

**GESTION DES PROCESSUS
SOUS WINDOWS**

REFERENCE: ATLANTIC/DOCTECH/
Systèmes d'exploitation/Gestion des processus sous windows
Mise à jour: 04/07/11



DOMAINE:
Systèmes d'exploitation

**GESTION DES PROCESSUS
SOUS WINDOWS**

VERSIONS & DATES	OBJET	AUTEUR(S)
<i>Version 1.0 – Janvier 2010</i>	<i>Création</i>	<i>Bernard GIACOMONI - Association A.T.L.A.N.T.I.C</i>

SOMMAIRE

I.GENERALITES:	4
I.1.OBJET DU DOCUMENT:	4
I.2.RAPPELS: APPLICATIONS TEMPS REEL:	4
I.3.SYSEME WINDOWS ET TEMPS REEL:	4
II.RAPPELS PRELIMINAIRES:	5
III.CARACTERISTIQUES DU NOYAU WINDOWS:	6
III.1.GENERALITES:	6
III.1.1.NOYAU DU SYSEME D'EXPLOITATION WINDOWS:.....	6
III.2.TRAITEMENT MULTI-TACHES:	6
III.2.1.LA NOTION DE PROCESSUS SOUS WINDOW:.....	6
III.2.2.LA NOTION DE THREAD SOUS WINDOW:.....	6
III.2.3.LA COMMUNICATION INTER-PROCESSUS:.....	7
III.2.4.ORDONNANCEMENT DES TACHES ET DES PROCESSUS:.....	7
III.2.4.1.GENERALITES:.....	7
III.2.4.2.FONCTIONNEMENT:.....	7
III.2.4.3.IMPLANTATION DE LA NOTION DE PRIORITES PREEMPTIVES:.....	7
III.2.4.4.TRAITEMENT AUTOMATIQUE DES PRIORITES.....	8
III.2.4.5.APTITUDE AU TRAITEMENT EN TEMPS REEL:.....	8
IV.CARACTERISTIQUES DE L'A.P.I. WIN32:	10
IV.1.RAPPEL SUR LE TYPAGE DES DONNEES:	10
IV.2.OBJETS DU NOYAU WIN32:	10
IV.2.1.GENERALITES:.....	10
IV.2.2.MANIPULATION DES OBJETS DU NOYAU:.....	10
V.OBJETS KERNEL DE GESTION DES PROCESSUS:	12
V.1.LES OBJETS PROCESSUS:	12
V.1.1.GENERALITES:.....	12
V.1.2.DECLARATION D'UN PROCESSUS:.....	12
V.1.3.CREATION DYNAMIQUE D'UN PROCESSUS-FONCTION CreateProcess:.....	13
V.1.4.TERMINER UN PROCESSUS-FONCTIONS ExitProcess ET TerminateProcess:.....	14
V.1.5.OBTENIR UN HANDLER SUR UN PROCESSUS-FONCTIONS GetCurrentHandle et GetModuleHandle:.....	15
V.1.6.OBTENIR LE PID D'UN PROCESSUS-FONCTIONS GetCurrentProcesId et GetProcessId:.....	15
V.1.7.MODIFIER LA CLASSE DE PRIORITE D'UN PROCESSUS-FONCTION SetPriorityClass:.....	15
V.2.LES OBJETS THREADS:	16
V.2.1.DECLARATION D'UN THREAD:.....	16
V.2.2.CREER D'UN THREAD-FONCTION CreateThread:.....	16
V.2.3.TERMINER UN THREAD-FONCTIONS ExitThread ET TerminateThread:.....	16
V.2.4.CREER UN HANDLER SUR LE THREAD EN COURS:.....	16
V.2.5.MODIFIER LA PRIORITE D'UN THREAD-FONCTION SetThreadPriority:.....	17
V.2.6.OBTENIR LE PID D'UN PROCESSUS-FONCTIONS GetCurrentProcesId et GetProcessId:.....	17
V.3.SYNCHRONISATION DES THREADS:	18
V.3.1.GENERALITES:.....	18
V.3.2.FONCTIONS DE SYNCHRONISATION-WaitForSingleObject ET WaitForMultipleObjects:.....	18
V.3.2.1.WaitForSingleObject:.....	18
V.3.2.2.WaitForMultipleObjects:.....	19
V.3.3.SYNCHRONISATION PAR LES SECTIONS CRITIQUES:.....	19
V.3.4.SYNCHRONISATION PAR LES OBJETS MUTEX:.....	20

