal est un facteur déterminant de la performance. Quand un thread est créé, le système lui assigne un processeur idéal via un algorithme de round-robin au sein du nœud NUMA du processus parent. Ce round-robin assure que les threads d'un même processus sont distribués de manière équilibrée entre les processeurs, évitant la concentration de tous les threads sur un seul cœur.

La préférence pour le dernier processeur exécuté (`LastProcessor`) est motivée par la localité cache. Quand un thread s'exécute sur un cœur, ses données de travail sont chargées dans les caches L1 et L2 de ce cœur. Si le thread est suspendu puis réveillé peu de temps après, il y a de bonnes chances que ses données soient encore dans le cache — à condition qu'il reprenne sur le même cœur. Le scheduler équilibre cette considération de cache hit avec la nécessité de ne pas laisser des processeurs idle : un thread sera migré vers un processeur idle même si cela implique un cache miss, car le coût d'un cache miss (~10-100 ns pour L2/L3) est largement inférieur au coût de laisser un processeur inutilisé pendant un quantum entier.

La topologie cache est découverte au démarrage via l'instruction CPUID et stockée dans les structures `KNODE` (nœuds NUMA) et `KPRCB`. Le scheduler connaît quels processeurs logiques partagent un cache L2 (typiquement les deux threads SMT d'un même cœur physique) et un cache L3 (typiquement tous les cœurs d'un même CCD/die). Cette information influence l'ordre de préférence dans le work stealing et le placement de threads.

Thread idle et C-states

Le System Idle Process (PID 0) est un processus spécial qui possède un thread idle par processeur logique. Quand aucun thread prêt n'est disponible pour un processeur, le thread idle de ce processeur est schedulé. Le thread idle exécute la routine `KiIdleLoop`, qui effectue plusieurs actions : drainer les DPC en attente, vérifier la deferred ready list, et si aucun travail n'est disponible, mettre le processeur en état d'économie d'énergie via l'instruction `MWAIT` (Monitor Wait) ou `HLT`.

Les C-states sont des états de puissance réduite définis par la spécification ACPI. C0 est l'état actif normal. C1 (Halt) arrête le pipeline mais maintient les caches. C2/C3 et au-delà peuvent vider les caches et réduire la fréquence d'horloge. Plus le C-state est profond, plus l'économie

d'énergie est importante, mais plus la latence de réveil est élevée. Le Power Manager de Windows, en coopération avec le scheduler, décide du C-state optimal en fonction de la charge prévue et des préférences d'alimentation configurées.

La latence de réveil d'un processeur en C-state profond peut être significative (10-100 µs pour C6), ce qui affecte directement la latence de dispatch des threads. C'est pourquoi le scheduler préfère réveiller un processeur en C1 plutôt qu'un processeur en C6 quand un thread de haute priorité devient prêt, même si le processeur en C6 serait topologiquement plus optimal. Cette décision est prise dans `KiSelectProcessor` via la consultation du `KPRCB.PowerState`.

Scheduler moderne (Windows 10, 11 et Server 2025)

Le scheduler de Windows 10 et ses successeurs représente une refonte significative par rapport au scheduler de Windows 7/8. Les évolutions sont motivées par trois facteurs convergents : l'explosion du nombre de cœurs logiques (256+ sur les serveurs haut de gamme), l'introduction des architectures hybrides Intel, et les exigences de sécurité liées à la virtualisation (VBS, Credential Guard). Cette section couvre les évolutions architecturales majeures du scheduler moderne.

Hybrid scheduling : queues locales et deferred ready list

Le modèle de scheduling de Windows 11 combine des ready queues strictement locales (per-CPU) avec une deferred ready list partagée. Ce modèle hybride élimine le besoin d'un verrou global tout en permettant le placement efficient de threads sur des CPUs distants. Quand un événement (complétion d'I/O, signal d'un objet de synchronisation) réveille un thread, le code exécuté sur le CPU courant ne tente pas d'acquérir le verrou de la ready queue du CPU cible. Au lieu de cela, il insère le thread dans une liste déferred et envoie un IPI (Inter-Processor Interrupt) au CPU cible. Le handler d'IPI sur le CPU cible traite la deferred ready list et insère le thread dans la ready queue locale. Ce schéma garantit que chaque ready queue n'est modifiée que par son propre CPU, éliminant la contention de verrou.

En pratique, l'implémentation est plus nuancée. Si le CPU cible est le CPU courant (le thread réveillé a son ideal processor sur le CPU qui exécute le code de réveil), l'insertion est directe sans IPI. Si le CPU cible est idle et en C-state léger, l'IPI est envoyé immédiatement. Si le CPU cible est en C-state profond, le scheduler peut choisir un CPU alternatif déjà actif pour éviter la latence de réveil.

Support NUMA

Les architectures NUMA (Non-Uniform Memory Access) sont omniprésentes dans les serveurs modernes. Un système à deux sockets AMD EPYC possède 2 nœuds NUMA, chacun avec son propre contrôleur mémoire et ses propres barrettes de RAM. L'accès à la mémoire locale est 2-3x plus rapide que l'accès à la mémoire distante (via l'interconnect Infinity Fabric ou UPI). Sur les processeurs AMD récents avec des CCD multiples, il peut y avoir 4, 8, ou même 16 nœuds NUMA par socket.

Le scheduler Windows est NUMA-aware depuis Windows Server 2003, mais les heuristiques ont été considérablement améliorées dans les versions récentes. Le principe fondamental est la localité mémoire : un thread devrait s'exécuter sur un processeur appartenant au même nœud NUMA que la majorité de sa mémoire allouée. Le champ `KTHREAD.IdealProcessor` est choisi au sein du nœud NUMA préféré du processus, et le work stealing respecte les frontières NUMA (vol intra-nœud avant vol inter-nœud).

L'API `KeQueryNodeActiveAffinity` permet au code noyau de déterminer quels processeurs appartiennent à un nœud NUMA donné. En user-mode, `GetNumaNodeProcessorMaskEx` fournit la même information. Les applications haute performance (bases de données, serveurs web) peuvent utiliser ces APIs pour allouer de la mémoire et placer des threads de manière NUMA-optimale.

Sur Windows Server 2025, les politiques NUMA sont plus agressives que sur le client. Le scheduler serveur pénalise davantage les migrations cross-NUMA et privilégie la localité mémoire au détriment de la réactivité. C'est un compromis approprié pour les workloads serveur (bases de données, serveurs d'application) où la bande passante mémoire est souvent le facteur limitant.

Processor Groups (>64 CPU)

L'architecture x86-64 de Windows utilise des masques d'affinité de 64 bits (`KAFFINITY`), ce qui limite à 64 le nombre de processeurs logiques adressables par un seul masque. Pour supporter les serveurs avec plus de 64 processeurs logiques, Windows a introduit les groupes de processeurs (Processor Groups) dans Windows 7 / Server 2008 R2. Chaque groupe contient au maximum 64 processeurs logiques, et un système peut avoir jusqu'à 20 groupes, soit un maximum théorique de 1280 processeurs logiques.

L'affinité d'un thread est exprimée comme un `GROUP_AFFINITY` : un couple (numéro de groupe, masque 64 bits). Par défaut, un thread est assigné au groupe 0 et peut s'exécuter sur n'importe quel processeur de ce groupe. Pour utiliser des processeurs d'autres groupes, l'application doit explicitement configurer l'affinité via `SetThreadGroupAffinity`.

Windows 11 24H2 et Server 2025 ont introduit des améliorations significatives au support multi-groupe. Le paramètre de registre `GroupAwareScheduling` permet au scheduler de placer automatiquement les threads sur le groupe optimal, sans que l'application ait besoin de gérer explicitement les groupes. L'API `GetActiveProcessorGroupCount` retourne le nombre de groupes actifs, et `GetActiveProcessorCount` retourne le nombre de processeurs dans un groupe donné.

```
// Énumérer les groupes de processeurs et leur topologie
WORD GroupCount = GetActiveProcessorGroupCount();
for (WORD g = 0; g < GroupCount; g++) {
    DWORD ProcCount = GetActiveProcessorCount(g);
    printf("Groupe %d : %d processeurs logiques\n", g, ProcCount);
}
```

Awareness cache et topologie

Le scheduler de Windows 11 intègre une connaissance fine de la topologie cache du processeur. Au démarrage, le noyau énumère la hiérarchie cache via CPUID leaf 4 (Intel) ou CPUID leaf 0x8000001D (AMD) et construit un modèle interne de la topologie. Ce modèle identifie quels processeurs logiques partagent chaque niveau de cache :

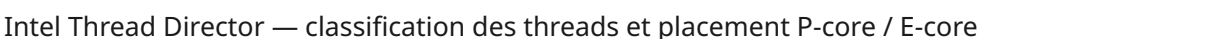
- **L1** — privé à chaque cœur logique (threads SMT sur le même cœur partagent le L1 instruction mais pas le L1 data sur certaines architectures)
- **L2** — typiquement partagé entre les deux threads SMT d'un même cœur physique (Intel), ou privé par cœur (AMD Zen)
- **L3** — partagé entre tous les cœurs d'un même CCD/die (AMD: 8 cœurs par CCD, Intel: variable)

La fonction `KiSelectProcessor` utilise cette topologie pour prendre des décisions de placement optimales. Quand plusieurs processeurs idle sont candidats, elle préfère celui qui partage le L2 ou L3 avec le dernier processeur exécuté par le thread. Quand aucun processeur idle n'est disponible, la topologie influence la décision de préemption : il vaut mieux préempter un thread de basse priorité sur un processeur partageant le L3 que sur un processeur distant.

Cette awareness topologique est particulièrement importante pour les architectures AMD Zen avec leurs CCD (Core Complex Dies) distincts. Un AMD EPYC 9654 a 12 CCD de 8 cœurs chacun, avec un L3 de 32 Mo par CCD. Un thread migré entre deux CCD perd la totalité de ses données en L3 (32 Mo de cache potentiellement inutilisé). Le scheduler de Windows Server 2025 est optimisé pour minimiser ces migrations cross-CCD.

CPU hybrides et Thread Director (Intel 12th Gen+)

L'introduction des architectures hybrides Intel (Alder Lake, 12ème génération, 2021) a imposé une refonte partielle du scheduler Windows. Pour la première fois dans l'histoire x86 grand public, un même package CPU contient des cœurs de performance inégale : les P-cores (Performance) et les E-cores (Efficiency) ont des microarchitectures différentes, des IPC différents, et des capacités différentes. Le scheduler ne peut plus traiter tous les processeurs logiques comme interchangeables.



Interaction entre le Thread Director Intel (HFI) et le scheduler Windows : classification IPC des threads et placement intelligent sur les cœurs Performance ou Efficient

Architecture P-core / E-core

Les P-cores (Performance cores) utilisent une microarchitecture haute performance : Golden Cove (Alder Lake), Raptor Cove (Raptor Lake), Lion Cove (Meteor Lake/Arrow Lake). Ils supportent l'Hyper-Threading (2 threads logiques par cœur), ont un pipeline large (6+ unités d'exécution), un cache L2 généreux (1.25-2 Mo), et sont optimisés pour l'IPC maximal sur un seul thread. Les E-cores (Efficiency cores) utilisent une microarchitecture plus compacte : Gracemont

(Alder Lake/Raptor Lake), Crestmont (Meteor Lake). Ils n'ont pas d'Hyper-Threading (1 thread par cœur), un pipeline plus étroit, un cache L2 partagé entre clusters de 4 cœurs, et sont optimisés pour le rapport performance/watt.

Sur un Core i9-13900K typique, il y a 8 P-cores (16 threads logiques) et 16 E-cores (16 threads logiques), soit 32 threads logiques au total. Les P-cores offrent environ 1.5-2x l'IPC des E-cores pour les charges mono-thread, mais les E-cores occupent beaucoup moins de surface silicium et consomment moins d'énergie. Cette asymétrie signifie que le placement d'un thread sur un P-core ou un E-core a un impact majeur sur la performance de ce thread.

Intel Thread Director et Hardware Feedback Interface

Intel Thread Director (ITD) est un composant matériel intégré dans les processeurs hybrides à partir d'Alder Lake. Il monitore en continu le comportement de chaque thread en exécution — type d'instructions (entier, flottant, vectoriel, mémoire), taux d'IPC, activité cache — et classifie le workload du thread en catégories. Cette classification est communiquée au système d'exploitation via la Hardware Feedback Interface (HFI), une table en mémoire partagée entre le firmware et l'OS.

La table HFI contient, pour chaque cœur logique, deux métriques par classe de workload : la capacité de performance (0-255) et la capacité d'efficacité énergétique (0-255). Un P-core aura typiquement une capacité de performance élevée (200+) et une capacité d'efficacité modérée (100-150). Un E-core aura une capacité de performance modérée (100-150) mais une capacité d'efficacité élevée (200+). Ces valeurs sont dynamiques et peuvent changer en fonction de la température (throttling thermique), de l'état de puissance, et du workload détecté.

Le système d'exploitation lit la table HFI via l'interface `GUID_CLASS` et utilise ces informations dans le scheduler pour prendre des décisions de placement. La fonction `KiSelectProcessor` intègre les capacités HFI dans son algorithme de sélection : un thread classifié comme « compute-heavy » (IPC élevé, instructions vectorielles) sera orienté vers un P-core, tandis qu'un thread classifié comme « background » (faible IPC, beaucoup d'I/O waits) sera orienté vers un E-core.

```
// Pseudo-code simplifié de l'intégration HFI dans le scheduler
ULONG KiSelectProcessor_Hybrid(KTHREAD *Thread) {
    UCHAR QosClass = Thread->QosLevel;

    if (QosClass == THREAD_QOS_CLASS_HIGH_PERFORMANCE) {
        // Chercher un P-core idle ou le P-core le moins chargé
        return SelectFromPCores(Thread);
    }
    else if (QosClass == THREAD_QOS_CLASS_ECO) {
        // Forcer sur E-core (EcoQoS / Efficiency Mode)
        return SelectFromECores(Thread);
    }
    else {
        // Classe par défaut : utiliser les recommandations HFI
        HFI_ENTRY *Entry = GetHfiEntryForThread(Thread);
        if (Entry->PerformanceCapability > THRESHOLD_PCORE)
            return SelectFromPCores(Thread);
        else
            return SelectFromECores(Thread);
    }
}
```

Heuristiques de placement

Le placement des threads sur les cœurs hybrides est une heuristique complexe qui intègre plusieurs signaux. La classification QoS du thread (définie par l'API `SetThreadInformation` avec `THREAD_POWER_THROTTLING_STATE`) est le signal le plus fort : un thread marqué EcoQoS est systématiquement orienté vers les E-cores avec une fréquence réduite. Le Multimedia Class Scheduler Service (MMCSS) peut marquer les threads audio/vidéo comme haute priorité, les orientant vers les P-cores.

En l'absence de classification explicite, le Thread Director utilise l'analyse comportementale. Les threads exécutant des instructions AVX-512 ou AVX2 intensives sont classifiés comme « heavy compute » et orientés vers les P-cores (les E-cores ne supportent pas AVX-512 et exécutent AVX2 moins efficacement). Les threads passant la majorité de leur temps en attente d'I/O sont classifiés comme « light » et orientés vers les E-cores. Les threads avec un profil mixte sont placés dynamiquement — ils peuvent migrer entre P-cores et E-cores selon l'évolution de leur workload.

Windows 11 utilise également le concept de « Favored Processor » pour les threads du premier plan. Les threads de l'application active sont marqués comme favoris et bénéficient d'une préférence pour les P-cores, indépendamment de leur classification comportementale. Ce mécanisme assure que l'application avec laquelle l'utilisateur interagit bénéficie de la meilleure performance disponible.

Impact sur performance et latence

L'impact du Thread Director sur la performance dépend fortement du workload. Pour les applications bien intégrées (Edge, Chrome récent, applications UWP), qui utilisent les APIs QoS pour classifier leurs threads, le résultat est excellent : les threads de rendu sont sur les P-cores,

les threads de service sont sur les E-cores, et la consommation d'énergie est optimisée sans compromettre la performance perçue. Pour les applications non-adaptées (anciens jeux, logiciels métier legacy), le résultat peut être décevant voire problématique.

Un cas classique de régression de performance est un jeu utilisant un pool de worker threads sans classification QoS. Le Thread Director peut classifier certains de ces workers comme « légers » (par exemple, pendant une phase d'attente de synchronisation) et les migrer sur des E-cores. Quand le workload de ces threads change soudainement (début d'une phase de calcul intensive), il y a un délai de quelques millisecondes avant que le Thread Director détecte le changement et recommande une migration vers un P-core. Ce délai se traduit par un micro-stutter perceptible.

Cas d'erreurs de scheduling et contournements

Les erreurs de scheduling hybride sont suffisamment courantes pour avoir engendré un écosystème d'outils de contournement. Process Lasso est un utilitaire tiers populaire qui permet de forcer l'affinité de processus spécifiques vers les P-cores ou les E-cores. L'utilitaire intégré de Windows, le Task Manager, permet depuis Windows 11 22H2 de définir le « mode d'efficacité » (EcoQoS) par processus.

Les développeurs de jeux ont commencé à implémenter explicitement la classification QoS de leurs threads. Unreal Engine 5 et Unity identifient le thread de rendu principal et le thread de jeu comme « high performance », tandis que les threads de chargement d'assets et de streaming sont marqués « eco ». Cette approche proactive résout les problèmes à la source mais nécessite une modification du code applicatif.

Un problème plus insidieux concerne les threads à priorité temps réel (16-31) sur les architectures hybrides. Le scheduler peut placer un thread temps réel sur un E-core si le P-core est occupé par un autre thread temps réel. Ce placement est correct du point de vue de la priorité (le thread obtient immédiatement du CPU), mais la performance IPC sur le E-core peut être 40-50% inférieure, ce qui crée une latence inattendue pour les applications sensibles. La solution est le pinning explicite par affinité sur les P-cores, mais cela nécessite une connaissance de la topologie matérielle.

> **À retenir :** Les architectures CPU hybrides (P-core/E-core) ajoutent une dimension de complexité majeure au scheduling. Intel Thread Director fournit des recommandations matérielles au scheduler, mais les heuristiques ne sont pas infaillibles. Les applications critiques doivent classifier explicitement leurs threads via les APIs QoS de Windows 11 pour garantir un placement optimal. Les développeurs de logiciels sensibles à la latence (jeux, audio temps réel) doivent tester spécifiquement sur des architectures hybrides.