V.3.5.SYNCHRONISATION PAR LES OBJETS SEMAPHORES:.....	21
V.3.6.SYNCHRONISATION PAR LES OBJETS EVENEMENTS:.....	23
V.4.PARTAGE DE DONNEES ENTRE PROCESSUS:.....	25
V.4.1.GENERALITES:.....	25
V.4.2.TRANSMISSION DE DONNEES PAR LE MECANISME DES SOCKETS:.....	25
V.4.3.PARTAGE DE DONNEES PAR LES FICHIERS:.....	25
V.4.4.PARTAGE DE DONNEES PAR DES ZONES MEMOIRES PARTAGEES:.....	25
V.4.4.1.MECANISME DE PROTECTION MEMOIRE INTER-PROCESSUS:.....	25
V.4.4.2.MECANISME DE PARTAGE MEMOIRE INTER-PROCESSUS:.....	26
V.4.4.3.LES OBJETS FILE MAPPING DE WINDOWS:.....	27
VI.EXEMPLES DE CODES.....	29
VI.1.CREATION D'UN PROCESSUS:.....	29
VI.2.TERMINAISON D'UN PROCESSUS:.....	30
VI.3.GESTION DES THREADS:.....	32
VI.4.PARTAGE DE DONNEES ENTRE PROCESSUS (OBJET FILE_MAPPING):.....	33
VI.5.UTILISATION D'UN TIMER.....	34

I.GENERALITES:

I.1.OBJET DU DOCUMENT:

Cet ouvrage s'adresse à des développeurs maîtrisant la programmation en langages C/C++. Son objectif est de leur fournir les bases nécessaires pour aborder la conception et le développement d'applications multitâches à contraintes temps réel sous les systèmes Windows équipés du noyau WIN32 (C'est à dire toutes les versions à partir de NT, VISTA incluse).

I.2.RAPPELS: APPLICATIONS TEMPS REEL:

Une application informatique est dite «**en temps réel**» (il serait plus exacte de dire «a contraintes temps réel») lorsque certains des signaux ou des données qu'elle élabore doivent impérativement être délivrés à la périphérie à des dates situées à l'intérieur de plages horaires strictes, sous peine de perdre toute utilité, ou même d'occasionner des dysfonctionnements du système contrôlé.

L'application est donc soumise à une exigence de **déterminisme temporel** plus ou moins forte. Ceci implique que le système d'exploitation permette une maîtrise suffisante du «timing» d'exécution des processus informatiques composant l'application. De tels systèmes d'exploitation sont dits «à capacités temps réel».

Les contraintes de type temps réel sont communément classées en trois niveaux:

- *Des contraintes temps réel sont dites «**strictes**» lorsque le déterminisme temporel est inférieur à la milliseconde (en pratique, souvent beaucoup moins) et doit impérativement être respecté. Les anglophones utilisent l'expression «**hard real Time**» pour désigner ce niveau de contrainte.*
- *Les contraintes temps réel sont dites «**souples**» lorsque l'on admet un certain pourcentage de non respect des plages de dates **ou bien** lorsque la largeur de ces plages est de l'ordre de quelques centaines de millisecondes (ce qui constitue le déterminisme temporel habituel des systèmes en temps partagé pour la gestion des processus). Les anglophones utilisent l'expression «**Soft real Time**» pour désigner ce niveau de contrainte.*
- *Entre ces deux niveaux se situe le domaine du temps réel «**médian**» (**Medium real time** en anglais). En général, le Temps Réel Médian concerne des applications pour lesquelles les plages de dates doivent être strictement respectées mais sont relativement larges (par exemple: 10 millisecondes).*

I.3.SYSEME WINDOWS ET TEMPS REEL:

Le système WINDOWS est relativement peu utilisé en informatique industrielle, en particulier dans les applications contraintes temps réel. Dans ces domaines, les développeurs lui préfèrent des systèmes d'exploitation spécialisés, souvent basés sur UNIX (OS9, VxWorks, Lynx-OS, RT-Linux, etc...). Les raisons les plus souvent invoquées sont le faible déterminisme temporel de ses mécanismes de gestion multitâches, la lenteur de certains traitements système et une fiabilité jugée parfois insuffisante.

En fait, si ces réserves semblent en partie justifiées, elles ne sont vraiment pertinentes que pour le développement d'applications supportant des contraintes temps réel qualifiées de «strictes». En revanche, il est souvent possible d'utiliser WINDOWS comme support d'applications d'informatique industrielle à contraintes temps réel plus souples et dont la criticité n'est pas trop élevée. En effet, nous allons voir dans la suite de cet ouvrage que le noyau WIN32 implémente des mécanismes analogues à ceux que l'on trouve dans les systèmes d'exploitation dédiés au temps réel (et en particulier, une grande partie des mécanismes recommandés par la norme POSIX 4-UNIX TEMPS REEL), et que le déterminisme temporel est suffisant pour satisfaire au temps réel « médian ».

Comme d'autre part, et du moins jusqu'à aujourd'hui, le fait de travailler sous WINDOWS permet de bénéficier d'outils de développement plus nombreux et plus évolués que ceux que l'on trouve sous les autres systèmes d'exploitation, le choix de WINDOWS pour certaines applications industrielle peut s'avérer judicieux en terme de délais de développement.

II. RAPPELS PRELIMINAIRES:

TERME OU SIGLE	DEFINITION
NOYAU d'un système d'exploitation	<p>On appelle NOYAU (KERNEL en anglais) d'un système d'exploitation l'ensemble des logiciels supportant les fonctionnalités de base de celui-ci, c'est à dire celles qui ne dépendent pas de la plate-forme matérielle:</p> <ul style="list-style-type: none">• Ordonnancement et synchronisation des tâches.• Gestion d'événement.• Gestion du système de fichiers• Gestion du temps (horloge, timers, etc.).• etc. <p>Le reste du système d'exploitation est constitué essentiellement par les gestionnaires d'équipements matériels (exemple: un pilote de périphérique, qui gèrent la communication entre le noyau et un interface d'entrée-sortie matériel).</p>
A.P.I	<p>Application Programming Interface: interface de programmation permettant à une application informatique de communiquer avec le système d'exploitation. Un A.P.I. Se présente comme une bibliothèque de fonctions assurant l'interface de programmation entre une application et un système d'exploitation (et en particulier avec les fonctions du noyau)</p>
Objet du noyau	<p>Pour l'A.P.I. WINDOWS, les différentes fonctions systèmes sont vues comme des «méthodes» activant des «objets du noyau» («kernel objects»).</p>
Handle	<p>Littéralement «poignée». Un handle est un nombre entier de 32 bits. Il s'agit en fait d'un pointeur d'accès à une ressource système.</p>
Sections critiques	<p>Définissent des segments de code pouvant être protégés par un mécanisme d'exclusion mutuelle des threads d'un même processus.</p>
Heap	<p>Zone de mémoire dynamique allouée pour un usage donné (à un processus, par exemple).</p>
Thread	<p>Littéralement: «filin»: Segment de code exécutable correspondant à un algorithme séquentiel et destiné à être exécuté en parallèle avec d'autres threads dans un environnement multitâches.</p>
Processus	<p>Un processus est un objet auquel sont attribués un espace mémoire propre et protégé des autres processus (contenant ses données et son code exécutable), et des ressources systèmes. A un instant donné, le code exécutable du processus peut être exécuté par un ou plusieurs threads qui ont accès à la totalité de son espace mémoire et de ses ressources.</p>

III.CARACTERISTIQUES DU NOYAU WINDOWS

III.1.GENERALITES:

III.1.1.NOYAU DU SYSTEME D'EXPLOITATION WINDOWS:

Les version NT et supérieures du système d'exploitation WINDOWS (y compris VISTA) utilisent le même noyau, quelquefois appelé WIN32. Ce noyau présente les caractéristiques suivantes:

- Comme son nom l'indique, il est conçu pour tourner sur des plates-formes 32 bits.
- Il est capable de traiter des configurations multi-processeurs d'une manière transparente pour le développeur.
- Enfin, il supporte un véritable traitement multi-tâches et des mécanismes de préemptivité immédiate en fonction de priorités qui le rendent apte dans certaines limites au temps réel.

REMARQUES:

- Dans les versions antérieures à NT, seul le processus lié à la fenêtre active était réellement actif. Les autres étaient à l'état dormant.
- Le noyau WINDOWS adapté aux processeurs 64 bits est souvent appelé WIN64. Ses fonctionnalités ne diffèrent pas notablement de celles de WIN32. Les applications développées pour l'environnement WIN32 sont en général portables sans modification sous WIN64.

III.2.TRAITEMENT MULTI-TACHES:

III.2.1.LA NOTION DE PROCESSUS SOUS WINDOW:

Sous WINDOW, comme sous tout système d'exploitation multitâche, un processus peut être défini comme une instance d'une application «active», c'est à dire **en cours d'exécution** ou en **attente d'exécution**. Une application Win32 possède un «processus racine» (ou processus père), à partir duquel peuvent être créés des processus fils. La fenêtre principale de l'application appartient au processus père.