Différences Windows 11 vs Server 2025

Windows 11 24H2 et Windows Server 2025 partagent le même noyau (build 26100) et donc le même code de scheduler. Les différences de comportement proviennent de paramètres de configuration distincts, de services système différents, et de priorités de design opposées. Comprendre ces différences est essentiel pour un consultant qui intervient à la fois sur des postes clients et des serveurs en production.

Objectifs de design

Le profil client (Windows 11) est optimisé pour la réactivité de l'interface utilisateur. L'objectif est que le clic d'un utilisateur produise une réponse visible en moins de 100 ms, que la lecture audio ne subisse pas de glitch, et que l'application au premier plan se sente « rapide » même si le système est chargé en arrière-plan. Pour atteindre ces objectifs, le scheduler client utilise des quantums courts (réaction rapide aux changements de priorité), des boosts agressifs pour le premier plan, et des mécanismes comme MMCSS et EcoQoS pour différencier les workloads.

Le profil serveur (Server 2025) est optimisé pour le débit global et l'équité entre services. Sur un serveur, il n'y a typiquement pas d'interface graphique interactive. L'objectif est de maximiser le nombre de transactions par seconde, de requêtes HTTP servies, ou de machines virtuelles supportées. Les quantums longs réduisent l'overhead de context switching (chaque switch coûte 2-5 µs de travail improductif). L'absence de boost de premier plan assure que tous les services de même priorité reçoivent un traitement équitable.

Quantum et latence

La différence de quantum est probablement la distinction la plus impactante entre les deux profils. Le quantum par défaut du client est de 2 intervalles d'horloge (environ 31.25 ms), variable selon le statut foreground/background. Le quantum par défaut du serveur est de 12 intervalles d'horloge (environ 187.5 ms), fixe pour tous les threads.

Paramètre	Windows 11 (Client)	Server 2025
Quantum par défaut	2 ticks (~31 ms), variable	12 ticks (~187 ms), fixe
Win32PrioritySeparation	0x26 (court, variable, boost max)	0x18 (long, fixe, pas de boost)
Boost foreground quantum	3x (PspForegroundQuantum[2])	1x (pas de boost)
Boost foreground priorité	+2 niveaux	Désactivé
Boost I/O completion	Actif	Actif (identique)
Anti-starvation	Actif (4 sec)	Actif (4 sec, identique)
Dynamic Tick	Actif	Actif
MMCSS	Actif	Disponible mais rarement utilisé
EcoQoS	Actif	Disponible

Le paramètre `Win32PrioritySeparation` est le levier principal pour modifier le comportement du quantum. On peut configurer un serveur avec le comportement client (ou inversement) en modifiant cette valeur de registre. C'est parfois fait sur des serveurs RDS (Remote Desktop Services) hébergeant des sessions utilisateur interactives, où la réactivité de l'interface est plus importante que le débit brut.

```
:: Configurer un Server 2025 avec le comportement quantum client
reg add "HKLM\SYSTEM\CurrentControlSet\Control\PriorityControl" /v
Win32PrioritySeparation /t REG_DWORD /d 0x26 /f

:: Vérifier la valeur actuelle
reg query "HKLM\SYSTEM\CurrentControlSet\Control\PriorityControl" /v
Win32PrioritySeparation

:: En WinDbg, vérifier le quantum actuel d'un thread
kd> dt nt!_KPROCESS ffffbd0840000000 QuantumReset
   +0x109 QuantumReset : 36    // 36 quantum units = quantum long (serveur)
```

Stratégies de priorité dynamique

Sur le client Windows 11, le boost de priorité du premier plan est implémenté par le sous-système Win32 (win32k.sys). Quand une fenêtre passe au premier plan via `SetForegroundWindow`, le sous-système appelle `KeSetPriorityThread` pour augmenter la priorité de base de tous les threads du processus de +2 (configurable via `Win32PrioritySeparation` bits 1-0). Simultanément, le quantum de ces threads est multiplié par le facteur de boost (jusqu'à 3x). Ce double boost (priorité + quantum) donne une avantage substantiel à l'application au premier plan.

Le Multimedia Class Scheduler Service (MMCSS) est un service Windows qui fonctionne comme un intermédiaire entre les applications multimédia et le scheduler. Une application audio peut appeler `AvSetMmThreadCharacteristics("Pro Audio", ...)` pour enregistrer son thread auprès de MMCSS. Le service élève périodiquement la priorité du thread à un niveau garanti (typiquement 26, dans la plage temps réel) pendant la durée de la période de traitement audio. Cela garantit que le thread audio obtient du CPU avec une latence minimale, même si d'autres threads de haute priorité sont actifs.

Sur Server 2025, ces mécanismes de boost client sont désactivés ou atténués. Il n'y a pas de notion de « fenêtre au premier plan » sur un serveur headless. MMCSS est installé mais rarement utilisé. En revanche, le serveur bénéficie de mécanismes spécifiques : la gestion des priorités I/O est plus sophistiquée pour les workloads de base de données et de stockage, et le scheduler intègre des heuristiques pour les workloads de machine virtuelle (détection de VP spin-wait).

EcoQoS : le mode efficacité de Windows 11

EcoQoS (Quality of Service Eco) est une fonctionnalité introduite dans Windows 11 qui permet au système et aux applications de déclarer que certains threads sont « non-critiques » et peuvent être exécutés à performance réduite en échange d'une économie d'énergie. Le mécanisme utilise l'API `SetProcessInformation` ou `SetThreadInformation` avec la classe `ProcessPowerThrottling` et le flag `PROCESS_POWER_THROTTLING_EXECUTION_SPEED`.

Quand un thread est marqué EcoQoS, le scheduler applique plusieurs effets combinés : le thread est orienté vers les E-cores sur les architectures hybrides, la fréquence du cœur est réduite (via le power manager), et le thread peut recevoir un quantum réduit. Le résultat est une réduction significative de la consommation d'énergie (15-30% selon les workloads) au prix d'une performance réduite pour les threads affectés.

Windows 11 applique automatiquement EcoQoS à certains scénarios : les onglets d'arrière-plan dans Edge/Chrome, les processus marqués « Efficiency Mode » dans le Task Manager, et les applications déclarées comme « background work » via le Package Manager (UWP). Le Task Manager affiche une icône de feuille verte pour les processus en mode EcoQoS.

```
// Marquer un thread comme EcoQoS en C
THREAD_POWER_THROTTLING_STATE state = {0};
state.Version = THREAD_POWER_THROTTLING_CURRENT_VERSION;
state.ControlMask = THREAD_POWER_THROTTLING_EXECUTION_SPEED;
state.StateMask = THREAD_POWER_THROTTLING_EXECUTION_SPEED;

SetThreadInformation(
    GetCurrentThread(),
    ThreadPowerThrottling,
    &state,
    sizeof(state)
);
```

Optimisations spécifiques au serveur

Windows Server 2025 intègre des optimisations de scheduling spécifiques aux workloads serveur. Le scheduling NUMA-aware est plus conservateur : le scheduler évite plus fortement les migrations cross-NUMA, même si cela signifie un déséquilibre temporaire de charge entre les nœuds. C'est le bon compromis pour les bases de données qui allouent leurs buffer pools sur un nœud NUMA spécifique via `VirtualAllocExNuma`.

L'affinité d'interruption pour les cartes réseau (RSS — Receive Side Scaling) est intégrée au scheduler. Les interruptions réseau sont distribuées entre les processeurs via RSS, et le scheduler tente de placer les threads applicatifs qui traitent les données réseau sur les mêmes processeurs que les interruptions RSS correspondantes. Cela maximise la localité cache : les données reçues par la carte réseau sont dans le cache L2 du processeur qui a traité l'interruption, et si le thread applicatif tourne sur le même processeur, il bénéficie d'un cache hit.

Pour les scénarios de virtualisation, le scheduler serveur implémente la détection de VP spin-wait : quand un thread de virtual processor (VP) d'une VM effectue un spin-wait (attente active sur un spinlock dans la VM invitée), le scheduler de l'hôte peut détecter ce pattern via les compteurs de performance matériels et céder le CPU à un autre VP, réduisant ainsi le gaspillage CPU dans les scénarios de VM overcommit. La commande `Get-VMProcessor` en PowerShell permet de voir et configurer les paramètres de scheduling des VPs.

Virtualisation et Hypervisor Scheduling

La virtualisation ajoute une couche supplémentaire de scheduling qui interagit de manière complexe avec le scheduler du système d'exploitation invité. Sur Windows Server 2025 et Windows 11, Hyper-V est un hyperviseur Type-1 (bare-metal) qui s'exécute directement sur le matériel, avec le système d'exploitation « hôte » (root partition) tournant comme une machine virtuelle privilégiée. Comprendre cette interaction est essentiel pour le dimensionnement des serveurs de virtualisation et pour l'analyse des problèmes de performance dans les environnements Active Directory virtualisés.

Interaction avec Hyper-V

Hyper-V implémente un hyperviseur de Type-1, ce qui signifie que le code de l'hyperviseur (`hvix64.exe` ou `hvax64.exe`) s'exécute à un niveau de privilège supérieur au noyau Windows. L'hyperviseur gère la mémoire physique (via SLAT/EPT), les interruptions (via le virtual APIC), et le scheduling des processeurs virtuels (VPs). Le noyau Windows de la root partition n'a pas d'accès direct au matériel physique — il communique avec l'hyperviseur via des hypercalls.

Le scheduling dans un environnement Hyper-V se fait à deux niveaux. Le scheduler de l'hyperviseur décide quel VP (de quelle VM) s'exécute sur quel processeur logique physique (LP). Le scheduler du système d'exploitation invité (dans chaque VM) décide quel thread s'exécute sur quel VP. Le scheduler invité n'a aucune visibilité sur les décisions de l'hyperviseur — il « croit » avoir accès à des processeurs physiques dédiés, alors qu'en réalité ses VPs sont multiplexés avec ceux d'autres VMs sur les mêmes LPs physiques.

L'enlightenment du scheduler est un mécanisme par lequel l'hyperviseur informe le système d'exploitation invité de certaines conditions de scheduling. Un invité Windows « enlightened » sait qu'il tourne dans une VM et peut adapter son comportement : par exemple, au lieu de faire un spin-wait agressif sur un spinlock, il effectue un `HvCallNotifyLongSpinWait` hypercall qui permet à l'hyperviseur de céder le VP à une autre VM. Cet enlightenment est critique pour la performance des VMs overcommitées.

Root Scheduler vs Core Scheduler

Hyper-V supporte trois types de scheduler, sélectionnables au démarrage. Le Classic Scheduler (legacy) : l'hyperviseur a son propre scheduler qui gère directement les VPs. Le Core Scheduler (par défaut depuis Server 2019) : l'hyperviseur schedule les VPs en respectant les frontières de cœurs physiques — deux VPs de VMs différentes ne partagent jamais le même cœur physique SMT. Le Root Scheduler (par défaut sur Server 2025 dans certaines configurations) : l'hyperviseur délègue le scheduling des VPs au scheduler du noyau Windows de la root partition.

Le Core Scheduler a été introduit principalement pour des raisons de sécurité. Les attaques par side-channel SMT (MDS, L1TF, Portsmash) exploitent le partage de ressources microarchitecturales (buffers de chargement, cache L1, ports d'exécution) entre les deux threads logiques d'un même cœur physique. Si deux VPs de VMs différentes sont co-schedulés sur le même cœur physique, une VM malveillante peut extraire des données de l'autre VM via ces side-channels. Le Core Scheduler élimine ce vecteur en garantissant que les deux threads SMT d'un même cœur appartiennent toujours à la même VM (ou sont idle).

```
# Vérifier le type de scheduler Hyper-V actif
Get-WinEvent -FilterHashtable @{
    LogName='Microsoft-Windows-Hyper-V-Hypervisor-Operational'
    Id=2
} -MaxEvents 1 | Format-List Message

# Configurer le type de scheduler (nécessite un redémarrage)
# 0x01 = Classic, 0x02 = Core, 0x04 = Root
bcdedit /set hypervisorschedulertype Core

# En WinDbg connecté au hyperviseur (hvix64)
# Afficher les VP assignés à chaque LP
!hv vp
```

Le Root Scheduler représente l'approche la plus récente. Au lieu que l'hyperviseur implémente son propre algorithme de scheduling, il présente les VPs des VMs invitées comme des threads spéciaux dans la root partition. Le scheduler Windows standard de la root partition schedule alors ces VP-threads exactement comme des threads normaux, avec les mêmes mécanismes de priorité, d'affinité, et de load balancing. L'avantage est que toutes les optimisations du scheduler Windows (NUMA awareness, topologie cache, Thread Director) bénéficient automatiquement aux VMs. L'inconvénient est une latence légèrement supérieure pour les opérations de scheduling (hypercall overhead).

Scheduling des vCPU et overcommit

Le scheduling des processeurs virtuels (VPs) par l'hyperviseur est fondamentalement similaire au scheduling de threads par le noyau : chaque VP a un état (Running, Ready, Waiting), une priorité relative, et un quantum. Cependant, les implications de l'overcommit (plus de VPs actifs que de LPs physiques) sont bien plus sévères dans le contexte de la virtualisation.

Quand un VP est « deschedulé » (préempté par l'hyperviseur pour donner le LP physique à un autre VP), le système d'exploitation invité n'en est pas conscient. Du point de vue de l'invité, son « processeur » a simplement cessé de fonctionner pendant un certain temps, comme si une interruption de durée anormale s'était produite. Cela peut avoir des effets néfastes : les spinlocks dans l'invité tournent inutilement (le thread attend un lock détenu par un VP qui a été deschedulé), les timers expirent de manière imprévisible, et les compteurs de performance sont faussés.

L'overcommit de vCPU est une source majeure de problèmes de performance dans les environnements de virtualisation. Un ratio de 4:1 (4 VPs par LP physique) est généralement considéré comme la limite supérieure acceptable pour des workloads de bureau, et 2:1 pour des workloads serveur. Au-delà, les effets de « VP starvation » deviennent perceptibles : latence accrue, jitter dans les timers, et dans les cas extrêmes, des timeouts réseau ou des échecs de clustering.

SMT awareness et sécurité

La gestion du Simultaneous Multi-Threading (SMT, appelé Hyper-Threading par Intel) dans le contexte de la virtualisation est un sujet de sécurité critique. Les vulnérabilités matérielles MDS (Microarchitectural Data Sampling), L1TF (L1 Terminal Fault), et TAA (TSX Asynchronous Abort) permettent à un thread s'exécutant sur un thread logique d'un cœur physique de lire des données appartenant au thread s'exécutant sur l'autre thread logique du même cœur.

Dans un contexte de virtualisation, si une VM malveillante est co-schedulée sur le même cœur physique qu'une VM victime (chacune utilisant un thread logique), l'attaquant peut potentiellement lire des données sensibles (clés cryptographiques, tokens d'authentification) de la VM victime. Cette attaque est pratique et a été démontrée dans des conditions réalistes.

Le Core Scheduler de Hyper-V est la mitigation principale. En s'assurant que les deux threads SMT d'un même cœur appartiennent toujours à la même VM, il élimine le vecteur d'attaque cross-VM SMT. Le coût est une réduction de la flexibilité de scheduling et potentiellement du débit global, car un LP ne peut pas être utilisé par une VM différente tant que l'autre LP du même cœur est occupé par la première VM.

L'alternative radicale est de désactiver SMT entièrement (`bcdedit /set hypervisorsmt disabled`), ce qui élimine le vecteur mais divise le nombre de threads logiques par deux. Cette approche est parfois recommandée pour les environnements de haute sécurité (hébergement multi-tenant, infrastructure financière) où le coût de performance est acceptable au regard du risque.

Impact de VBS sur le scheduling

Virtualization-Based Security (VBS) utilise l'hyperviseur Hyper-V pour créer un environnement d'exécution isolé (Virtual Secure Mode, VSM) au sein même de la root partition ou d'une VM. Credential Guard stocke les hashes NTLM et les tickets Kerberos dans un processus isolé (`lsaiso.exe`) s'exécutant dans VSM, inaccessible même au noyau Windows. HVCI (Hypervisor-enforced Code Integrity) utilise l'hyperviseur pour vérifier que tout le code chargé en mode noyau est signé.

L'impact de VBS sur le scheduling est mesurable mais souvent surestimé. Chaque transition entre le mode normal (VTL 0) et le mode sécurisé (VTL 1) implique un VMEXIT/VMENTER, avec un coût typique de 1-3 µs. Pour les appels système normaux, cet overhead n'est pas dans le chemin critique. Cependant, pour les opérations fréquentes qui touchent des structures protégées par HVCI (chargement de drivers, modification de structures critiques du noyau), l'overhead peut s'accumuler. Les mesures de Microsoft indiquent un impact de 5-15% sur les benchmarks synthétiques de context switching, mais un impact <5% sur les workloads applicatifs réels.

Pour les consultants en sécurité et persistence, VBS représente un changement de paradigme : les techniques de persistence qui modifient des structures noyau (SSDT hooking, callback modification) sont bloquées par HVCI. Le scheduler lui-même bénéficie de cette protection : les structures `KTHREAD` et `KPRCB` ne peuvent pas être modifiées arbitrairement par un rootkit noyau quand HVCI est actif.

À retenir : Hyper-V ajoute un niveau de scheduling supplémentaire entre les VPs et les processeurs physiques. Le Core Scheduler garantit l'isolation SMT entre VMs (mitigation MDS/L1TF). Le Root Scheduler délègue le scheduling des VPs au noyau Windows de la root partition. L'overcommit de vCPU est la source la plus fréquente de problèmes de performance dans les environnements virtualisés. VBS/HVCI ajoutent un overhead mesurable mais acceptable sur le scheduling.

Cycle de vie d'un Thread (Deep Dive)

Le cycle de vie d'un thread Windows, de sa création à sa terminaison, traverse un automate à états finis complexe dont la compréhension est essentielle pour diagnostiquer les problèmes de performance et de synchronisation. Cette section détaille chaque phase du cycle de vie, depuis l'appel système de création jusqu'au context switch et à l'expiration du quantum.