Un processus dispose:

- D'un espace d'adressage propre (4 Go) contenant son code exécutable et ses données (fichier «exe»), et les codes et données des D.L.L (Dynamic Loadable Librarys ou bibliothèques dynamiques) requises.
- D'un certain nombre de ressources (fichiers, allocation dynamique de mémoires, fenêtre graphique, threads...), qui lui sont allouées pendant la durée de sa «vie».

A l'image de ce qui se passe dans un système basé sur UNIX, le système d'exploitation windows attribue à chaque processus un identificateur unique, le **Process Identificator**, abrégé en «**PID**». Le PID est un nombre entier positif. Il permet d'identifier sans équivoque un processus pendant toute la durée de son exécution. Après la fin d'exécution d'un processus, la valeur de son PID est réallouée par le système à un nouveau processus, en fonction des besoins.

III.2.2.LA NOTION DE THREAD SOUS WINDOW:

Lorsqu'un processus Window vient d'être créé, il est «inerte» ou «dormant». Pour qu'il s'exécute, il faut que le scheduler lance l'exécution d'un segment du code exécutable du processus. Ce segment de code est appelé «thread» (filin, fil en anglais). En termes plus techniques, on dit que le scheduler attribue le «flot de contrôle» du processeur à ce processus.

Pour exécuter du code, il faut donc que le processus possède au moins un «thread», sont thread «principal». Nous verrons plus tard qu'en langage C, ce thread principal débute à la section WinMain (ou main, si l'application est «monoconsole», qui marque le début du code objet exécutable d'un processus.

Le noyau WIN32 peut exécuter plusieurs threads en parallèle, le parallélisme pouvant être réel (dans un système multi-processeur) ou apparent (système monoprocesseur). Un processus peut posséder plusieurs threads, se déroulant en parallèle avec les autres threads actifs (de ce processus et des autres processus). Les ressources d'un processus sont partagées par tous ses threads (et en particulier, son espace mémoire: une pile est allouée à chaque thread dans cet

espace).

Lorsqu'un processus ne possède plus aucun thread actif, il est éliminé par le système et ses ressources sont libérées.

REMARQUES:

- Un thread ne possède donc pas d'espace mémoire propre, puisqu'il utilise celui du processus auquel il est attaché. De ce fait, on appelle parfois les threads «**processus légers**».
- A l'image de ce qui se passe dans un système basé sur UNIX, le système d'exploitation windows attribue à chaque thread un identificateur unique, sous la forme d'un entier positif.

III.2.3.LA COMMUNICATION INTER-PROCESSUS:

L'espace d'adressage d'un processus est protégé contre toute intrusion d'un autre processus. De ce fait, Win32 intègre un certain nombre de mécanismes permettant d'échanger des données entre deux processus. Ces mécanismes sont le plus souvent basés sur la technique des fichiers «mappés» en mémoire (ce qui veut dire que leur contenu n'est pas situé sur une mémoire de masse mais dans la mémoire vive). Ce mécanisme est assez analogue aux «shared memories» des systèmes de type UNIX-LINUX. Des mécanismes de signaux et de synchronisation sont également disponibles.

III.2.4.ORDONNANCEMENT DES TACHES ET DES PROCESSUS:

III.2.4.1.GENERALITES:

Les systèmes Windows sont, à la base, des systèmes de traitement en temps partagé (time shearing), analogues à un UNIX classique dans leur principe, du moins à partir de Windows 3.1. Les versions du système Windows basées sur WIN32 intègrent à cette structure un traitement multitâches à priorités préemptives. En effet:

- Une notion de priorité est associée au traitement des threads par le scheduler (32 niveaux de priorité, du plus faible = 0 au plus fort = 31)
- Tout thread actif (non en attente passive) est **préemptif** par rapport aux threads de niveau prioritaire inférieur. Ceci signifie que dès qu'un thread en attente devient actif, il interrompt immédiatement tout thread dont le niveau de priorité est inférieur au sien.
- Le scheduling de différents threads ayant même priorité s'effectue suivant la technique du «temps partagé», le système allouant cycliquement à chacun une tranche fixe de temps CPU («round robbin»).

III.2.4.2.FONCTIONNEMENT:

Supposons qu'à un instant donné, la plus forte priorité de threads actifs soit n (le «round robbin» s'effectue donc entre tous les threads de priorité n). Soit T1 le thread de priorité n en cours d'exécution:

SI un thread T2 possédant une priorité plus élevée ($> n$) devient actif, pendant la plage de temps partagé allouée à T1, par exemple parce qu'il vient d'être lancé ou libéré de l'attente d'une ressource, T2 interrompt immédiatement le thread T1.