Machine à états d'un thread Windows — les 8 états et transitions

Cycle de vie complet : Initialized → Ready → Standby → Running → Waiting/Terminated, avec les fonctions noyau déclenchant chaque transition

Création d'un thread

La création d'un thread commence en user-mode par un appel à `CreateThread` (Win32) ou `RtlCreateUserThread` (ntdll), qui se traduit par l'appel système `NtCreateThreadEx`. Ce syscall transite vers le noyau où la fonction `PspCreateThread` orchestre la création :

1. **Allocation de ETHREAD/KTHREAD** — une structure `ETHREAD` (qui contient `KTHREAD` comme premier membre) est allouée depuis le pool non-paginé. La taille est d'environ 0x600 octets sur x64.
2. **Allocation de la pile noyau** — une pile noyau de 24 Ko (par défaut sur x64) est allouée. `InitialStack` est configuré au sommet de cette pile, `StackLimit` à la base, et `StackBase` est calculé.
3. **Initialisation des champs KTHREAD** — priorité de base (héritée du processus), quantum (hérité du processus), affinité (héritée du processus), processeur idéal (round-robin dans le nœud NUMA), état initial (`Initialized`).
4. **Configuration de la pile initiale** — une frame de context switch est poussée sur la pile noyau, simulant un context switch fictif. Le pointeur de retour est configuré pour pointer vers `KiThreadStartup`, la routine de démarrage de thread.
5. **Enregistrement de l'APC de démarrage** — un APC (Asynchronous Procedure Call) noyau est enqueué pour le thread. Cet APC invoquera `PspUserThreadStartup` quand le thread obtiendra son premier quantum. `PspUserThreadStartup` configurera le contexte utilisateur et effectuera le retour en mode utilisateur vers le point d'entrée du thread.
6. **Insertion dans la ready queue** — `KiDispatcherReadyThread` est appelée pour placer le thread dans l'état `Ready` (ou `DeferredReady`) sur le processeur sélectionné.

```
// Séquence simplifiée de PspCreateThread
NTSTATUS PspCreateThread(
    PEPROCESS Process,
    PVOID StartAddress,
    PVOID Parameter,
    ...)
{
    PETHREAD Thread;

    // 1. Allouer la structure ETHREAD
    Thread = ObCreateObject(PsThreadType, sizeof(ETHREAD));

    // 2. Initialiser KTHREAD
    KeInitializeThread(&Thread->Tcb,  // KTHREAD
                       KernelStack,
                       PspUserThreadStartup,  // SystemRoutine
                       StartAddress,          // StartRoutine
                       Parameter,
                       NULL,                  // TrapFrame
                       Process);

    // 3. Hériter la priorité et le quantum du processus
    Thread->Tcb.BasePriority = Process->Pcb.BasePriority;
    Thread->Tcb.Quantum = Process->Pcb.QuantumReset;

    // 4. Sélectionner le processeur idéal (round-robin)
    Thread->Tcb.IdealProcessor =
        KeSelectIdealProcessor(&Process->Pcb);

    // 5. Insérer dans la liste de threads du processus
    InsertTailList(&Process->Pcb.ThreadListHead,
                   &Thread->Tcb.ThreadListEntry);

    // 6. Rendre le thread prêt
    KiDispatcherReadyThread(&Thread->Tcb);

    return STATUS_SUCCESS;
}
```

Un aspect subtil mais important : le thread ne commence pas à exécuter le code utilisateur immédiatement après `NtCreateThreadEx`. Il est placé dans la ready queue, et ce n'est que lorsque le scheduler lui alloue un quantum (potentiellement sur un autre CPU) qu'il exécute `KiThreadStartup` → `PspUserThreadStartup` → retour en user-mode → exécution de la routine de démarrage. Ce délai entre la création et la première exécution est observable via ETW (événement Thread Start) et peut varier de quelques microsecondes (processeur idle disponible) à plusieurs dizaines de millisecondes (tous les processeurs occupés par des threads de priorité supérieure).

États d'un thread

L'automate à états d'un thread Windows comporte huit états :

État	Valeur	Description	Transitions possibles
Initialized	0	Thread créé mais pas encore inséré dans la ready queue	→ DeferredReady, Ready
Ready	1	Prêt à s'exécuter, dans la ready queue d'un processeur spécifique	→ Standby
Running	2	En cours d'exécution sur un processeur logique	→ Ready (preemption), Waiting, Terminated
Standby	3	Sélectionné comme prochain thread pour un processeur (KPRCB.NextThread)	→ Running
Terminated	4	Thread terminé, en attente de nettoyage	→ (destruction)
Waiting	5	En attente d'un objet de synchronisation (mutex, event, semaphore, timer)	→ Ready, DeferredReady
Transition	6	Prêt mais pile noyau paginée (en cours de chargement)	→ Ready
DeferredReady	7	Prêt mais pas encore assigné à un processeur spécifique	→ Ready

L'état `Standby` est un état transitoire critique. Quand `KiSearchForNewThread` sélectionne un thread dans la ready queue, il le passe en état `Standby` et le stocke dans `KPRCB.NextThread`. Le thread reste en `Standby` jusqu'à ce que le context switch soit effectivement réalisé, à quel point il passe en `Running`. Si, entre la sélection et le context switch, un thread de priorité encore supérieure arrive, le thread en `Standby` peut être replacé en `Ready` (une situation rare mais possible).

L'état `DeferredReady` est spécifique au modèle multi-processeur. Quand un thread devient prêt à cause d'un événement sur un CPU différent de son processeur cible, il est placé en `DeferredReady` dans la deferred ready list du CPU courant. Le thread sera traité ultérieurement pour être inséré dans la ready queue du CPU cible (état `Ready`). Cet état intermédiaire permet d'éviter d'acquérir le verrou de la ready queue d'un autre CPU dans un contexte potentiellement critique (DPC).

L'état `Transition` est rare mais important. Il se produit quand un thread en état `Waiting` est réveillé mais que sa pile noyau a été paginée (swappée sur disque). Le thread passe en `Transition` le temps que le gestionnaire de mémoire charge sa pile noyau depuis le page file. Une fois la pile chargée, le thread passe en `Ready`. La pagination de la pile noyau est un mécanisme d'économie de mémoire qui s'active quand le système est sous pression mémoire — les threads en attente prolongée (minutes/heures) sont de bons candidats pour la pagination de pile.

```
// Inspecter l'état d'un thread dans WinDbg
kd> !thread ffffbd0843ae7080
THREAD ffffbd0843ae7080  Cid 0004.0008  Teb: 0000000000000000
   Win32Thread: 0000000000000000 RUNNING on processor 0
   Not impersonating
   DeviceMap ffffc18c3f20a760
   Owning Process ffffbd0840200000  Image: System
   Attached Process N/A  Image: N/A
   Wait Start TickCount 12542  Ticks: 0
   Context Switch Count 152847  IdealProcessor: 2
   UserTime 00:00:00.000  KernelTime 00:00:34.671
   Win32 Start Address 0x0000000000000000
   Stack Init fffffa8003e4fc90 Current fffffa8003e4f1a0
   Base fffffa8003e50000 Limit fffffa8003e4a000 Call 0000000000000000
   Priority 12 BasePriority 8 PriorityDecrement 4

// Lister tous les threads d'un processus avec leur état
kd> !process ffffbd0840200000 2
   THREAD ffffbd0843ae7080  Cid 0004.0008  RUNNING  on processor 0
   THREAD ffffbd0843b01080  Cid 0004.000c  WAIT     on ffffbd0843b01110
   THREAD ffffbd0843b15080  Cid 0004.0010  READY    on processor 2
   THREAD ffffbd0843c22080  Cid 0004.0014  WAIT     on ffffbd0843c22110
```

Context switch

Le context switch est l'opération fondamentale du scheduler : sauvegarder l'état complet du thread sortant et restaurer l'état du thread entrant. Sur x86-64, un context switch complet implique :

1. **Sauvegarde des registres du thread sortant** — les registres non-volatiles (RBX, RBP, RSI, RDI, R12-R15) sont poussés sur la pile noyau du thread sortant. Le pointeur de pile résultant est sauvegardé dans `KTHREAD.KernelStack`.
2. **Sauvegarde de l'état FPU/SSE/AVX** — si le thread sortant a utilisé les registres FPU, XMM, ou YMM, leur contenu est sauvegardé via `XSAVE` dans une zone de sauvegarde associée au thread. Windows utilise le « lazy FPU save » : la sauvegarde n'est effectuée que si le thread a réellement modifié l'état FPU depuis le dernier context switch.
3. **Changement de pile** — le pointeur de pile RSP est changé pour pointer vers la pile noyau du thread entrant, en restaurant `KTHREAD.KernelStack`.
4. **Changement d'espace d'adressage (si nécessaire)** — si le thread entrant appartient à un processus différent du thread sortant, le registre CR3 est chargé avec le `DirectoryTableBase` du nouveau processus, ce qui change l'espace d'adressage virtuel. Ce changement invalide les entrées TLB taguées avec l'ancien PCID (Process Context ID), sauf si les PCIDs sont actifs.
5. **Restauration des registres du thread entrant** — les registres non-volatiles sont dépilés depuis la pile noyau du thread entrant.
6. **Mise à jour des structures** — `KPRCB.CurrentThread` est mis à jour, les compteurs de context switch sont incrémentés, et le TEB (Thread Environment Block) en user-mode est mis à jour via la mise à jour du segment GS.

La fonction `KiSwapContext` est le point central du context switch. Elle est écrite en assembleur pour un contrôle total de la manipulation des registres et de la pile :

```
;; KiSwapContext — extrait simplifié (x64)
;; Entrée: rcx = KTHREAD du thread sortant
;;         rdx = KTHREAD du thread entrant
KiSwapContext:
    ;; Sauvegarder les registres non-volatiles
    push rbx
    push rbp
    push rsi
    push rdi
    push r12
    push r13
    push r14
    push r15

    ;; Sauvegarder le pointeur de pile dans le thread sortant
    mov [rcx + KTHREAD_KernelStack], rsp

    ;; Restaurer le pointeur de pile du thread entrant
    mov rsp, [rdx + KTHREAD_KernelStack]

    ;; Changer l'espace d'adressage si nécessaire
    mov rax, [rdx + KTHREAD_ApcState + KAPC_STATE_Process]
    mov rbx, [rcx + KTHREAD_ApcState + KAPC_STATE_Process]
    cmp rax, rbx
    je  .same_address_space
    mov cr3, [rax + KPROCESS_DirectoryTableBase]
.same_address_space:

    ;; Mettre à jour KPRCB.CurrentThread
    mov gs:[KPRCB_CurrentThread], rdx

    ;; Restaurer les registres non-volatiles
    pop r15
    pop r14
    pop r13
    pop r12
    pop rdi
    pop rsi
    pop rbp
    pop rbx

    ret  ;; Retourne dans le contexte du thread entrant
```

Le coût d'un context switch sur du matériel moderne est typiquement de 1-5 µs, selon que le changement d'espace d'adressage est nécessaire, que l'état FPU doit être sauvegardé, et que les mitigations KVAS (Meltdown) sont actives. KVAS (Kernel Virtual Address Shadow) double le coût du changement de CR3 car deux changements sont nécessaires : un pour basculer vers les page tables shadow (user-mode), et un pour revenir aux page tables complètes (kernel-mode). Les PCIDs (Process Context IDs) atténuent le coût en évitant un flush complet du TLB lors du changement de CR3.

Interruptions et préemption

La préemption d'un thread peut se produire dans deux scénarios distincts : la préemption involontaire (un thread de priorité supérieure devient prêt) et le switch volontaire (le thread courant appelle une fonction d'attente).

La préemption involontaire est déclenchée quand un thread de priorité supérieure au thread courant devient prêt. Cela peut se produire via une complétion d'I/O (DPC), un signal d'objet de synchronisation, ou un changement de priorité. La séquence est : le code qui rend le thread prêt appelle `KiDispatcherReadyThread`, qui compare la priorité du nouveau thread avec celle du thread courant sur le processeur cible. Si le nouveau thread est plus prioritaire, il est placé en `Standby` dans `KPRCB.NextThread`, et un DPC de rescheduling est enqueué (ou un IPI est envoyé si le processeur cible est différent). Au retour du DPC/IPI, quand l'IRQL redescend en dessous de DISPATCH_LEVEL, le context switch est effectué.

Le switch volontaire se produit quand un thread appelle `KeWaitForSingleObject`, `KeWaitForMultipleObjects`, `KeDelayExecutionThread` (Sleep), ou toute autre fonction d'attente. Le thread passe de l'état `Running` à l'état `Waiting`, et le scheduler sélectionne immédiatement le prochain thread dans la ready queue. Ce switch volontaire est plus rapide qu'une préemption car il se produit dans un contexte contrôlé — le thread qui cède le CPU a déjà sauvegardé son état de manière cohérente.

L'interruption d'horloge joue un rôle central dans la préemption basée sur le quantum. À chaque tick, le handler d'interruption d'horloge (`KeUpdateRunTime`) décrémente le compteur de quantum du thread courant. Quand ce compteur atteint zéro, le handler n'effectue pas directement le context switch (ce serait trop complexe dans un handler d'interruption) — il enqueue un DPC pour `KiQuantumEnd`. La véritable décision de scheduling est prise dans `KiQuantumEnd`, au niveau IRQL DISPATCH_LEVEL, quand l'interruption d'horloge a déjà été acquittée.

Expiration du quantum

L'expiration du quantum est le mécanisme qui assure le time-sharing entre threads de même priorité. La fonction `KiQuantumEnd` est invoquée via DPC quand le quantum du thread courant atteint zéro. Son comportement :

```
// Pseudo-code de KiQuantumEnd
void KiQuantumEnd(KPRCB *CurrentPrcb) {
    KTHREAD *CurrentThread = CurrentPrcb->CurrentThread;

    // 1. Réinitialiser le quantum
    CurrentThread->Quantum = CurrentThread->Process->QuantumReset;

    // 2. Décrémenter la priorité si un boost est actif (decay)
    if (CurrentThread->PriorityDecrement > 0 &&
        CurrentThread->Priority > CurrentThread->BasePriority) {
        CurrentThread->Priority--;
        CurrentThread->PriorityDecrement--;
    }

    // 3. Chercher un thread de même priorité ou supérieure
    //    dans la ready queue
    ULONG Summary = CurrentPrcb->ReadySummary;
    ULONG CurrentPri = CurrentThread->Priority;

    // Masquer les priorités inférieures
    ULONG HigherOrEqual = Summary & ~((1UL << CurrentPri) - 1);

    if (HigherOrEqual != 0) {
        // Un thread de priorité >= existe dans la ready queue
        // Insérer le thread courant en fin de sa queue
        InsertTailList(
            &CurrentPrcb->DispatcherReadyListHead[CurrentPri],
            &CurrentThread->WaitListEntry);
        CurrentPrcb->ReadySummary |= (1UL << CurrentPri);
        CurrentThread->State = Ready;

        // Sélectionner le thread de plus haute priorité
        ULONG HighPri;
        _BitScanReverse(&HighPri, HigherOrEqual);
        KTHREAD *NextThread = DequeueHead(
            &CurrentPrcb->DispatcherReadyListHead[HighPri]);

        // Context switch
        CurrentPrcb->NextThread = NextThread;
        NextThread->State = Standby;
        KiSwapContext(CurrentThread, NextThread);
    }
    // Sinon : le thread courant continue avec un nouveau quantum
}
```

Un point subtil : la priorité du thread peut décroître (decay) à chaque expiration de quantum si un boost est actif. Par exemple, un thread boosté de priorité 8 (base) à 14 (après un boost I/O de +6) verra sa priorité redescendre progressivement : 14 → 13 → 12 → 11 → 10 → 9 → 8, perdant un niveau à chaque quantum expiré. Cela signifie que le boost a un impact temporel fini : il donne un avantage pendant environ 6 quantums (6 × 31 ms ≈ 190 ms sur un client), puis le thread retrouve son niveau de base.

Pour les threads dans la classe temps réel (priorité 16-31), il n'y a pas de decay. Le quantum expire et est réinitialisé, mais la priorité reste constante. C'est un comportement important pour les applications critiques : un thread à priorité 24 qui consomme tout son quantum ne sera

jamais rétrogradé — il conservera sa priorité 24 indéfiniment, ce qui peut causer une starvation totale des threads de priorité inférieure si le thread temps réel ne cède jamais volontairement le CPU.

> **À retenir :** Le cycle de vie d'un thread traverse 8 états possibles. Le context switch (`KiSwapContext`) sauvegarde/restaure les registres, la pile, et potentiellement l'espace d'adressage (CR3). Le coût typique est de 1-5 µs, augmenté par les mitigations KVAS (Meltdown). L'expiration du quantum déclenche un round-robin entre threads de même priorité et la décroissance (decay) de la priorité boostée. Les threads temps réel (16-31) ne subissent jamais de decay — leur priorité est fixe.

10. Tracing et analyse des performances avec ETW

10.1 Introduction à ETW — Event Tracing for Windows

Event Tracing for Windows (ETW) constitue le mécanisme fondamental d'instrumentation du noyau Windows depuis Windows 2000. Contrairement aux approches de logging traditionnelles, ETW repose sur une architecture **producteur-consommateur asynchrone** à très faible overhead, conçue dès l'origine pour supporter l'instrumentation du scheduler sans perturber les mesures. L'overhead typique d'une session ETW kernel bien configurée se situe entre 1 et 3 % de CPU — suffisamment bas pour capturer des traces en production.

L'architecture ETW s'articule autour de trois composants fondamentaux :

- **Providers** — sources d'événements identifiées par un GUID. On distingue les *manifest-based providers* (schéma XML décrivant chaque événement), les *TraceLogging providers* (auto-descriptifs, format binaire compact) et les legacy *MOF providers*. Le noyau expose le provider spécial `SystemTraceControlGuid` qui regroupe les événements kernel en flags (process, thread, disk I/O, network, etc.).
- **Sessions (controllers)** — buffers circulaires en mémoire kernel qui collectent les événements. Chaque session possède ses propres buffers, sa politique de flush et son fichier de sortie (.etl). La session spéciale `NT Kernel Logger` (remplacée par les *system trace sessions* depuis Windows 8) capture les événements kernel. Windows supporte jusqu'à 64 sessions concurrentes, plus 8 sessions système.
- **Consumers** — processus qui lisent les événements, soit en temps réel (via callback), soit en post-traitement depuis un fichier .etl. Windows Performance Analyzer (WPA), PerfView, TraceEvent et `tracerpt.exe` sont les consumers les plus courants.

La distinction entre providers **kernel-mode** et **user-mode** est capitale pour l'analyse du scheduler. Les événements kernel (context switch, ready thread, DPC/ISR) transitent par le chemin `SystemTraceControlGuid` avec des flags spécifiques comme `EVENT_TRACE_FLAG_CSWITCH` (0x00000010) et `EVENT_TRACE_FLAG_DISPATCHER` (0x00000800). Ces événements sont émis directement depuis le code du dispatcher kernel — la fonction `KiSwapContext` émet le CSwitch,

`KiReadyThread` émet le ReadyThread. L'overhead est minimal car le chemin d'émission est optimisé : vérification du flag dans un registre dédié, écriture lock-free dans le buffer circulaire per-CPU.

Les providers user-mode comme `Microsoft-Windows-Kernel-Scheduler` (GUID `{b3a0c2c8-83bb-4ddf-9f8d-4b22d3c38191}`) offrent une vue plus structurée des décisions du scheduler, avec des événements de haut niveau comme les changements de priorité, les boosts et les décisions d'affinité. En combinant les deux niveaux, on obtient une image complète de l'activité du scheduler.

10.2 Providers scheduler spécialisés

Plusieurs providers ETW sont directement pertinents pour l'analyse du scheduler :

Provider	GUID	Événements clés
NT Kernel (Dispatcher)	SystemTraceControlGuid	CSwitch, ReadyThread, ThreadPriority
Microsoft-Windows-Kernel-Scheduler	{b3a0c2c8-83bb-4ddf-9f8d-4b22d3c38191}	DetailedScheduling, AntiStarvationBoost
Microsoft-Windows-Kernel-Processor-Power	{0f67e49f-fe51-4e9f-b490-6f2948cc6027}	PerfStateChange, ParkingAction, CoreParking
Microsoft-Windows-Kernel-Power	{331c3b3a-2005-44c2-ac5e-77220c37d6b4}	CState transitions, power plan changes
SampledProfile (PMC)	SystemTraceControlGuid + FLAG_PROFILE	SampledProfile (IP sampling)

Les événements les plus critiques pour l'analyse du scheduler sont :

- **CSwitch** (Event ID 36) — émis à chaque context switch. Contient l'identité complète des threads ancien et nouveau, les priorités, la raison d'attente du thread sortant et l'état du thread entrant. C'est l'événement fondamental pour le graphe *CPU Usage (Precise)* de WPA.
- **ReadyThread** (Event ID 50) — émis quand un thread passe à l'état Ready (typiquement après une attente satisfaite). Contient le TID du thread prêt, le TID du thread qui l'a rendu prêt (*readying thread*), et la raison de l'ajustement de priorité. Cet événement est essentiel pour mesurer la **latence de scheduling** — le temps entre ReadyThread et le CSwitch correspondant.
- **SampledProfile** (Event ID 46) — émis par l'interruption PMC (Performance Monitoring Counter) à une fréquence configurable (1 kHz par défaut). Chaque échantillon contient l'instruction pointer (IP) et le thread/processus courant. Permet la construction de profils statistiques et de flame graphs.

10.3 Windows Performance Recorder (WPR)

WPR (`wpr.exe`) est l'outil de capture ETW intégré à Windows depuis Windows 8. Il fournit des profils de capture prédéfinis qui activent les bons providers avec les bons flags et keywords, éliminant la complexité de la configuration manuelle des sessions ETW.

Les profils intégrés pertinents pour l'analyse du scheduler sont :

```
# Capture CPU précise (context switches + ready thread + sampling)
wpr -start CPU

# Capture incluant DPC/ISR (critical pour latence)
wpr -start DPC_ISR

# Capture complète (CPU + DPC + disk + network)
wpr -start GeneralProfile

# Arrêter la capture et sauvegarder
wpr -stop C:\traces\scheduler-analysis.etl "Analyse scheduler"

# Capture avec durée limitée (mode circulaire, buffer 512 MB)
wpr -start CPU -start DPC_ISR -filemode
# ... reproduire le problème ...
wpr -stop C:\traces\output.etl
```

Pour une analyse scheduler approfondie, un profil personnalisé (.wprp) permet de combiner précisément les providers nécessaires :

```
<?xml version="1.0" encoding="utf-8"?>
<WindowsPerformanceRecorder Version="1.0">
  <Profiles>
    <SystemCollector Id="SystemCollector" Name="NT Kernel Logger">
      <BufferSize Value="1024" />
      <Buffers Value="256" />
    </SystemCollector>

    <SystemProvider Id="SchedulerProvider">
      <Keywords>
        <Keyword Value="CSwitch" />
        <Keyword Value="ReadyThread" />
        <Keyword Value="SampledProfile" />
        <Keyword Value="DPC" />
        <Keyword Value="Interrupt" />
        <Keyword Value="ProcessThread" />
        <Keyword Value="Loader" />
      </Keywords>
      <Stacks>
        <Stack Value="CSwitch" />
        <Stack Value="ReadyThread" />
        <Stack Value="SampledProfile" />
      </Stacks>
    </SystemProvider>

    <EventProvider Id="KernelScheduler"
      Name="Microsoft-Windows-Kernel-Scheduler"
      Level="5" />

    <EventProvider Id="ProcessorPower"
      Name="Microsoft-Windows-Kernel-Processor-Power"
      Level="5" />

    <Profile Id="SchedulerAnalysis.Verbose.File"
      Name="SchedulerAnalysis"
      Description="Deep scheduler analysis"
      LoggingMode="File"
      DetailLevel="Verbose">
      <Collectors>
        <SystemCollectorId Value="SystemCollector">
          <SystemProviderId Value="SchedulerProvider" />
        </SystemCollectorId>
      </Collectors>
    </Profile>
  </Profiles>
</WindowsPerformanceRecorder>
```

```
# Utiliser le profil personnalisé
wpr -start scheduler-deep.wprp!SchedulerAnalysis
# ... reproduire le scénario ...
wpr -stop C:\traces\scheduler-deep.etl
```

10.4 Windows Performance Analyzer (WPA)

Windows Performance Analyzer transforme les fichiers .etl en graphiques interactifs. Pour l'analyse du scheduler, les graphes suivants sont indispensables :

CPU Usage (Precise) — construit à partir des événements CSwitch, ce graphe montre exactement quel thread s'exécutait sur quel CPU à chaque instant. Contrairement au sampling (statistique), cette vue est *déterministe* : chaque tranche de temps CPU est attribuée au thread qui l'a consommée. Les colonnes clés sont : `New Process`, `New Thread Id`, `New Thread Stack` (la pile au moment du switch-in), `Ready (µs)` (temps passé dans la ready queue), `CPU Usage (ms)`, `% CPU Usage`, `Switch-In Time`. C'est le graphe de référence pour toute analyse de contention CPU.

CPU Usage (Sampled) — construit à partir des SampledProfile, il fournit des profils statistiques similaires à ceux d'un profiler classique. Chaque échantillon capture l'instruction pointer et la pile d'appels complète. En agrégeant par module et fonction, on identifie les hotspots. Le *flame graph* (accessible via le menu View) offre une visualisation hiérarchique des chemins d'exécution les plus coûteux.

DPC/ISR Duration — affiche la durée de chaque DPC (Deferred Procedure Call) et ISR (Interrupt Service Routine). Les DPC longs (> 100 µs) et les ISR longs (> 50 µs) sont problématiques car ils s'exécutent à `IRQL >= DISPATCH_LEVEL`, ce qui signifie que le scheduler ne peut pas préempter leur exécution. Un DPC de 500 µs crée un « trou » de 500 µs pendant lequel aucun thread utilisateur ne peut s'exécuter sur ce CPU.

Ready Thread — montre les événements ReadyThread, permettant d'analyser la chaîne de causalité : quel thread a réveillé quel autre thread, et combien de temps le thread a passé dans la ready queue avant d'obtenir le CPU. Une ready wait time élevée (> 1 ms) indique une contention CPU significative.

10.5 Analyse des événements clés

L'événement **CSwitch** est le plus riche en information. Chaque champ raconte une partie de l'histoire :

Champ	Description	Usage analytique
`NewThreadId`	TID du thread qui prend le CPU	Identifier le thread consommateur
`OldThreadId`	TID du thread qui quitte le CPU	Identifier le thread préempté ou bloqué
`NewThreadPriority`	Priorité du thread entrant	Vérifier les inversions de priorité
`OldThreadPriority`	Priorité du thread sortant	Détecter préemption par priorité supérieure
`OldThreadWaitReason`	Raison d'attente (Executive, FreePage, UserRequest, etc.)	Comprendre pourquoi le thread a cédé le CPU
`OldThreadState`	Nouvel état du thread sortant (Waiting, Ready, Running)	Distinguer blocage volontaire vs préemption
`PreviousCState`	C-state du CPU avant le switch	Mesurer la latence de réveil du CPU
`NewThreadWaitTime`	Temps passé en attente (100ns units)	Mesurer la scheduling latency

L'interprétation du champ `OldThreadState` est particulièrement révélatrice. Si le thread sortant passe en état **Waiting** (5), il s'est bloqué volontairement (attente I/O, mutex, événement). S'il passe en état **Ready** (1), il a été *préempté* — un thread de priorité supérieure est arrivé. La combinaison `OldThreadState=Ready` + `OldThreadPriority < NewThreadPriority` confirme une préemption classique par priorité.

L'événement **ReadyThread** complète le tableau en fournissant le *readying thread* — le thread qui a causé le passage à l'état Ready. Par exemple, si le thread A libère un mutex et que le thread B (qui attendait ce mutex) reçoit un ReadyThread avec `AdjustReason=Unwait`, on peut reconstruire la chaîne de dépendance. Les raisons d'ajustement incluent :

- `Unwait (0)` — le thread a été réveillé suite à la satisfaction de son attente
- `Boost (1)` — un priority boost a été appliqué (typiquement après une I/O completion)
- `DeferredReady (2)` — le thread a été marqué comme ready sur un autre CPU (inter-processor ready)

Le **SampledProfile** fonctionne sur un principe statistique. L'interruption PMC (configurée par défaut à 1 kHz, soit un échantillon toutes les millisecondes) capture l'instruction pointer courant et la pile d'appels. Sur une capture de 10 secondes, on obtient environ 10 000 échantillons par CPU. Si une fonction apparaît dans 2 000 échantillons sur un CPU, elle consomme environ 20 % du temps CPU de ce core. La fréquence d'échantillonnage peut être augmentée (jusqu'à 8 kHz) pour plus de précision, au prix d'un overhead accru.

Les événements **DPC/ISR** méritent une attention particulière dans le contexte du scheduling. Un DPC s'exécute à `IRQL DISPATCH_LEVEL` (2), ce qui bloque le scheduler sur le CPU concerné. Un pilote réseau mal écrit qui traite des paquets dans un DPC au lieu d'utiliser NDIS polling peut générer des DPC de plusieurs millisecondes, causant des latences de scheduling imprévisibles. WPA affiche les DPC par durée, module source et CPU cible, permettant d'identifier rapidement les coupables.

10.6 Méthodologie d'analyse

Une analyse scheduler structurée suit cette séquence :

Étape 1 : Identifier la contention CPU. Ouvrir le graphe CPU Usage (Precise), grouper par CPU. Vérifier si des CPUs sont saturés (> 95 %) pendant que d'autres sont idle. Un déséquilibre indique un problème d'affinité ou de placement NUMA.

Étape 2 : Mesurer la latence de scheduling. Dans CPU Usage (Precise), ajouter la colonne `Ready (µs)`. Trier par ready time décroissant. Des valeurs supérieures à 1 ms pour des threads interactifs indiquent une contention. Des valeurs supérieures à 16 ms (un quantum complet) signalent un problème sévère — le thread attend plus d'un tour complet de scheduling.

Étape 3 : Détecter les migrations excessives. Dans CPU Usage (Precise), grouper par `New Process / New Thread Id / CPU`. Si un thread apparaît sur de nombreux CPUs différents, il migre fréquemment. Chaque migration invalide potentiellement les caches L1/L2 du core source. Sur des architectures NUMA, les migrations cross-node sont particulièrement coûteuses.

Étape 4 : Analyser la profondeur des ready queues. Le nombre de threads prêts (ready) à un instant donné indique le niveau de surcharge. Si la moyenne est supérieure au nombre de CPUs, le système est en oversubscription. Utiliser les événements ReadyThread pour compter les threads simultanément prêts.

Étape 5 : Rechercher les inversions de priorité. Filtrer les CSwitch où `OldThreadPriority > NewThreadPriority` et `OldThreadState = Ready`. Cela ne devrait pas se produire dans un scheduling normal — un thread de priorité supérieure ne devrait pas être préempté par un thread de priorité inférieure. Si cela arrive, c'est généralement dû à un hard affinity qui force le thread haute priorité sur un CPU spécifique déjà occupé.

Étape 6 : Vérifier l'impact DPC/ISR. Ouvrir le graphe DPC/ISR Duration. Tout DPC supérieur à 100 µs est suspect. Identifier le module source (pilote) et le CPU affecté. Un pilote réseau générant des DPC de 1 ms sur le CPU 0 crée un « angle mort » de scheduling récurrent sur ce core.

> **Workflow ETW minimal pour l'analyse scheduler :**
>
> 1) `wpr -start CPU -start DPC_ISR` — démarrer la capture.
> 2) Reproduire le scénario problématique pendant 15-30 secondes.
> 3) `wpr -stop trace.etl` — arrêter et sauvegarder.
> 4) Ouvrir dans WPA : CPU Usage (Precise) pour la vue déterministe, CPU Usage (Sampled) pour les hotspots, DPC/ISR pour les interférences kernel.
> 5) Mesurer : ready wait time, context switch rate, DPC duration, migration frequency.

11. Debug avancé avec WinDbg

11.1 Commandes essentielles

WinDbg (et sa version moderne WinDbg Preview) est l'outil de référence pour l'inspection directe des structures internes du scheduler. En mode kernel debug (live ou crash dump), il donne accès aux structures KTHREAD, KPRCB et aux ready queues — les données brutes sur lesquelles repose toute décision de scheduling.

La commande `!thread` affiche l'état complet d'un thread du point de vue du scheduler :

```
kd> !thread ffffb90c`3a4e1080
THREAD ffffb90c3a4e1080  Cid 0e4c.1234  Teb: 0000006c`7e8fa000
    Win32Thread: ffffb90c3b2f01a0 WAIT: (UserRequest) UserMode Non-Alertable
        ffffb90c3a55e8c0  SynchronizationEvent
    Not impersonating
    DeviceMap     ffffca8214c37aa0
    Owning Process        ffffb90c3a2b4080  Image: app.exe
    Attached Process      N/A
    Wait Start TickCount  1582345
    Context Switch Count  48291     IdealProcessor: 4
    UserTime              00:00:02.156
    KernelTime            00:00:00.468
    Win32 Start Address   0x00007ff6`12345678
    Stack Init fffffd00`c4b8fc90 Current fffffd00`c4b8f430
    Base fffffd00`c4b90000 Limit fffffd00`c4b8a000
    Priority 10 BasePriority 8 PriorityDecrement 2
    Quantum: 12 (remaining)
    Affinity: processor mask = 0x000000ff
    IdealProcessor: 4  LastProcessor: 6  NextProcessor: -1
```

Chaque ligne révèle un aspect du scheduling : l'état (WAIT avec raison UserRequest), la priorité courante (10) vs base (8) montrant un boost actif de +2, le quantum restant (12 unités), le processeur idéal (4) vs le dernier processeur utilisé (6) indiquant une migration récente, et le masque d'affinité (0xFF = CPUs 0-7).

Pour obtenir la vue globale des processus et threads :

```
# Lister tous les processus (vue sommaire)
kd> !process 0 0

# Détail complet d'un processus avec tous ses threads
kd> !process ffffb90c3a2b4080 7

# Afficher la structure KTHREAD brute
kd> dt nt!_KTHREAD ffffb90c3a4e1080
   +0x000 Header           : _DISPATCHER_HEADER
   +0x018 SListFaultAddress : (null)
   +0x020 QuantumTarget    : 0x5e2d90
   +0x028 InitialStack     : 0xfffffd00`c4b8fc90
   +0x098 Priority         : 10 ''
   +0x17c IdealProcessor   : 4
   +0x184 ThreadFlags      : 0x00040000
   +0x1c0 WaitTime         : 0x18249d
   +0x1e8 BasePriority     : 8 ''
   +0x1e9 PriorityDecrement : 2 ''
```

La commande `!prcb` expose l'état du scheduler sur un CPU spécifique :

```
# État du KPRCB pour le CPU 0
kd> !prcb 0
PRCB for Processor 0 at fffff805`12340180:
Current IRQL -- 0
Threads--  Current ffffb90c3a4e1080 Next ffffb90c3b221080
           Idle    fffff805`12356700
Number  0  SetMember         1
Interrupt Count -- 0015a7c2
Times -- Dpc    00000068 Interrupt 0000001a
         Kernel 000123ff  User      0005678a
         DpcQueueDepth 0  Max 3

# Afficher les ready queues
kd> !ready
Processor 0: Ready Threads at priority 13
    THREAD ffffb90c3b445080  Cid 0234.0a1c  Teb: 0000005a`12340000
Processor 0: Ready Threads at priority 8
    THREAD ffffb90c3b442080  Cid 0e4c.1a20  Teb: 0000005a`12341000
    THREAD ffffb90c3b443080  Cid 0e4c.1a24  Teb: 0000005a`12342000
Processor 2: Ready Threads at priority 10
    THREAD ffffb90c3b448080  Cid 07a0.0c14  Teb: 0000005a`12343000

# Threads en cours d'exécution sur chaque CPU
kd> !running
Processor 0: Thread ffffb90c3a4e1080 (app.exe)
Processor 1: Thread fffff805`12356700 (Idle)
Processor 2: Thread ffffb90c3b221080 (svchost.exe)
Processor 3: Thread ffffb90c3b225080 (sqlservr.exe)
...
```

11.2 Analyse d'un deadlock scheduler

Un deadlock de scheduling se manifeste quand des threads forment une chaîne circulaire d'attentes mutuelles. Le diagnostic suit une procédure systématique :

Étape 1 : identifier les threads bloqués. Dans un crash dump ou une session live, commencer par repérer les threads en état WAIT depuis longtemps :

```
# Lister les threads du processus suspect avec détail
kd> !process ffffb90c3a2b4080 7

# Pour chaque thread en WAIT, examiner l'objet d'attente
kd> !thread ffffb90c3a4e1080
    WAIT: (Executive) KernelMode Non-Alertable
        ffffb90c3b667040  Mutant - owning thread ffffb90c3b221080

# Le thread 3a4e1080 attend un Mutant (mutex) possédé par 3b221080
# Examiner le thread propriétaire
kd> !thread ffffb90c3b221080
    WAIT: (Executive) KernelMode Non-Alertable
        ffffb90c3b668040  Mutant - owning thread ffffb90c3a4e1080
```

Étape 2 : construire le graphe d'attente. Thread A attend le mutex M1 possédé par Thread B. Thread B attend le mutex M2 possédé par Thread A. Cycle détecté = deadlock confirmé.