SINON le thread T1 ira au bout de sa plage allouée, puis laissera place à un autre thread actif de même priorité ou, s'il n'en existe plus d'éligible, de priorité inférieure.

III.2.4.3.IMPLANTATION DE LA NOTION DE PRIORITES PREEMPTIVES:

Contrairement aux systèmes UNIX dits «temps réel» (c'est à dire répondant à la norme POSIX4 ou aux normes dérivées), la notion de priorité préemptive est implantée dans WIN32 par l'intermédiaire des threads et non des processus. Ceci n'a, en fait, pas beaucoup d'importance du point de vue du fonctionnement, car dans tous les cas, un scheduler ordonnance des threads et non des processus. Un processus n'est actif que s'il possède au moins un thread actif: ordonner ce thread revient à ordonner l'exécution du processus par rapport aux autres processus.

Win32 de Windows permet d'affecter à un thread un niveau de priorité préemptive, pris parmi 32 niveaux possibles. Cependant, l'attribution de ces priorités aux threads n'est pas aussi souple qu'elle peut l'être sous un système UNIX multitâche préemptif, car elle se combine avec la notion de «classe de priorité des processus»:

La «Classe de Priorité» d'un processus est fixée à la création de celui-ci. Il existe 4 classes distinctes. Chaque classe

définit un niveau de priorité qui est attribué par défaut à tous les threads du processus. Le tableau ci-dessous décrit les caractéristiques des ces classes:

CLASSE	Flag correspondant dans la primitive CreateProcess de l'API Win32	NIVEAU DE PRIORITE PAR DEFAUT
Idle	IDLE_PRIORITY_CLASS	4
Normal	NORMAL_PRIORITY_CLASS	7
Hight	HIGH_PRIORITY_CLASS	13
Real Time	REALTIME_PRIORITY_CLASS	24

D'autre part, il est possible (en positionnant des flags avec la fonction SetThreadPriority de l'API) de modifier la priorité relative d'un thread par rapport à la priorité de base de son processus. Le tableau ci-dessous définit les différentes valeurs de «flag» possibles et leurs effets sur la priorité du thread auquel ils sont attribués:

VALEUR DE FLAG (Passée par la fonction SetThreadPriority)	EVOLUTION DE LA PRIORITE DU THREAD PAR RAPPORT LA PRIORITE DE BASE FIXEE PAR LA CLASSE DU PROCESSUS
THREAD_PRIORITY_LOWEST	-2
THREAD_PRIORITY_BELOW_NORMAL	-1
THREAD_PRIORITY_NORMAL	Inchangée
THREAD_PRIORITY_ABOVE_NORMAL	+1
THREAD_PRIORITY_HIGHEST	+2
THREAD_PRIORITY_IDLE	Priorité fixée à 1, sauf pour la classe REAL_TIME de processus, pour laquelle elle est fixée à 16.
THREAD_PRIORITY_TIME_CRITICAL	Priorité fixée à 15, sauf pour la classe REAL_TIME de processus, pour laquelle elle est fixée à 31.

EXEMPLES:

- Par défaut, un thread d'un processus de classe REALTIME_PRIORITY_CLASS se voit attribuer la priorité 24.
- Si on lui applique le flag THREAD_PRIORITY_LOWEST, sa priorité devient $24-2 = 22$.
- Si on lui applique le flag THREAD_PRIORITY_IDLE, il se voit attribuer la priorité 16.

III.2.4.4. TRAITEMENT AUTOMATIQUE DES PRIORITES

- En l'absence de spécifications de classe de priorité, la plupart des processus utilitaires de Windows sont créés dans la classe normale.
- Windows majore automatiquement de 2 la priorité du processus possédant la fenêtre sur laquelle l'utilisateur travaille, de façon à favoriser son affichage.

III.2.4.5. APTITUDE AU TRAITEMENT EN TEMPS REEL:

Ces mécanismes permettent effectivement de créer des threads en choisissant de leur attribuer une priorité choisie parmi 32 niveaux de priorité préemptives. Cependant, il faut reconnaître que le mécanisme manque un peu de souplesse.

D'autre part, nous ne pouvons pas vraiment parler d'aptitude au «temps réel dur». En effet, les mécanismes d'ordonnement et de synchronisation ne garantissent pas «à tout coup» les temps de réponse: le déterminisme n'est donc pas complètement assuré, ce qui peut être une contre-indication pour certaines applications sensibles, notamment dans le domaine de l'informatique industrielle. L'indéterminisme temporel de la gestion des tâches reste cependant assez faible (de l'ordre de 10 ms pour Window NT et XP, pourvu qu'on ne touche pas aux fenêtres affichées).