Étape 3 : examiner les piles d'appels pour comprendre le contexte d'acquisition :

```
kd> .thread ffffb90c3a4e1080
kd> kn
 # Child-SP          RetAddr           Call Site
00 fffffd00`c4b8f2a0 fffff805`1a2b3456 nt!KiSwapContext+0x76
01 fffffd00`c4b8f3e0 fffff805`1a2b1234 nt!KiSwapThread+0x501
02 fffffd00`c4b8f490 fffff805`1a2b0abc nt!KiCommitThreadWait+0x132
03 fffffd00`c4b8f530 fffff805`1a300100 nt!KeWaitForSingleObject+0x233
04 fffffd00`c4b8f5d0 fffff805`1a300050 nt!ExpWaitForResource+0x6c
05 fffffd00`c4b8f650 fffff807`2a100abc driver.sys!AcquireLockB+0x4c
06 fffffd00`c4b8f6a0 fffff807`2a100200 driver.sys!ProcessRequest+0x120
```

La pile montre que le thread attend dans `ExpWaitForResource` appelé depuis `driver.sys!AcquireLockB`. En combinant les deux piles, on identifie la violation d'ordre d'acquisition des locks dans le driver — classique erreur de programmation concurrente.

11.3 Analyse d'un CPU à 100 %

Quand un CPU est bloqué à 100 %, la première question est : qui consomme ? La méthodologie WinDbg est directe :

```
# Voir ce qui tourne sur chaque CPU
kd> !running -t

# Si un CPU montre un temps DPC élevé, c'est un DPC storm
kd> !dpcs
CPU  DPC-Queue-Depth  Max-Queue-Depth
  0      0              3
  1      0              1
  2      47             128    <-- DPC accumulation anormale

# Examiner le thread fautif
kd> .thread ffffb90c3b225080
kd> !thread ffffb90c3b225080
    Running on processor 3
    Priority 15 BasePriority 15
    KernelTime 00:45:12.000  <-- 45 minutes de kernel time
kd> kn
 # Call stack shows tight loop in kernel driver
```

Si le CPU est consommé en user mode, la pile montrera du code applicatif. Si c'est en kernel mode, on verra des fonctions nt! ou un driver tiers. La distinction est cruciale : un CPU à 100 % en user mode est généralement un problème applicatif (boucle infinie, algorithme $O(n^2)$ sur un grand n), tandis qu'un CPU à 100 % en kernel mode est souvent un bug driver (spinlock contention, DPC storm, watchdog timeout imminent).

11.4 Inspection des ready queues

L'inspection directe des ready queues permet de diagnostiquer la starvation et la surcharge. La structure KPRCB contient 32 listes chaînées `DispatcherReadyListHead[0..31]`, une par niveau de priorité :

```
# Examiner la structure brute du PRCB
kd> dt nt!_KPRCB fffff805`12340180 DispatcherReadyListHead
   +0x7c80 DispatcherReadyListHead : [32] _LIST_ENTRY

# Vérifier si la liste pour priorité 8 est vide
kd> dt _LIST_ENTRY fffff805`12340180+0x7c80+8*0x10
   +0x000 Flink : 0xfffff805`12347d00  <-- pointe vers lui-même = vide
   +0x008 Blink : 0xfffff805`12347d00

# Liste non-vide pour priorité 13
kd> dt _LIST_ENTRY fffff805`12340180+0x7c80+13*0x10
   +0x000 Flink : 0xffffb90c`3b445098  <-- pointe vers un KTHREAD
   +0x008 Blink : 0xffffb90c`3b446098

# Parcourir la liste des threads prêts à priorité 13
kd> !list -x "!thread @$extret" ffffb90c`3b445098
```

Un nombre élevé de threads dans les ready queues basse priorité (0-7) combiné à des queues haute priorité (8-15) constamment occupées indique un risque de starvation. Le mécanisme d'anti-starvation de Windows booste les threads affamés à priorité 15 pendant un quantum, mais ce mécanisme est un palliatif — si la surcharge est permanente, les threads basse priorité n'obtiendront qu'une fraction dérisoire du CPU.

> **Commandes WinDbg essentielles pour le scheduler :**
>
> `!thread <addr>` — état complet du thread (priorité, wait reason, quantum, affinité).
> `!ready` — contenu des ready queues par CPU et priorité.
> `!running` — threads en cours d'exécution sur chaque CPU.
> `!prcb <cpu>` — état du KPRCB (current/next/idle thread, compteurs).
> `dt nt!_KTHREAD <addr>` — dump brut de la structure KTHREAD.
> `dt nt!_KPRCB <addr> DispatcherReadyListHead` — accès direct aux ready queues.

12. Pathologies courantes du scheduling

12.1 Thread starvation

La starvation (famine) se produit quand un thread ne reçoit jamais ou presque jamais de temps CPU, bien qu'il soit dans l'état Ready. Ce phénomène survient quand des threads de priorité supérieure consomment en permanence 100 % des CPUs disponibles, ne laissant aucune opportunité aux threads de priorité inférieure.

Le scénario classique : un serveur applicatif avec 8 CPUs exécute 12 threads de priorité 13 (classe Above Normal) en boucle de calcul. Les threads des services d'arrière-plan à priorité 8 (Normal) ne s'exécutent jamais — ils restent indéfiniment dans les ready queues. Les symptômes visibles incluent : services Windows qui ne répondent plus, mises à jour qui ne s'installent pas, event log qui arrête d'écrire.

La détection via ETW est précise. Dans WPA, filtrer CPU Usage (Precise) par `Ready (µs)` et trier par valeur décroissante. Des ready times de plusieurs secondes voire dizaines de secondes confirment la starvation. Le graphe ReadyThread permet de voir le timestamp exact auquel le thread est devenu Ready et de mesurer le délai avant exécution.

Windows intègre un mécanisme d'**anti-starvation** : le *Balance Set Manager* (thread kernel exécuté périodiquement) scanne les ready queues et identifie les threads qui n'ont pas reçu de CPU depuis environ 4 secondes (environ 300 ticks du clock interrupt à ~64 Hz). Ces threads reçoivent un boost temporaire à priorité 15 pour un seul quantum. Ce mécanisme a deux limitations importantes : premièrement, 4 secondes est un délai considérable pour un service interactif. Deuxièmement, un seul quantum (environ 15 ms sur un client) est souvent insuffisant pour que le thread accomplisse du travail utile — il retombe immédiatement à sa priorité base et retourne en starvation.

Sur Windows Server 2025, les quantum plus longs (180 ms) rendent le boost d'anti-starvation plus efficace — le thread affamé reçoit un quantum complet de 180 ms au lieu de 15 ms, lui permettant d'avancer significativement dans son travail. C'est un des avantages parfois sous-estimés des longs quantum serveur.

12.2 Priority inversion

L'inversion de priorité est un problème classique des systèmes temps réel qui affecte également Windows dans certains scénarios. Le cas canonique implique trois threads :

- **Thread H** (haute priorité, par exemple 15) — doit acquérir un mutex M
- **Thread M** (priorité moyenne, par exemple 10) — thread de calcul, n'utilise pas M
- **Thread L** (basse priorité, par exemple 4) — possède le mutex M, en cours d'exécution dans la section critique

La séquence pathologique : Thread L acquiert le mutex M et commence sa section critique. Thread H devient prêt et tente d'acquérir M — il se bloque car M est possédé par L. Thread M devient prêt. Comme M a priorité 10 > L à priorité 4, M préempte L. Résultat : Thread H (priorité 15) est effectivement bloqué par Thread M (priorité 10), qui n'a pourtant aucun rapport avec le mutex. L'inversion est complète : la priorité effective de H est réduite à celle de L.

La mitigation standard est le **priority inheritance** : quand H se bloque sur le mutex possédé par L, le scheduler devrait temporairement élever la priorité de L au niveau de H, permettant à L de terminer sa section critique rapidement. Windows implémente cette technique, mais **uniquement pour les mutex kernel** (objets KMUTANT). Les *critical sections* user-mode, les *SRW locks* et les *slim reader/writer locks* ne bénéficient d'aucune forme de priority inheritance. C'est une limitation significative : la majorité de la synchronisation applicative utilise des primitives user-mode.

En pratique, l'inversion de priorité sur Windows se manifeste le plus souvent dans les scénarios suivants : un thread temps réel (MMCSS, priorité 26) attend une critical section possédée par un thread normal ; un driver kernel utilise un spinlock (pas affecté par l'inversion, car les spinlocks élèvent l'IRQL) mais le code autour utilise un fast mutex sans inheritance ; des threads avec des priorités très différentes partagent un pool de connexions ou un cache.

La détection via ETW : chercher les événements CSwitch où un thread haute priorité entre en état WAIT pour un objet Mutant/Event, puis vérifier que le thread propriétaire est de priorité inférieure et est lui-même préempté par des threads de priorité intermédiaire.

12.3 Oversubscription de threads

L'oversubscription se produit quand le nombre de threads exécutables (état Ready + Running) dépasse significativement le nombre de CPUs logiques. Dans un système à 8 CPUs avec 200 threads prêts, le scheduler doit effectuer des context switches constants, chaque thread n'obtenant qu'une fraction de quantum avant d'être préempté.

L'overhead du context switching lui-même n'est pas négligeable : chaque switch coûte entre 2 et 10 µs (sauvegarde/restauration des registres, flush TLB partiel, invalidation branch predictor). Mais le coût indirect est bien supérieur : chaque switch provoque potentiellement des cache misses L1/L2 quand le nouveau thread accède à son working set, qui a été évincé pendant qu'il ne s'exécutait pas. Sur un processeur moderne avec 48 KB de L1D et 1.25 MB de L2, un thread qui n'a pas tourné depuis 200 ms aura très probablement perdu tout son état cache.

La règle empirique pour le dimensionnement des thread pools est simple :

- **Calcul CPU-bound** : nombre de threads = nombre de CPUs logiques (ou physiques si SMT est désactivé pour la sécurité)
- **I/O-bound** : nombre de threads = nombre de CPUs × (1 + temps_attente / temps_calcul). Pour des I/O réseau avec 90 % d'attente, cela donne environ 10 × CPUs.
- **Mixte** : utiliser les I/O completion ports (IOCP) de Windows, qui maintiennent automatiquement un nombre de threads actifs proche du nombre de CPUs.

La détection via ETW : un taux de context switch supérieur à 15 000/s par CPU indique généralement une oversubscription. Dans WPA, le graphe CPU Usage (Precise) avec `Count (Context Switches)` agrégé par seconde donne cette métrique directement.

12.4 Pénalités NUMA

Sur les systèmes multi-socket (typiques des serveurs), l'architecture NUMA (Non-Uniform Memory Access) introduit une asymétrie fondamentale : l'accès mémoire local (même socket) coûte environ 80-100 ns, tandis que l'accès distant (autre socket via QPI/UPI) coûte 130-200 ns — un facteur 1.5 à 2×. Ce ratio, appelé *NUMA ratio*, varie selon l'architecture (Intel Sapphire Rapids : ~1.4, AMD EPYC : ~1.8 avec les CCD multiples).

Le scheduler Windows est NUMA-aware : il tente de placer les threads sur le même nœud NUMA que la mémoire qu'ils utilisent le plus. Le champ `IdealNode` dans KTHREAD indique le nœud NUMA préféré. Cependant, en cas de déséquilibre de charge, le scheduler peut migrer un thread vers un nœud distant. Le thread continue alors d'accéder à sa mémoire sur le nœud d'origine, subissant la pénalité de latence sur chaque accès.

La détection combine ETW (context switch events avec CPU ID → mapper sur la topologie NUMA) et les PMC (compteurs d'accès mémoire locale vs distante). Windows Performance Monitor expose les compteurs `NUMA Node Memory\Local & Remote Bytes/sec` qui quantifient directement le trafic cross-node.

12.5 Cache thrashing par migration

Chaque fois qu'un thread migre d'un core à un autre, il perd potentiellement son état dans le cache L1 et L2 du core source (ces caches sont privés par core sur toutes les architectures x86 modernes). Le L3 (LLC) est partagé au niveau du socket/CCD, donc une migration intra-socket préserve le L3. Une migration cross-socket perd les trois niveaux de cache.

L'impact est mesurable avec les PMC (Performance Monitoring Counters). Les compteurs pertinents sont `L1D.REPLACEMENT` (évictions L1 data), `L2_LINES_IN.ALL` (remplissages L2) et `LLC-LOAD-MISSES` (misses LLC). En corrélant les pics de cache misses avec les événements CSwitch montrant une migration, on peut quantifier le coût exact des migrations.

Le champ `LastProcessor` dans KTHREAD indique le dernier CPU utilisé, et `IdealProcessor` le CPU préféré. Quand ces deux valeurs divergent fréquemment, le thread souffre de migrations excessives. La contre-mesure est d'utiliser `SetThreadIdealProcessor` (soft affinity) plutôt que `SetThreadAffinityMask` (hard affinity) pour guider le scheduler sans le contraindre.

13. Optimisations avancées

13.1 CPU affinity tuning

L'affinité CPU contraint un thread à s'exécuter uniquement sur un sous-ensemble de processeurs logiques. Windows expose deux niveaux d'API :

```
// Hard affinity — le thread ne pourra s'exécuter QUE sur ces CPUs
DWORD_PTR mask = 0x0F; // CPUs 0-3
SetThreadAffinityMask(hThread, mask);
SetProcessAffinityMask(hProcess, mask);

// Soft affinity — le scheduler préfère ce CPU mais peut migrer si nécessaire
SetThreadIdealProcessor(hThread, 4); // Préférence pour CPU 4

// Windows 10+ : API étendue pour les systèmes à plus de 64 CPUs
GROUP_AFFINITY affinity;
affinity.Group = 0;
affinity.Mask = 0x0F;
SetThreadGroupAffinity(hThread, &affinity, NULL);
```

L'affinité dure (*hard affinity*) est un outil puissant mais dangereux. Elle est appropriée dans les cas suivants : applications temps réel qui ne peuvent pas tolérer les latences de migration cache, benchmarking contrôlé où on veut isoler l'effet du scheduling, et isolation de workloads (dédier les CPUs 0-3 au réseau, 4-7 à l'application). Les risques sont réels : si le CPU cible est occupé par un DPC ou un ISR, le thread ne peut pas migrer vers un CPU libre. Si le nombre de threads affinés à un CPU dépasse 1, ils se partagent ce seul CPU même si d'autres sont idle.

En pratique, pour les applications latency-sensitive, la combinaison optimale est : hard affinity sur un set de CPUs (pas un seul), avec `SetThreadIdealProcessor` pour la préférence au sein du set. Par exemple, affinité sur les CPUs 4-7 (même CCX/CCD sur AMD, même module sur Intel) avec processeur idéal = 4 :

```
DWORD_PTR mask = (1ULL << 4) | (1ULL << 5) | (1ULL << 6) | (1ULL << 7);
SetThreadAffinityMask(hThread, mask);
SetThreadIdealProcessor(hThread, 4);
```

13.2 Allocation NUMA-aware

Sur les systèmes NUMA, aligner l'allocation mémoire avec le placement des threads est fondamental pour les performances. Windows fournit des API explicites :

```
// Allouer de la mémoire sur un nœud NUMA spécifique
LPVOID buffer = VirtualAllocExNuma(
    GetCurrentProcess(),
    NULL,                    // adresse suggérée
    64 * 1024 * 1024,       // 64 MB
    MEM_RESERVE | MEM_COMMIT,
    PAGE_READWRITE,
    0                        // nœud NUMA 0
);

// Déterminer le nœud NUMA courant
USHORT nodeNumber;
GetNumaProcessorNodeEx(&processorNumber, &nodeNumber);

// Créer un thread sur un nœud NUMA spécifique
// (indirectement, via affinité + allocation)
HANDLE hThread = CreateThread(NULL, 0, WorkerFunc, buffer, CREATE_SUSPENDED, NULL);
GROUP_AFFINITY ga = { .Mask = numaNodeMask[0], .Group = 0 };
SetThreadGroupAffinity(hThread, &ga, NULL);
ResumeThread(hThread);
```

La stratégie optimale pour une application NUMA-aware est la suivante : au démarrage, détecter la topologie NUMA avec `GetLogicalProcessorInformationEx`. Pour chaque nœud NUMA, créer un pool de threads affiné à ce nœud et allouer les données de ce pool avec `VirtualAllocExNuma` sur le même nœud. Les structures de données partagées entre nœuds doivent être répliquées (chaque nœud a sa copie locale) ou partitionnées (chaque nœud accède à sa tranche).

13.3 Design de thread pool

Le dimensionnement du thread pool est critique pour éviter l'oversubscription. Windows fournit une API de thread pool intégrée (depuis Vista) qui gère automatiquement le nombre de threads :

```
// Créer un pool personnalisé avec contrôle du nombre de threads
PTP_POOL pool = CreateThreadpool(NULL);
SetThreadpoolThreadMinimum(pool, 4);
SetThreadpoolThreadMaximum(pool, 8);  // = nombre de CPUs

// Environnement de callback lié au pool
TP_CALLBACK_ENVIRON ce;
InitializeThreadpoolEnvironment(&ce);
SetThreadpoolCallbackPool(&ce, pool);

// Soumettre du travail
PTP_WORK work = CreateThreadpoolWork(WorkCallback, context, &ce);
SubmitThreadpoolWork(work);
```

Les I/O Completion Ports (IOCP) représentent le mécanisme le plus sophistiqué de Windows pour le dimensionnement dynamique. Un IOCP maintient un nombre de threads actifs (non bloqués) égal au `NumberOfConcurrentThreads` spécifié (généralement = nombre de CPUs). Quand un thread se bloque (I/O, lock), l'IOCP libère automatiquement un thread supplémentaire pour maintenir la concurrence cible. Ce mécanisme évite à la fois l'oversubscription et la sous-utilisation.

```
// Créer un IOCP avec concurrence = nombre de CPUs
HANDLE hIocp = CreateIoCompletionPort(
    INVALID_HANDLE_VALUE,
    NULL,
    0,
    0  // 0 = nombre de CPUs logiques
);

// Associer un handle de fichier/socket à l'IOCP
CreateIoCompletionPort(hSocket, hIocp, (ULONG_PTR)context, 0);

// Worker threads : boucle de déqueue
DWORD bytes;
ULONG_PTR key;
LPOVERLAPPED ov;
while (GetQueuedCompletionStatus(hIocp, &bytes, &key, &ov, INFINITE)) {
    // Traiter l'I/O complétée
    ProcessCompletion(key, ov, bytes);
}
```

13.4 Priorité et QoS tuning

Windows 11 et Server 2025 introduisent des mécanismes de Quality of Service (QoS) au niveau du scheduler qui vont au-delà de la simple priorité :

EcoQoS (Efficiency Quality of Service) indique au scheduler qu'un processus peut s'exécuter sur les E-cores et à fréquence réduite. C'est l'outil privilégié pour les tâches d'arrière-plan :

```
PROCESS_POWER_THROTTLING_STATE throttling;
throttling.Version = PROCESS_POWER_THROTTLING_CURRENT_VERSION;
throttling.ControlMask = PROCESS_POWER_THROTTLING_EXECUTION_SPEED;
throttling.StateMask = PROCESS_POWER_THROTTLING_EXECUTION_SPEED;

SetProcessInformation(
    hProcess,
    ProcessPowerThrottling,
    &throttling,
    sizeof(throttling)
);
```

MMCSS (Multimedia Class Scheduler Service) fournit des garanties de scheduling pour les applications multimédia. Un thread enregistré auprès de MMCSS reçoit une priorité dans la plage temps réel (16-31) pendant les périodes de traitement :

```
DWORD taskIndex = 0;
HANDLE hTask = AvSetMmThreadCharacteristicsW(L"Pro Audio", &taskIndex);
// Le thread est maintenant à priorité ~26, protégé de la préemption
// par les threads normaux

// À la fin du traitement
AvRevertMmThreadCharacteristics(hTask);
```

Les classes MMCSS prédéfinies (`Audio`, `Pro Audio`, `Capture`, `Games`, `Playback`) ont des paramètres de scheduling différents définis dans le registre sous `HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Multimedia\SystemProfile\Tasks`.

13.5 Réduction des context switches

Chaque context switch inutile est une perte de performance. Les techniques de réduction incluent :

Batching du travail — au lieu de réveiller un thread pour chaque unité de travail, accumuler N unités et traiter le lot. Réduit le nombre de transitions Wait→Ready→Running.

Algorithmes lock-free — les structures de données lock-free (via `InterlockedCompareExchange`, `InterlockedPushEntrySList`) évitent les blocages mutex qui causent des context switches. La SList (Singly-Linked List) interlocked de Windows est particulièrement efficace : push et pop atomiques sans lock, utilisée massivement dans le noyau lui-même (lookaside lists).

Spinning adaptatif — pour les locks à courte durée, un spin (boucle active) de quelques microsecondes est plus efficace qu'un context switch. Les critical sections Windows implémentent cela via `InitializeCriticalSectionAndSpinCount` : le thread spin pendant N itérations avant de se bloquer. Le spin count optimal dépend du workload : 4000 est la valeur recommandée par Microsoft pour le heap manager, mais des valeurs de 100-400 suffisent pour des locks à très courte durée.

```
CRITICAL_SECTION cs;
InitializeCriticalSectionAndSpinCount(&cs, 4000);
// Sur un système monoprocesseur, le spin count est ignoré
// (pas de sens de spinner si aucun autre CPU ne peut libérer le lock)
```

User-Mode Scheduling (UMS) — introduit dans Windows 7, UMS permettait aux applications de gérer elles-mêmes le scheduling de threads user-mode sans transitionner vers le kernel. SQL Server l'utilisait. UMS est désormais **déprécié** depuis Windows 11 — Microsoft recommande les thread pools avec IOCP comme alternative.

13.6 Optimisations spécifiques par workload

HPC / calcul intensif — la stratégie est de minimiser les interférences. Pinner chaque thread de calcul sur un core dédié via hard affinity. Désactiver le SMT (un thread par core physique) pour éviter le partage des unités d'exécution. Utiliser les large pages (2 MB au lieu de 4 KB) pour réduire les TLB misses — `VirtualAlloc` avec `MEM_LARGE_PAGES` requiert le privilège `SeLockMemoryPrivilege`. Configurer le prefetch matériel via les MSR si le pattern d'accès mémoire est prévisible. Sur un cluster HPC, chaque nœud de calcul devrait avoir le power plan High Performance pour désactiver le core parking et les C-states profonds.

Pentest / bruteforce — les outils de bruteforce comme hashcat optimisent déjà massivement le scheduling GPU. Pour la partie CPU (John the Ripper, Hydra), la parallélisation doit saturer tous les cores sans oversubscription. Sur un système NUMA, distribuer les threads uniformément entre les nœuds. Hashcat utilise principalement le GPU ; côté CPU, le thread de contrôle doit avoir une priorité Normal pour ne pas interférer avec les threads GPU qui émettent des commandes via le scheduler WDDM.

Parsing massif (logs, PCAP, forensics) — ces workloads sont typiquement I/O-bound puis CPU-bound par phases. La stratégie optimale est un pipeline : threads de lecture asynchrone (avec IOCP, count = nombre de disques), threads de parsing (count = nombre de CPUs), connected par des queues lock-free. Éviter de créer un thread par fichier — c'est la recette de l'oversubscription.

SQL / bases de données — SQL Server implémente son propre scheduler coopératif (SQLOS) au-dessus du scheduler Windows. SQLOS crée un scheduler user-mode par CPU logique, chaque scheduler ayant sa propre file de tâches. Les threads SQL cèdent volontairement le contrôle aux points de yield explicites dans le code SQL Server. Le paramètre `MAXDOP` (Max Degree of Parallelism) contrôle le parallélisme des requêtes. Sur un système NUMA, configurer MAXDOP = nombre de cores par nœud NUMA pour éviter les requêtes parallèles cross-node. Le `cost threshold for parallelism` (défaut: 5) détermine le seuil de coût à partir duquel une requête est parallélisée.

Virtualisation — en environnement Hyper-V, le placement des vCPUs (Virtual Processors) sur les CPUs physiques suit les mêmes principes. Le VP pinning (affinité VM→CPUs physiques) est critique pour les VMs latency-sensitive. L'overcommit CPU (plus de vCPUs que de pCPUs) cause exactement les mêmes problèmes que l'oversubscription de threads, amplifiés par le double scheduling (hyperviseur + OS invité). La règle en production : ratio vCPU:pCPU maximal de 4:1 pour les workloads généraux, 1:1 pour les workloads critiques. Sur les systèmes NUMA, configurer chaque VM pour qu'elle tienne dans un seul nœud NUMA (toute sa mémoire et tous ses vCPUs sur le même nœud).

> **Règles d'optimisation par workload :**
>
> **CPU-bound** : threads = CPUs, hard affinity, large pages, disable SMT si sécurité critique.
> **I/O-bound** : IOCP + thread pool dynamique, async I/O, pas d'oversubscription.
> **NUMA** : allouer mémoire sur le nœud local, affinité thread-nœud, répliquer les données partagées.
> **Latency-sensitive** : MMCSS ou priorité élevée, spin count élevé, disable core parking, C-states shallow only.

14. Optimisations système

14.1 Power plans et core parking

Le power plan Windows influence directement le comportement du scheduler via deux mécanismes principaux : la gestion de fréquence CPU (P-states) et le core parking. Le plan **High Performance** désactive le core parking et maintient la fréquence CPU au maximum, éliminant la latence de réveil des cores parkés et de montée en fréquence. Le plan **Balanced** (défaut) active le core parking et la gestion dynamique de fréquence, économisant l'énergie au prix d'une latence accrue quand la charge augmente.

Le core parking est contrôlé par le paramètre `MinimumUnparkedProcessorPercentage`. À 100 %, aucun core n'est parké (équivalent à High Performance pour cette dimension). La valeur par défaut en Balanced est typiquement 50 % — sur un système 8 cores, 4 cores peuvent être parkés en idle.

```
# Voir le plan actif
powercfg /getactivescheme

# Lister les sous-groupes relatifs au processeur
powercfg /query SCHEME_CURRENT SUB_PROCESSOR

# Modifier le pourcentage minimum de cores non-parkés (AC = secteur)
# GUID SUB_PROCESSOR = 54533251-82be-4824-96c1-47b60b740d00
# GUID MinUnparked = 0cc5b647-c1df-4637-891a-dec35c318583
powercfg /setacvalueindex SCHEME_CURRENT 54533251-82be-4824-96c1-47b60b740d00 0cc5b647-
c1df-4637-891a-dec35c318583 100

# Appliquer les modifications
powercfg /setactive SCHEME_CURRENT

# Forcer le plan High Performance (GUID standard)
powercfg /setactive 8c5e7fda-e8bf-4a96-9a85-a6e23a8c635c
```

Sur Windows Server 2025, un nouveau plan **Optimized** est disponible. Il combine les avantages de Balanced (économie d'énergie en idle) avec un algorithme de montée en fréquence plus agressif : la latence de transition idle→full performance est réduite de 30-50 ms (Balanced) à 5-10 ms. C'est le plan recommandé pour les serveurs qui alternent entre périodes d'activité et d'idle.

L'impact sur le scheduling est direct : un core parké est retiré des candidats pour le placement de threads. Quand un thread devient Ready et que le core idéal est parké, le scheduler doit soit unpark le core (latence de 50-200 µs selon le C-state), soit placer le thread sur un core moins optimal. Pour les workloads latency-sensitive, le core parking doit être désactivé.

14.2 BIOS tuning

Les paramètres BIOS/UEFI affectent profondément le comportement du scheduler via les C-states, le SMT et le Turbo Boost :

Paramètre	Valeur perf max	Valeur balanced	Impact scheduling
C-States	C1 only (C1E disabled)	Tous activés (C0-C10)	C6+ : réveil 100+ µs, latence scheduling
Package C-State	C0 (No package idle)	Auto	Package C6 : réveil 1+ ms
SMT / Hyper-Threading	Activé (sauf sécurité)	Activé	2× threads logiques, partage ressources core
Turbo Boost	Activé	Activé	Fréquence dynamique, dépend du # cores actifs
Hardware P-States (HWP)	Native mode	Native mode	CPU gère la fréquence, réaction plus rapide
NUMA Interleaving	Désactivé	Désactivé	Si activé, détruit la localité NUMA

Les C-states profonds (C6 et au-delà) sont les plus impactants pour le scheduling. En C6, le core a vidé son cache L1/L2 et coupé son alimentation. Le réveil prend 100-200 µs — une éternité pour un thread qui attend depuis 10 µs dans la ready queue. Sur les serveurs de base de données et les systèmes temps réel, limiter les C-states à C1 (halt, réveil < 1 µs) est une pratique courante.

Le SMT (Hyper-Threading chez Intel) double le nombre de threads logiques mais partage les ressources d'exécution (ALU, FPU, cache L1/L2) entre les deux threads du même core. Pour les workloads CPU-bound, le gain est de 15-30 %. Mais en contexte sécurité, le SMT est un vecteur d'attaques side-channel (Spectre, MDS, L1TF) — voir section 16. La désactivation du SMT est recommandée pour les hyperviseurs hébergeant des VMs non-fiables.

14.3 Interrupt steering et RSS

Receive Side Scaling (RSS) distribue le traitement des interruptions réseau sur plusieurs CPUs au lieu de surcharger le CPU 0 (comportement par défaut historique). Chaque queue RSS est associée à un CPU via une table d'indirection, et les paquets sont distribués entre les queues par hash (typiquement sur le 4-tuple IP source/dest, port source/dest).

```
# Voir la configuration RSS actuelle
Get-NetAdapterRss

# Configurer RSS : 8 queues sur les CPUs 0-7
Set-NetAdapterRss -Name "Ethernet0" -NumberOfReceiveQueues 8 -BaseProcessorNumber 0

# Configurer l'affinité d'interruption manuellement
# (via le registre, sous la clé du driver réseau)
# HKLM\SYSTEM\CurrentControlSet\Control\Class\{4d36e972-...}\0001
# MessageSignaledInterruptProperties\0000005\MSISupported = 1
# MessageSignaledInterruptProperties\0000005\InterruptPolicy =
#   IrqPolicySpecifiedProcessors
```

L'alignement RSS avec l'affinité des threads applicatifs est une optimisation souvent négligée. Si l'application réseau exécute ses workers sur les CPUs 4-7, configurer RSS pour distribuer les interruptions réseau sur ces mêmes CPUs évite les inter-processor interrupts (IPI) pour le transfert de données entre le stack réseau et l'application.

14.4 Network stack tuning

Windows Server 2025 introduit le **NDIS polling mode**, une avancée majeure pour le traitement réseau haute performance. Au lieu du modèle traditionnel interrupt-driven (chaque paquet génère une interruption → DPC → indication NDIS), le polling mode permet au stack réseau de interroger la NIC à intervalles réguliers, éliminant l'overhead des interruptions et DPC pour les workloads à haut débit.

```
# Activer le mode polling NDIS (Server 2025)
Set-NetAdapterAdvancedProperty -Name "Ethernet0" `
    -RegistryKeyword "*NdisPoll" -RegistryValue 1

# Configurer la modération d'interruptions
Set-NetAdapterAdvancedProperty -Name "Ethernet0" `
    -RegistryKeyword "*InterruptModeration" -RegistryValue 1

# Ajuster le taux de modération (microsecondes entre interrupts)
Set-NetAdapterAdvancedProperty -Name "Ethernet0" `
    -RegistryKeyword "ITR" -RegistryValue 100
```

L'impact sur le scheduler est significatif : le polling mode réduit drastiquement le nombre de DPC réseau, libérant du temps CPU pour les threads applicatifs. Sur un serveur traitant 1 million de paquets/seconde, le passage d'interrupt-driven à polling peut libérer 15-20 % de CPU en réduisant le temps passé en DPC.

15. Benchmarks et cas pratiques

15.1 Méthodologie benchmark

Un benchmark scheduler fiable requiert un protocole rigoureux. Les variables parasites sont nombreuses : Turbo Boost qui varie avec la température, C-states qui ajoutent de la latence au premier appel, background processes qui consomment des cycles, et le propre overhead du benchmark qui perturbe ce qu'il mesure.

Le protocole recommandé est le suivant :

- **Environnement contrôlé** — power plan High Performance, core parking désactivé, C-states limités à C1, Turbo Boost stable (soit désactivé, soit après warm-up thermique)
- **Warm-up** — exécuter le workload pendant 30 secondes avant de commencer les mesures, pour stabiliser le branch predictor, les caches et la fréquence CPU
- **Répétitions** — minimum 10 itérations, reporter médiane et percentiles (P95, P99), pas la moyenne (sensible aux outliers)
- **Instrumentation** — capturer une trace ETW pendant le benchmark pour corréler les résultats avec le comportement du scheduler

- **Outils** — Geekbench (single/multi-thread), Cinebench (sensible au scheduling multi-core), custom micro-benchmarks basés sur `QueryPerformanceCounter` pour les mesures de latence

15.2 Comparaison Windows 11 vs Server 2025

La différence fondamentale entre Windows 11 et Server 2025 réside dans les quantum : 15.6 ms (client) vs ~180 ms (serveur). L'impact est mesurable sur deux dimensions opposées :

Métrique	Windows 11 (client)	Server 2025	Delta
Context switches/sec (8 threads CPU-bound, 4 CPUs)	~3 200	~280	-91 %
Throughput calcul brut (Cinebench multi)	~12 400 pts	~12 950 pts	+4.4 %
Latence interactive (click-to-render)	~8 ms (P50)	~45 ms (P50)	+460 %
Latence scheduling (ready→running, P95)	~0.3 ms	~1.2 ms	+300 %
Cache miss rate L2 (migration-induced)	~2.8 %	~1.1 %	-61 %

Les résultats confirment le compromis architecturel : les quantum longs du serveur améliorent le throughput brut de ~4-5 % grâce à moins de context switches et moins de cache pollution. En contrepartie, la latence de scheduling augmente considérablement — un thread prêt peut attendre jusqu'à 180 ms avant que le thread courant termine son quantum.

15.3 Impact de l'affinité CPU

Le pinning de threads montre des gains variables selon le workload. Sur un benchmark de hashing SHA-256 (8 threads, système 8 cores/16 threads) :

Configuration	Throughput (GH/s)	Context switches/s	L2 miss rate
Sans affinité (défaut scheduler)	4.82	1 240	3.1 %
Affinité soft (IdealProcessor)	4.91	890	2.4 %
Affinité hard (1 thread/core physique)	5.12	45	1.2 %
Affinité hard + SMT disabled	4.98	42	0.9 %

L'affinité hard donne le meilleur throughput (+6.2 %) grâce à l'élimination quasi-totale des migrations. La désactivation du SMT réduit le throughput malgré un meilleur cache hit rate — le pipeline bénéficie de l'alternance entre les deux threads SMT quand l'un est en attente mémoire.

15.4 Impact NUMA

Sur un serveur bi-socket (2 × 32 cores, 2 nœuds NUMA), l'allocation mémoire NUMA-aware montre un impact spectaculaire :

Scénario	Latence mémoire (ns)	Bande passante (GB/s)	Throughput app
Threads et mémoire sur même nœud	82	45.2	100 % (baseline)
Threads et mémoire sur nœuds différents	148	28.1	62 %
Threads migrant entre nœuds (sans affinité)	115 (moyenne)	35.4	78 %

Le placement cross-NUMA réduit le throughput de 38 %. Même avec le scheduler NUMA-aware de Windows, l'absence d'affinité explicite cause des migrations inter-nœuds qui dégradent les performances de 22 %. Sur les workloads memory-intensive (bases de données, analytics), la configuration NUMA est souvent le facteur de performance numéro un.

15.5 Impact du scheduler hybride P/E-core

Sur un processeur Intel de 13e/14e génération (8 P-cores + 16 E-cores), le placement correct du scheduler a un impact considérable :

Scénario	Score single-thread	Score multi-thread
Thread Director actif (Win11 23H2+)	2 050	24 800
Thread Director désactivé	1 680 (-18 %)	23 100 (-7 %)
Tous threads forcés sur P-cores	2 050	16 400 (-34 %)
Tous threads forcés sur E-cores	1 220 (-40 %)	19 600 (-21 %)

Le Thread Director maximise les performances single-thread en plaçant les threads IPC-critiques sur les P-cores, tout en exploitant les E-cores pour le parallélisme massif. Forcer tous les threads sur les P-cores (seulement 8) pénalise massivement le multi-thread. Le Thread Director atteint un placement quasi-optimal dans la majorité des cas, grâce aux métriques matérielles (instruction mix, IPC, comportement mémoire) transmises en temps réel au scheduler.