Cependant, nous avons pu vérifier que, sous windows (des versions NT à XP, à l'exception de Window95), et à condition de disposer d'un processeur assez puissant (au moins 700 mhz), les temps de réponse aux interruptions et la durée des changements de contextes sont assez proches de la milliseconde, ce qui permet de l'employer dans des

contextes «temps réel» autres que ceux que l'on désigne habituellement par «hard real time» en anglais ou «temps réel strict» en français.

REMARQUE:

Des solutions WINDOWS telles que l'extension temps réel RTX (ex: version RTX 5.1, distribué, par la société VenturCom depuis le 25/10/2001) applicables aux versions de Windows NT, 2000, XP et Embedded (Window CE) permettent d'atteindre un niveau de performances «temps réel» plus élevé.

IV.CARACTERISTIQUES DE L'A.P.I. WIN32:

IV.1.RAPPEL SUR LE TYPAGE DES DONNEES:

L'API Win32 définit et utilise un certain nombre de noms de types de données qui lui sont propres. La plupart se réfèrent à des types connus en langage C. En général, le nom du type de base est écrit en majuscule. Il est précédé des lettres P pour pointeur, LP pour pointeur long, H pour Handler, etc. Par exemple:

NOM DE TYPE WIN32	SIGNIFICATION
DWORD	Entier de 4 octets (identique au type long du C)
LPDWORD	Pointeur long sur entier long
HANDLE	Handler d'un objet du noyau
HINSTANCE	Handle d'Instance sur un objet processus
LPSTR	Pointeur long sur une chaîne
LPVOID	Pointeur long non typé
LPSECURITY_ATTRIBUTES	Pointeur sur l'attribut SECURITE d'un thread
BOOL	Variable logique booléenne (true/false)

Etc...

IV.2.OBJETS DU NOYAU WIN32:

IV.2.1.GENERALITES:

Le système d'exploitation Windows gère les ressources systèmes par l'intermédiaire de bibliothèques d'objets: les «objets du noyau» («kernel objects»). Ces objets encapsule les différents types d'entités systèmes: fenêtres, processus, users, GDI (Graphic Device Interface), etc.

Ils mettent à disposition des développeurs des méthodes qui leur permettent d'agir sur ces entités (pour les créer, les gérer, les supprimer, etc.). l'ensemble de ces méthodes constitue un A.P.I (Application Programming Interface). Par rapport à ses prédécesseurs, l'API WIN32 permet de gérer un certain nombre de ressources nouvelles en rapport avec les capacités multitâches du noyau WIN32: threads, événements, sémaphores, mutex, etc.

Un objet du noyau est accessible par toutes les applications actives. De ce fait, ces objets permettent les interactions entre deux processus différents (en particulier, la synchronisation des threads).

IV.2.2.MANIPULATION DES OBJETS DU NOYAU:

- La manipulation des objets du noyau à partir d'un processus donné s'effectue par l'intermédiaire d'un handler. Un handler est une donnée de type HANDLE, qui permet à un processus donné de «pointer» vers un objet du noyau donné.
- Les objets du noyau peuvent être partagés entre plusieurs processus. Or, un handler n'est valide qu'à l'intérieur d'un processus donné. De ce fait, chaque processus utilisateur doit obtenir un handler sur les objets du noyau qu'il veut utiliser.
- Le processus qui crée l'objet le fait par la procédure de **création** de l'objet (CreateWindow, CreateThread, CreateMutex, etc...). Lors de la création, un **nom** est attribué à l'objet.
- Une fois un objet du noyau créé par un processus donné, un autre processus pourra obtenir un handler sur cet objet en utilisant une fonction d'**ouverture** (OpenMutex, par exemple). Pour spécifier l'objet à ouvrir, il utilisera le nom de cet objet (passé en paramètre de la procédure de création).

- Tous les threads d'un processus pourront, bien sûr, se partager le handler obtenu par création ou ouverture.
- La fonction CloseHandle permet de détruire le handler sur un objet. Utilisée par le processus créateur, cette fonction détruit également l'objet lui-même. Seul le créateur d'un objet du noyau peut le détruire.

V.OBJETS KERNEL DE GESTION DES PROCESSUS

V.1.LES OBJETS PROCESSUS:

V.1.1.GENERALITES:

Les objets **PROCESSUS** permettent de manipuler les processus des systèmes Windows.