16. Sécurité et scheduler

16.1 Side-channels SMT

Le Simultaneous Multi-Threading partage les ressources microarchitecturales entre deux threads logiques sur le même core physique. Cette intimité architecturale a ouvert un vaste champ d'attaques side-channel depuis 2018 :

Spectre (variantes 1 et 2) — exploite l'exécution spéculative pour lire de la mémoire normalement inaccessible. En contexte SMT, un thread attaquant sur le sibling logique peut entraîner le branch predictor partagé pour rediriger l'exécution spéculative de la victime. La variante 2 (Branch Target Injection) est particulièrement dangereuse en SMT car le Branch Target Buffer (BTB) est partagé.

MDS (Microarchitectural Data Sampling) — famille de vulnérabilités (RIDL, Fallout, ZombieLoad) exploitant les buffers internes du CPU (store buffers, fill buffers, load ports). Un thread sur un sibling SMT peut observer les données transitant dans ces buffers, même si elles appartiennent à l'autre thread. L'impact est catastrophique : lecture de données arbitraires depuis un autre thread du même core, traversant les frontières de processus et de VM.

L1 Terminal Fault (L1TF / Foreshadow) — exploite le cache L1 data partagé entre siblings SMT. Un thread peut accéder au contenu L1 de l'autre thread via une faille dans la gestion des entrées de table de pages. Particulièrement critique en environnement virtualisé : un VM malveillante peut lire la mémoire de l'hyperviseur ou d'autres VMs si elles partagent un core physique.

La mitigation principale côté scheduler est le **core scheduler** de Hyper-V (activé par défaut depuis Windows Server 2019). Au lieu de scheduler les vCPUs individuellement sur les threads logiques, le core scheduler alloue des *cores physiques entiers* aux VMs. Deux vCPUs d'une même VM peuvent partager un core (ils sont dans le même domaine de confiance), mais deux vCPUs de VMs différentes ne partagent jamais un core. Cela élimine les attaques cross-VM via SMT, au prix d'une réduction de densité (moins de VMs par serveur).

16.2 Isolation CPU

L'isolation CPU va au-delà de la simple affinité. Pour les workloads sensibles à la sécurité, plusieurs niveaux d'isolation sont disponibles :

- **Process affinity** — restreindre les processus sensibles à un sous-ensemble de CPUs via `SetProcessAffinityMask` ou la propriété d'affinité dans le Task Manager. Empêche le co-scheduling avec des processus non fiables.
- **CPU sets (Windows 10+)** — `SetProcessDefaultCpuSets` permet de réserver des CPUs à des processus spécifiques. Les CPUs dans un set ne sont pas utilisés par le scheduler pour d'autres processus, fournissant une isolation plus forte que la simple affinité.
- **Hyper-V CPU groups** — dans Hyper-V, les CPU groups permettent de partitionner les CPUs physiques entre VMs avec des garanties d'isolation matérielle.
- **VBS (Virtualization-Based Security)** — utilise l'hyperviseur pour créer un environnement isolé (Secure Kernel / VTL 1) où les secrets (credentials, clés) sont protégés. Le scheduler de VTL 1 est distinct de celui de VTL 0 — les threads Secure Kernel sont schedulés indépendamment.

16.3 Scheduling et attaques timing

Le scheduler lui-même peut être un canal d'information. Un attaquant peut inférer l'activité d'un processus victime en mesurant ses propres temps de scheduling :

Covert timing channels — deux processus coopérants (par exemple, un malware et son C2 sur le même hôte) peuvent communiquer via le scheduler. Le processus émetteur module sa consommation CPU (charge/idle), et le récepteur mesure ses propres temps de scheduling. Si le CPU est partagé, la latence de scheduling du récepteur est corrélée à l'activité de l'émetteur. Le débit est faible (~100 bits/s) mais le canal est invisible aux outils de monitoring réseau.

Cache-based side channels — en mesurant le temps d'accès à des lignes de cache spécifiques (techniques Prime+Probe, Flush+Reload), un thread peut déterminer quelles lignes de cache la victime a accédées, révélant potentiellement des clés cryptographiques. Le scheduling affecte ces attaques : un thread doit être co-schedulé sur le même core (pour L1/L2) ou le même socket (pour LLC) que la victime.

Scheduler-based information leakage — le simple fait d'observer *quand* un thread est schedulé (via `QueryPerformanceCounter` à chaque wake-up) révèle le pattern d'activité du système. Sur un système chargé, les variations de scheduling latency corrèlent avec l'activité des autres processus.

16.4 Mitigations

L'écosystème de mitigations combine hardware, firmware et scheduler :

Mitigation	Cible	Mécanisme	Impact perf
Core Scheduler (Hyper-V)	Cross-VM SMT attacks	Allocation par core physique	5-10 % (densité réduite)
Retpoline	Spectre v2 (BTI)	Remplace les indirect branches par des séquences retpoline	1-5 %
SSBD (Speculative Store Bypass Disable)	Spectre v4	Désactive le bypass spéculatif des stores	2-8 %
L1D Flush	L1TF	Flush du cache L1D au VM-exit	3-15 % (selon VM-exit rate)
MDS mitigations (VERW)	MDS/TAA	Vidage des buffers microarchitecturaux	3-10 %
HVCI (Hypervisor Code Integrity)	Code injection kernel	Vérification d'intégrité des pages kernel	5-15 %
SMT désactivé	Toutes attaques SMT	Un thread par core physique	20-30 % (throughput multi-thread)

La décision de désactiver le SMT est un compromis coût-bénéfice. Pour un hyperviseur public (cloud multi-tenant), le core scheduler de Hyper-V est le minimum requis, et la désactivation complète du SMT est recommandée par Microsoft pour les workloads les plus sensibles (voir documentation Microsoft). Pour un serveur dédié mono-tenant, le core scheduler seul est généralement suffisant.

Du côté applicatif, les développeurs de code cryptographique doivent utiliser des implémentations *constant-time* (pas de branches conditionnelles dépendant des secrets, pas d'accès mémoire indexés par des secrets) pour neutraliser les side-channels. Les bibliothèques comme BoringSSL, libsodium et les implémentations AES-NI matérielles sont conçues avec cette contrainte.

Sécurité et scheduler — principes fondamentaux :

Le SMT est un vecteur d'attaques side-channel (Spectre, MDS, L1TF). En environnement multi-tenant, activer le **core scheduler Hyper-V** au minimum, envisager la désactivation SMT pour les workloads critiques. L'isolation CPU (CPU sets, VBS, CPU groups) fournit des garanties de non-interférence. Les attaques timing via le scheduler existent mais sont à faible débit — la menace principale reste le partage de ressources microarchitecturales.

17. Évolutions futures

17.1 Scheduler ML-driven

L'Intel Thread Director représente la première incursion du machine learning dans le scheduling production. Le firmware du CPU exécute un modèle de classification entraîné sur des métriques microarchitecturales (instruction mix, ratio IPC, taux de cache miss, utilisation des unités vectorielles) pour classer chaque thread dans une catégorie de workload. Cette classification est transmise au scheduler OS via le registre HRESET/ITD, permettant un placement P-core/E-core optimal.

L'évolution logique est un scheduler OS lui-même piloté par ML. Les candidats pour l'inférence ML incluent : prédiction de la durée de burst CPU (pour optimiser le quantum), prédiction du pattern I/O (pour anticiper les wake-ups), et prédiction de la localité mémoire (pour optimiser le placement NUMA). Google a publié des recherches sur ghOSt, un framework de scheduling Linux piloté par un agent user-mode, permettant d'expérimenter des politiques ML sans modifier le noyau. Microsoft Research explore des approches similaires avec leur travail sur les schedulers adaptifs.

Les défis sont considérables : le scheduling est un hot path (appelé des milliers de fois par seconde), donc le coût d'inférence doit être négligeable (< 1 µs). Les modèles doivent être robustes aux adversarial workloads — un processus malveillant ne doit pas pouvoir tromper le scheduler ML pour obtenir un avantage injuste. Et la déterminabilité des systèmes temps réel exclut les comportements probabilistes d'un scheduler ML.

17.2 Hardware scheduling

L'approche ARM big.LITTLE (précurseur du modèle hybride Intel) évolue vers des architectures à plus de deux classes de performance. Les processeurs ARM v9 introduisent des configurations tri-cluster (prime + performance + efficiency). Qualcomm Snapdragon X Elite utilise cette approche pour Windows on ARM, avec des implications directes sur la complexité du scheduler Windows.

Intel poursuit l'intégration de fonctionnalités de scheduling dans le hardware. Au-delà du Thread Director, les futures architectures pourraient inclure des mécanismes de migration hardware-assistée (le CPU déplace un thread entre P-core et E-core sans intervention OS, simplement en reconfigurant le pipeline) et des schedulers de tâches intégrés au chipset pour les workloads de type data flow.

17.3 CPU heterogeneity avancée

L'ère des chiplets (AMD EPYC, Intel Ponte Vecchio) et du 3D stacking (AMD 3D V-Cache) introduit une hétérogénéité intra-socket. Tous les cores d'un même socket n'ont pas les mêmes caractéristiques : les cores proches du V-Cache ont une latence L3 inférieure, les cores sur des chiplets différents ont des latences inter-core différentes. Le scheduler doit intégrer cette topologie dans ses décisions de placement.

Les futures architectures pourraient même combiner des cores avec des ISA extensions différentes (par exemple, certains cores avec AMX pour les opérations matricielles, d'autres sans). Le scheduler devrait alors matcher les besoins d'instruction set du thread avec les capacités du core — un niveau de complexité que les schedulers actuels n'adressent pas.

17.4 Tendances cloud et hyperscale

Le scheduling au niveau hyperscale opère sur des contraintes différentes. Les micro-VMs (Firecracker, développé par AWS pour Lambda) ont des temps de démarrage de 125 ms — le scheduling de ces micro-VMs devient aussi critique que le scheduling de threads dans un OS. Le cold start des fonctions serverless est directement lié à la vitesse à laquelle l'orchestrateur peut allouer et scheduler une micro-VM.

Les architectures d'isolation par processus (gVisor, Kata Containers) ajoutent une couche de scheduling supplémentaire. Un appel système dans un conteneur gVisor est intercepté par le sentry (un processus Go), qui effectue un context switch user-mode avant d'appeler le kernel si nécessaire. Cette double indirection a un coût, et l'optimisation du scheduling à ce niveau est un axe de recherche actif.

18. Conclusion

Le scheduler de Windows est un système d'une complexité remarquable, forgé par trois décennies d'évolution depuis Windows NT 3.1. Sa conception — scheduler préemptif à priorités fixes avec quantum variable — reste fondamentalement celle de Dave Cutler en 1993, mais les optimisations accumulées (NUMA-awareness, support hybride P/E-core, Thread Director, core scheduler Hyper-V) l'ont transformé en un système adaptatif capable de gérer des topologies hardware impensables à l'époque.

Plusieurs enseignements ressortent de cette analyse approfondie :

Premièrement, la compréhension des structures internes (KTHREAD, KPRCB, ready queues) est indispensable pour diagnostiquer les problèmes de performance. Les outils de surface (Task Manager, Resource Monitor) sont insuffisants pour les problèmes complexes. ETW et WinDbg sont les instruments de précision qui permettent de voir exactement ce que fait le scheduler et pourquoi.

Deuxièmement, la dichotomie Windows 11 (client) vs Server 2025 (serveur) dans le traitement des quantum reflète un compromis fondamental qui n'a pas de solution universelle. Les quantum courts favorisent l'interactivité, les longs favorisent le throughput. Le choix doit être guidé par le workload, et dans certains cas, un tuning intermédiaire via le registre est justifié.

Troisièmement, l'architecture hybride P-core/E-core représente un changement de paradigme pour le scheduling. Le Thread Director d'Intel est une première réponse, mais l'hétérogénéité croissante des futures architectures (chiplets, 3D stacking, ISA variable) va exiger des schedulers de plus en plus intelligents. Le scheduler de Windows est, pour l'instant, le plus avancé dans ce domaine grâce à son intégration étroite avec Thread Director.

Quatrièmement, la sécurité et le scheduling sont désormais indissociables. Les attaques side-channel via SMT ont forcé des changements architecturaux majeurs (core scheduler Hyper-V) et des compromis de performance significatifs (désactivation SMT, L1D flush, VERW). Toute discussion sur l'optimisation du scheduling doit intégrer la dimension sécurité, surtout en environnement multi-tenant.

Pour le praticien — administrateur système, développeur kernel, pentester ou architecte infrastructure — la maîtrise du scheduler Windows est un avantage décisif. Elle permet d'optimiser les performances de manière scientifique (mesurer avec ETW, diagnostiquer avec WinDbg, corriger avec les bons paramètres) plutôt que par tâtonnement. Elle permet aussi de comprendre les implications sécuritaires du placement de threads, un aspect de plus en plus critique dans un monde où les attaques microarchitecturales sont devenues réalité.

Le scheduler Windows continuera d'évoluer. Les tendances ML-driven, l'hétérogénéité hardware croissante et les exigences du cloud hyperscale vont imposer des architectures de scheduling radicalement différentes dans la prochaine décennie. Les fondamentaux — priorités, quantum, affinité, NUMA — resteront pertinents, mais leur orchestration sera de plus en plus déléguée à des algorithmes adaptatifs et à du hardware spécialisé. Comprendre les principes sous-jacents aujourd'hui, c'est se préparer à maîtriser les systèmes de demain.

19. Annexes

19.1 Structures internes — dumps WinDbg

Les structures KTHREAD et KPRCB sont les deux piliers du scheduler. Voici les champs les plus pertinents avec leur offset et leur rôle :

```
kd> dt nt!_KTHREAD -r1
   +0x000 Header            : _DISPATCHER_HEADER
   +0x018 SListFaultAddress  : Ptr64 Void
   +0x020 QuantumTarget      : Uint8B          // Cycle count cible pour fin de quantum
   +0x028 InitialStack       : Ptr64 Void      // Base du kernel stack
   +0x030 StackLimit         : Ptr64 Void
   +0x038 StackBase          : Ptr64 Void
   +0x040 ThreadLock         : Uint8B           // Spinlock per-thread
   +0x048 CycleTime          : Uint8B           // Cycles CPU consommés (total)
   +0x058 CurrentRunTime     : Uint4B           // Temps d'exécution courant
   +0x05c ExpectedRunTime    : Uint4B           // Prédiction du runtime
   +0x060 KernelStack        : Ptr64 Void
   +0x074 Running            : UChar            // 1 si en cours d'exécution
   +0x075 Alerted            : [2] UChar
   +0x098 Priority           : Char             // Priorité courante (0-31)
   +0x099 BasePriority       : Char             // Priorité base
   +0x09a PriorityDecrement  : UChar            // Décrément après boost
   +0x09b Preempted          : UChar            // 1 si préempté
   +0x09c AdjustReason       : UChar            // Raison du dernier ajustement
   +0x09d AdjustIncrement    : Char             // Incrément du dernier boost
   +0x168 WaitTime           : Uint4B           // Timestamp du début d'attente
   +0x17c IdealProcessor     : Uint4B           // CPU préféré
   +0x180 LastProcessor      : Uint4B           // Dernier CPU utilisé
   +0x184 ThreadFlags        : Uint4B
   +0x188 Spare19            : [3] UChar
   +0x1d0 Affinity           : _GROUP_AFFINITY  // Masque d'affinité
   +0x1e8 Process            : Ptr64 _KPROCESS  // Processus parent
   +0x220 State              : UChar            // 0=Init,1=Ready,2=Running,
                                                // 3=Standby,4=Terminated,5=Waiting
   +0x221 WaitReason         : UChar            // Raison d'attente (enum)
   +0x222 WaitMode           : UChar            // KernelMode(0) / UserMode(1)
   +0x254 SchedulerAssist    : Uint4B           // Données Thread Director (Win11+)
   +0x280 EnergyValues       : Ptr64            // QoS / EcoQoS state
```

```
kd> dt nt!_KPRCB -r1
   +0x000 MxCsr              : Uint4B
   +0x008 Number             : UChar            // Numéro du CPU
   +0x010 CurrentThread      : Ptr64 _KTHREAD   // Thread en cours
   +0x018 NextThread         : Ptr64 _KTHREAD   // Prochain thread (si standby)
   +0x020 IdleThread         : Ptr64 _KTHREAD   // Thread idle
   +0x028 NestingLevel       : UChar
   +0x02c PrcbPad02          : [3] UChar
   +0x030 Number             : Uint4B
   +0x038 SetMember          : Uint8B           // Bit correspondant au CPU
   +0x040 PrcbLock           : Uint8B
   +0x4c80 ReadySummary      : Uint4B           // Bitmask des priorités non-vides
   +0x4c84 QueueIndex        : Uint4B
   +0x4c88 DeferredReadyListHead : _SINGLE_LIST_ENTRY
   +0x7c80 DispatcherReadyListHead : [32] _LIST_ENTRY  // 32 ready queues
   +0x7e80 InterruptCount    : Uint4B
   +0x7e84 KernelTime        : Uint4B
   +0x7e88 UserTime          : Uint4B
   +0x7e8c DpcTime           : Uint4B
   +0x7e90 InterruptTime     : Uint4B
   +0x8b00 ParentNode        : Ptr64 _KNODE     // Nœud NUMA parent
```