RAPPELS:

- Un processus est une **instance** d'un programme donnée **en cours d'exécution**. De ce fait, plusieurs objets processus peuvent se voir attribuer le même code exécutable.
- Le système identifie les processus grâce à l'identificateur qui leur attribue lors de leur lancement, le P.I.D (Process Id). Il existe donc deux moyens d'identifier un processus: par le handler de l'objet processus correspondant, valide dans le processus en cours et par le PID de ce processus, valide pour tous les processus.
- A chaque processus est attribué un espace mémoire qui lui est propre: il est donc impossible à un processus donné d'accéder directement à l'espace mémoire d'un autre processus: pour communiquer entre processus, il faut donc recourir à des outils systèmes.

V.1.2.DECLARATION D'UN PROCESSUS:

Le code objet d'un processus débute par la déclaration de sa procédure principale (qui sera également son thread principal). La fonction API Win32 WinMain permet de déclarer une telle section de code, dont le schéma de base est le suivant:

```
int WINAPI WinMain (    HINSTANCE  hinstExe    // Handler d'instance du processus (en
                        // fait, adresse de base du processus)
                      , HINSTANCE  hinsPrev  // Handler de l'instance précédente
                        // du processus
                      , LPSTR      lpstCmdLine // Pointeur sur la ligne de commande
                        // passée au processus
                      , int         nCmdShow  // Spécifications concernant la fenêtre
                        // principale de l'application.
                      )
{
    // -----
    // --- Corps du processus ---
    // -----
}
```

Le paramètre nCmdShow est en fait une file d'indicateurs binaires permettant de spécifier le mode d'affichage de la fenêtre principale du processus. Chaque indicateur binaire correspond à un «define», dont voici les libellés et la signification (Ces indicateurs sont identiques à ceux qu'utilise la fonction ShowWindow):

INDICATEURS D'AFFICHAGE D'UNE FENÊTRE	
SW_HIDE	Cache la fenêtre et en active une autre
SW_MAXIMIZE	Maximise la fenêtre.
SW_MINIMIZE	Minimise la fenêtre et active la dernière fenêtre de niveau supérieur.
SW_RESTORE	Active et affiche la fenêtre. Permet également de ramener à sa valeur normale une fenêtre maximisée ou minimisée
SW_SHOW	Active une fenêtre et l'affiche dans sa position et sa taille courante.
SW_SHOWMAXIMIZED	Active une fenêtre et l'affiche comme une fenêtre maximisée.
SW_SHOWMINIMIZED	Active une fenêtre et l'affiche comme une fenêtre minimisée.
SW_SHOWMINNOACTIVE	Affiche sans l'activer une fenêtre comme une fenêtre minimisée.

SW_SHOWNA	Affiche sans l'activer une fenêtre dans sa position et sa taille courante.
SW_SHOWNOACTIVATE	Affiche sans l'activer une fenêtre avec ses dernières positions et dimensions.
SW_SHOWNORMAL	Active et affiche une fenêtre en rétablissant sa position et ses dimensions courantes. Cet indicateur doit être spécifié quand une fenêtre est affichée pour la première fois.

REMARQUES:

- Les instructions WinMain (...) ou main() ne correspondent pas à l'appel d'une fonction: en fait, il s'agit plutôt de la **déclaration** d'une fonction C d'un type spécial (WINAPI). C'est pour cela que la liste entre parenthèse correspond une déclaration d'arguments (munis de leur type) et non à une liste de paramètres.
- La fonction WinApi permet d'initialiser l'application et d'afficher sa fenêtre principale. D'autre part, elle initialise le mécanisme de «boucle de messages» permettant à l'application de détecter et de traiter les événements liés à l'interface fenêtré (*boucle while (GetMessage....) { TranslateMessage et DispatchMessage }*).
- La fonction WinMain doit être obligatoirement utilisée lorsqu'on veut développer un logiciel permettant de manipuler le système de fenêtrage graphique. En revanche, si l'application n'utilise pas ce genre d'interface, on peut utiliser le schéma classique du langage C:

```
main ()
{
    // -----
    // --- Corps du processus ---
    // -----
}
```

Il s'agit alors d'une application de type «console», qui communique avec l'exploitant par l'intermédiaire d'une «fenêtre texte».

V.1.3.CREATION DYNAMIQUE D'UN PROCESSUS-FONCTION CreateProcess:

A chaque processus correspond un programme informatique, matérialisé par un fichier exécutable (fichier *.exe). La création du processus revient à lancer l'exécution de ce fichier dans l'espace d'exécution alloué au processus. Ceci entraîne la création d'une instance exécutable de ce programme en mémoire vive et le lancement du thread principal du processus (correspondant au code objet débutant par WinMain ou main) par le scheduler.