19.2 Scripts ETW PowerShell

Scripts PowerShell pour l'automatisation de la capture et l'analyse des traces scheduler :

```
# ============================================================
# Script 1 : Capture scheduler ETW avec analyse automatique
# ============================================================

param(
    [int]$DurationSeconds = 30,
    [string]$OutputPath = "C:\traces"
)

# Créer le dossier de sortie
New-Item -ItemType Directory -Force -Path $OutputPath | Out-Null

$timestamp = Get-Date -Format "yyyyMMdd_HHmmss"
$etlFile = Join-Path $OutputPath "scheduler_$timestamp.etl"

Write-Host "[*] Démarrage capture scheduler ETW ($DurationSeconds secondes)..."

# Démarrer WPR avec les profils CPU et DPC/ISR
wpr -start CPU -start DPC_ISR -filemode

# Attendre la durée spécifiée
Start-Sleep -Seconds $DurationSeconds

# Arrêter la capture
wpr -stop $etlFile "Scheduler analysis capture"

Write-Host "[+] Trace sauvegardée : $etlFile"
Write-Host "[*] Taille : $([math]::Round((Get-Item $etlFile).Length / 1MB, 2)) MB"

# Analyse rapide avec xperf (si installé)
if (Get-Command xperf -ErrorAction SilentlyContinue) {
    Write-Host "[*] Analyse des context switches..."
    xperf -i $etlFile -o (Join-Path $OutputPath "cswitch_$timestamp.csv") `
        -a cswitch -summary

    Write-Host "[*] Analyse DPC/ISR..."
    xperf -i $etlFile -o (Join-Path $OutputPath "dpcisr_$timestamp.csv") `
        -a dpcisr -summary
}

Write-Host "[+] Analyse terminée. Ouvrir $etlFile dans WPA pour l'analyse détaillée."
```

```
# ============================================================
# Script 2 : Monitoring temps réel des context switches
# ============================================================

$counterPaths = @(
    "\System\Context Switches/sec",
    "\System\Processor Queue Length",
    "\Processor(_Total)\% Processor Time",
    "\Processor(_Total)\% DPC Time",
    "\Processor(_Total)\% Interrupt Time"
)

Write-Host "[*] Monitoring scheduler (Ctrl+C pour arrêter)"
Write-Host "=" * 80

while ($true) {
    $counters = Get-Counter -Counter $counterPaths -SampleInterval 1
    $values = $counters.CounterSamples

    $cswitch = [math]::Round($values[0].CookedValue)
    $queueLen = [math]::Round($values[1].CookedValue, 1)
    $cpuPct = [math]::Round($values[2].CookedValue, 1)
    $dpcPct = [math]::Round($values[3].CookedValue, 2)
    $isrPct = [math]::Round($values[4].CookedValue, 2)

    $status = ""
    if ($cswitch -gt 50000) { $status = " [!] OVERSUBSCRIPTION" }
    if ($queueLen -gt (Get-CimInstance Win32_Processor |
        Measure-Object -Property NumberOfLogicalProcessors -Sum).Sum * 2) {
        $status += " [!] QUEUE SATURÉE"
    }
    if ($dpcPct -gt 5) { $status += " [!] DPC ÉLEVÉ" }

    $ts = Get-Date -Format "HH:mm:ss"
    Write-Host ("[$ts] CS/s: {0,8:N0} | Queue: {1,4} | CPU: {2,5}% | DPC: {3,5}% | ISR:
{4,5}%{5}" -f `
        $cswitch, $queueLen, $cpuPct, $dpcPct, $isrPct, $status)
}
```

```
# ============================================================
# Script 3 : Capture ciblée d'un processus spécifique
# ============================================================

param(
    [Parameter(Mandatory=$true)]
    [string]$ProcessName,
    [int]$DurationSeconds = 15
)

$process = Get-Process -Name $ProcessName -ErrorAction Stop | Select-Object -First 1
$pid = $process.Id

Write-Host "[*] Cible : $ProcessName (PID $pid)"
Write-Host "[*] Threads : $($process.Threads.Count)"
Write-Host "[*] Working Set : $([math]::Round($process.WorkingSet64 / 1MB, 2)) MB"

# Session ETW avec logman (alternative à WPR pour plus de contrôle)
$sessionName = "SchedulerTrace_$pid"
$etlFile = "C:\traces\sched_${ProcessName}_${pid}.etl"

# Créer la session avec les flags kernel scheduler
logman create trace $sessionName -o $etlFile `
    -p "Microsoft-Windows-Kernel-Scheduler" 0xFFFFFFFF 0xFF `
    -p "{ce1dbfb4-137e-4da6-87b0-3f59aa102cbc}" 0xFFFFFFFF 0xFF `
    -bs 1024 -nb 256 512 -mode Circular -max 512 -ets

# Ajouter les événements kernel (context switch + ready thread)
logman update trace $sessionName `
    -p "Windows Kernel Trace" "(cswitch,dispatcher)" -ets

Write-Host "[*] Capture en cours ($DurationSeconds secondes)..."
Start-Sleep -Seconds $DurationSeconds

logman stop $sessionName -ets
Write-Host "[+] Trace sauvegardée : $etlFile"
```

19.3 Registry tuning

Les clés de registre suivantes permettent d'ajuster le comportement du scheduler. **Attention** : modifier ces valeurs peut déstabiliser le système. Toujours tester en environnement non-production d'abord.

Clé de registre	Valeur	Type	Desc
`HKLM\SYSTEM\CurrentControlSet\Control\PriorityControl\Win32PrioritySeparation`	0x26 (défaut client) 0x18 (défaut serveur)	REG_DWORD	Cont boos fixed quar ratio
`HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\kernel\ThreadDpcEnable`	0 ou 1	REG_DWORD	Activ (exéc au lie
`HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management\LargePageMinimum`	Taille en octets	REG_DWORD	Seuil page
`HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Multimedia\SystemProfile\SystemResponsiveness`	0-100 (défaut: 20)	REG_DWORD	Pour non- MMC
`HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Multimedia\SystemProfile\Tasks\<TaskName>\Priority`	1-8	REG_DWORD	Prior plus
`HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Multimedia\SystemProfile\Tasks\<TaskName>\Scheduling Category`	High, Medium, Low	REG_SZ	Caté déte
`HKLM\SYSTEM\CurrentControlSet\Control\Power\PowerSettings\54533251-... \0cc5b647-...\DefaultValue`	0-100	REG_DWORD	Mini — po

Le registre `Win32PrioritySeparation` mérite une explication détaillée car il encode plusieurs paramètres dans un seul DWORD :

```
# Décoder Win32PrioritySeparation
# Valeur 0x26 = 0b00_10_0110
# Bits [1:0] = 10 = foreground boost ratio 3:1 (3× le quantum pour le foreground)
# Bits [3:2] = 01 = short quantum (6 unités ≈ 2 clock ticks ≈ 31.25 ms)
# Bits [5:4] = 10 = variable length quantum (dépend du boost)

# Pour un serveur haute performance : 0x28
# Bits [1:0] = 00 = pas de foreground boost
# Bits [3:2] = 10 = long quantum (12 unités ≈ 4 clock ticks ≈ 62.5 ms)
# Bits [5:4] = 10 = variable length

# Modifier via PowerShell (ATTENTION: nécessite reboot)
Set-ItemProperty -Path "HKLM:\SYSTEM\CurrentControlSet\Control\PriorityControl" `
    -Name "Win32PrioritySeparation" -Value 0x28 -Type DWord

# Vérifier la valeur actuelle
Get-ItemProperty -Path "HKLM:\SYSTEM\CurrentControlSet\Control\PriorityControl" `
    -Name "Win32PrioritySeparation" | Select-Object -ExpandProperty
Win32PrioritySeparation
```

19.4 Glossaire

Terme	Définition
Affinity (affinité)	Masque de bits définissant les CPUs sur lesquels un thread peut s'exécuter. Hard affinity = contrainte stricte, soft affinity (IdealProcessor) = préférence.
Context Switch	Opération de sauvegarde de l'état CPU d'un thread et restauration de l'état d'un autre. Coûte 2-10 µs directs, plus les coûts indirects de cache.
C-State	État d'économie d'énergie du CPU. C0 = actif, C1 = halt (réveil < 1 µs), C6 = power gate (réveil ~100 µs), C10 = package idle (réveil ~1 ms).
DPC (Deferred Procedure Call)	Callback kernel exécuté à IRQL DISPATCH_LEVEL, utilisé pour le traitement différé des interruptions. Bloque le scheduler sur le CPU concerné.
E-core (Efficiency core)	Core à haute efficacité énergétique dans les architectures hybrides Intel (Gracemont, Crestmont). IPC inférieur aux P-cores mais consommation réduite.
ETW (Event Tracing for Windows)	Infrastructure d'instrumentation kernel/user-mode à faible overhead. Base de WPR/WPA et de la plupart des outils de diagnostic Windows.
ISR (Interrupt Service Routine)	Handler d'interruption matérielle, exécuté à IRQL élevé. Doit être le plus court possible — le travail est différé en DPC.
KPRCB (Kernel Processor Control Block)	Structure per-CPU contenant l'état du scheduler local : current/next/idle thread, ready queues, compteurs de performance.
KTHREAD	Structure kernel représentant un thread. Contient la priorité, l'affinité, le quantum, l'état, les wait blocks — toutes les données de scheduling.
NUMA (Non-Uniform Memory Access)	Architecture mémoire où la latence d'accès dépend de la proximité entre le CPU et la mémoire. Chaque socket constitue typiquement un nœud NUMA.
P-core (Performance core)	Core haute performance dans les architectures hybrides Intel (Golden Cove, Raptor Cove). IPC maximal, consommation plus élevée.
Quantum	Durée maximale d'exécution continue d'un thread avant préemption par le scheduler. ~15.6 ms (client), ~180 ms (serveur). Mesuré en unités internes (1 unité = 1/3 de clock tick).
Ready Queue	File d'attente FIFO per-CPU et per-priorité (32 niveaux) contenant les threads prêts à s'exécuter. Le scheduler sélectionne toujours le premier thread de la plus haute priorité non-vide.
SMT (Simultaneous Multi-Threading)	Technique permettant à un core physique d'exécuter deux threads logiques simultanément en partageant les ressources d'exécution. Hyper-Threading chez Intel.
Thread Director	Mécanisme Intel Hardware Feedback Interface fournissant des indications temps réel au scheduler OS sur les caractéristiques de chaque thread, pour le placement P-core/E-core optimal.

Points clés de cet article :

1. Le scheduler Windows NT est un **scheduler préemptif à priorités fixes** (32 niveaux) avec quantum variable — la priorité la plus haute gagne toujours le CPU.

2. Les structures **KTHREAD** (état du thread) et **KPRCB** (état per-CPU) contiennent toutes les données de scheduling. Leur maîtrise via WinDbg est indispensable pour le diagnostic avancé.

3. **ETW** (Event Tracing for Windows) est l'outil de mesure : les événements CSwitch et ReadyThread fournissent une vue déterministe et complète de chaque décision du scheduler.

4. Les **quantum** diffèrent radicalement entre client (~15.6 ms) et serveur (~180 ms), reflétant le compromis interactivité vs throughput.

5. L'architecture **hybride P/E-core** et le Thread Director Intel transforment le scheduling en un problème de classification de workload, pas seulement de priorité.

6. Les pathologies courantes — **starvation, inversion de priorité, oversubscription, pénalités NUMA** — sont détectables avec ETW et corrigeables par tuning d'affinité, de priorité et de thread pool.

7. La **sécurité** impose des contraintes sur le scheduling : le core scheduler Hyper-V et la désactivation SMT sont des mitigations contre les attaques side-channel microarchitecturales.

8. L'optimisation système (power plans, C-states, RSS, NDIS polling) est aussi importante que l'optimisation applicative pour les workloads latency-sensitive.

9. Le **dimensionnement des thread pools** (= CPUs pour CPU-bound, IOCP pour I/O-bound) est la première optimisation à mettre en œuvre avant toute autre.

10. Les évolutions futures (ML-driven scheduling, hétérogénéité hardware, micro-VMs) vont continuer à complexifier le scheduler, rendant la compréhension des fondamentaux encore plus critique.

FAQ — Questions fréquentes

Qu'est-ce que le scheduler Windows et quel est son rôle exact ?

Le scheduler Windows (ordonnanceur) est le composant du noyau NT responsable de la distribution du temps CPU entre les threads. Son rôle est de décider, à chaque instant, quel thread doit s'exécuter sur chaque processeur logique. Il implémente un algorithme **préemptif à priorités fixes** : les 32 niveaux de priorité (0-31) déterminent l'ordre d'exécution, et un thread de priorité supérieure préempte toujours un thread de priorité inférieure. Le scheduler gère également le quantum (durée maximale d'exécution continue), le boost de priorité (accélération temporaire après une I/O ou une interaction utilisateur), le placement NUMA (affinité mémoire), et depuis Windows 11, le placement hybride P-core/E-core. Le code du scheduler réside

principalement dans `KiSwapThread`, `KiReadyThread` et `KiSelectNextThread` au sein du noyau `ntoskrnl.exe`. Chaque CPU possède ses propres ready queues (32 files, une par priorité) dans sa structure KPRCB, permettant un scheduling per-CPU sans contention de lock globale dans la majorité des cas.

Comment fonctionne le Thread Director d'Intel et quel est son impact ?

Le Thread Director est un composant hardware des processeurs Intel hybrides (12e génération et ultérieures) qui fournit des indications en temps réel au scheduler Windows sur les caractéristiques de chaque thread. Le firmware du CPU exécute un modèle de classification basé sur des métriques microarchitecturales : ratio instructions entières/flottantes, taux de cache miss, utilisation des unités vectorielles (AVX-512 par exemple), et intensité mémoire. Chaque thread est classé dans une catégorie (compute-intensive, memory-bound, vectorized, etc.) et le Thread Director communique ces classifications au scheduler via l'interface ACPI HFI (Hardware Feedback Interface). Le scheduler utilise ces données pour placer les threads compute-intensive sur les P-cores (haute performance) et les threads légers ou d'arrière-plan sur les E-cores (efficacité). L'impact est significatif : sans Thread Director, le scheduler ne peut se baser que sur la priorité et l'historique, et peut placer un thread AVX-heavy sur un E-core (qui n'a pas d'AVX-512) ou un thread idle-mostly sur un P-core gaspillant de l'énergie. Avec Thread Director, le placement est quasi-optimal dans la majorité des cas, avec des gains de 10-20 % en performance et 15-30 % en efficacité énergétique.

Comment diagnostiquer un problème de scheduling sous Windows ?

Le diagnostic suit une approche en trois étapes. **Première étape : observation** — utiliser Task Manager (onglet Performance → CPU) pour identifier la charge globale, puis Resource Monitor (onglet CPU) pour voir les threads individuels et leur temps CPU. Si le problème est évident (un processus à 100 % CPU), cette étape peut suffire. **Deuxième étape : capture ETW** — lancer `wpr -start CPU -start DPC_ISR`, reproduire le problème pendant 15-30 secondes, puis `wpr -stop trace.etl`. Ouvrir la trace dans Windows Performance Analyzer (WPA) et examiner les graphes CPU Usage (Precise) pour la vue déterministe des context switches, Ready (µs) pour la latence de scheduling, et DPC/ISR pour les interférences kernel. **Troisième étape : debug kernel** — pour les problèmes complexes (deadlock, starvation subtile, pathologie driver), connecter WinDbg en kernel debug et utiliser `!ready` pour voir les ready queues, `!running` pour les threads actifs, et `!thread` pour l'état détaillé d'un thread spécifique. La combinaison ETW (vue temporelle) + WinDbg (vue instantanée) couvre la totalité des scénarios de diagnostic. Pour un guide approfondi sur l'utilisation d'ETW en contexte forensique et d'analyse, consultez notre article sur les bonnes pratiques ETW/WPR en forensics.

Quelles sont les différences de scheduling entre Windows 11 et Windows Server 2025 ?

Les différences sont calibrées pour les cas d'usage respectifs. **Quantum** : Windows 11 utilise des quantum courts (~15.6 ms, 2 clock ticks) pour maximiser la réactivité interactive ; Server 2025 utilise des quantum longs (~180 ms, 12 clock ticks) pour maximiser le throughput en minimisant les context switches. **Foreground boost** : Windows 11 applique un boost de quantum ×3 au processus foreground (la fenêtre active), inexistant sur Server (pas d'interface interactive en

usage normal). **Thread Director** : pleinement actif sur Windows 11 pour le placement P-core/E-core ; moins pertinent sur Server qui utilise rarement des CPUs hybrides (les Xeon sont homogènes). **Core parking** : plus agressif sur Windows 11 (économie batterie) ; configurable finement sur Server via les power plans. **NUMA** : Server 2025 intègre des optimisations NUMA plus avancées pour les systèmes multi-socket (placement inter-nœud intelligent, heuristiques de mémoire locale), tandis que Windows 11 gère rarement plus d'un nœud NUMA. Pour une vue plus large de l'architecture serveur, voir notre article sur l'architecture système de Windows Server 2025.

Comment optimiser le scheduling pour les applications multi-core intensives ?

L'optimisation commence par le **dimensionnement du thread pool** : pour un workload CPU-bound pur, le nombre de threads doit égaler le nombre de CPUs logiques (ou physiques si le SMT est désactivé). Pour un workload I/O-bound, utiliser les I/O Completion Ports (IOCP) qui maintiennent automatiquement le bon niveau de concurrence. Ensuite, considérer l'**affinité** : pour les workloads latency-sensitive, utiliser `SetThreadIdealProcessor` (soft affinity) pour réduire les migrations sans empêcher le load balancing. Pour les workloads HPC, le hard affinity (`SetThreadAffinityMask`) avec un thread par core élimine toute interférence de migration. Sur les systèmes NUMA, allouer la mémoire sur le même nœud que les threads avec `VirtualAllocExNuma`. Côté système, désactiver le core parking (`powercfg` High Performance ou `MinimumUnparkedProcessorPercentage = 100`), limiter les C-states à C1 pour réduire la latence de réveil, et configurer RSS pour aligner les interruptions réseau avec les threads applicatifs. Pour les workloads cryptographiques ou de bruteforce, la considération sécurité s'ajoute : évaluer le risque side-channel SMT (voir notre article sur l'escalade de privilèges sous Windows Server 2025) et envisager la désactivation SMT si des données sensibles sont traitées.

Ressources complémentaires

- **Windows Internals, 7th Edition, Part 1** (Yosifovich, Ionescu, Russinovich, Solomon) — Chapitres 4 et 5, la référence absolue sur le scheduler NT. Microsoft Press.
- **Microsoft Learn — Thread Scheduling** — Documentation officielle : Scheduling, Scheduling Priorities, Processor Groups.
- **Windows Performance Toolkit** — WPR et WPA documentation, inclus dans le Windows ADK.
- **Sysinternals Suite** — Process Explorer, Process Monitor, RAMMap — outils de Mark Russinovich pour l'analyse système.
- **Intel Thread Director** — Documentation Intel sur l'intégration hardware/scheduler.
- **Hyper-V Scheduler Types** — Root scheduler vs Core scheduler, implications sécurité et performance.
- **ETW et forensics** — Bonnes pratiques ETW/WPR en forensics (Ayi NEDJIMI Consultants).
- **Architecture Windows Server 2025** — Architecture système de Windows Server 2025 (Ayi NEDJIMI Consultants).
- **Sécurité Windows** — Escalade de privilèges sous Windows Server 2025 (Ayi NEDJIMI Consultants).

Ayi NEDJIMI Consultants — Expert cybersécurité offensive & intelligence artificielle

ayinedjimi-consultants.fr · ayi@ayinedjimi-consultants.fr