Un processus peut être lancé soit par une action de l'opérateur (double clic sur le fichier *.exe, par exemple), soit par l'intermédiaire d'un fichier de commande, soit à partir d'un processus déjà actifs, en utilisant la fonction CreateProcess:

```
BOOL CreateProcess
(
    LPCTSTR          lpszImageName // Nom du fichier executable (Default: NULL)
    LPCTSTR          lpszCommandLine // Pointeur sur ligne commande a lancer au
    // départ du processus(Default: NULL)
    LPSECURITY_ATTRIBUTES lpsaProcess // Attribut sécurité relatif au
    // processus(NULL=>héritage des attributs du
    // processus père)
    LPSECURITY_ATTRIBUTES lpsaThread // Attribut sécurité relatif au thread
    // (NULL=>héritage des attributs du
    // processus père)
    BOOL            fInheritHandles // Héritage des handles par les processus
    // fils (true/false) (Default: false)
    DWORD           fdwCreate // flags affectant le nouveau processus
    // (mode debugg, etc...) (Default: 0)
    LPVOID          lpvEnvironment // pointeur sur chaines définissant
    // l'environnement (NULL=>héritage de l'envi-
    // ronnement du processus père)
    LPTSTR          lpszCurDir // Lecteur et répertoire actifs (Default: NULL)
    LPSTARTUPINFO   lpsiStartInfo // Pointeur sur structure _STARTUPINFO
    // concernant la fenêtre d'affichage
    // du processus
    LPPROCESS_INFORMATION lppiProcInfo // Pointeur vers structure
    // _PROCESS_INFORMATION contenant des
    // informations relatives au processus
    // (disponibles en retour de la fonction)
```

```
);
```

STRUCTURE STARTUPINFO:

```
typedef struct _STARTUPINFO
{
    DWORD cb; // Taille en octets de la structure
    LPTSTR lpReserved; // Réserve (défaut: NULL)
    LPTSTR lpDesktop; // Chaîne de caractères qui permet de définir le bureau et
                    // la fenêtre attribués au processus.
                    // - SI = NULL, le process hérite du père.
                    // - SI chaîne vide, le système peut créer un nouveau bureau
                    // et/ou une nouvelle fenêtre.
    LPTSTR lpTitle; // Titre de la nouvelle fenêtre console (ou NULL)
    DWORD dwX; // Si dwFlags contient STARTF_USEPOSITION, position
    DWORD dwY; // en pixel de l'angle haut-gauche de la fenêtre.
    DWORD dwXSize; // Largeur de la nouvelle fenêtre console en pixel
    DWORD dwYSize; // Hauteur de la nouvelle fenêtre console en pixel
    DWORD dwXCountChars; // Si dwFlags contient STARTF_USECOUNTCHARS, nombre de
    DWORD dwYCountChars; // caractères de la nouvelle fenêtre en largeur et hauteur
    DWORD dwFillAttribute; // Si dwFlags contient STARTF_USEFILLATTRIBUTE, permet
                    // de définir les couleurs de fond et de caractères de la
                    // nouvelle fenêtre.
    DWORD dwFlags; // File d'indicateurs binaires déterminant les membres de la
                    // structure dont les valeurs seront prises en compte lors
                    // de la création du processus. Les valeurs de ces
                    // indicateurs correspondent à des «defines»:
                    // STARTF_USESHOWWINDOW, STARTF_USEPOSITION, STARTF_USESIZE
                    // STARTF_USECOUNTCHARS, STARTF_USEFILLATTRIBUTE,
                    // STARTF_FORCEONFEEDBACK, STARTF_FORCEOFFFEEDBACK,
                    // STARTF_USESTDHANDLES.
    WORD wShowWindow; // Si dwFlags contient STARTF_USESHOWWINDOW, Ce membre
                    // permet de spécifier certaines modalités d'affichage
                    // de la fenêtre, sous forme d'une file d'indicateurs
                    // binaires identiques à ceux utilisés par la fonction
                    // ShowWindow (voir tableau ci-dessus)
    WORD cbReserved2; // Réserve (défaut: NULL)
    LPBYTE lpReserved2; // Réserve (défaut: NULL)
    HANDLE hStdInput; // Si dwFlags contient STARTF_USESTDHANDLES, ces trois
    HANDLE hStdOutput; // membres retournent des handlers sur les flux standards
    HANDLE hStdError; // d'entrée, de sortie et d'erreur du processus
};
```

STRUCTURE PROCESS_INFORMATION:

```
typedef struct _PROCESS_INFORMATION
{
    HANDLE hProcess; // Handler sur le processus crée
    HANDLE hThread; // Handler sur le thread principal du processus crée
    DWORD dwProcessId; // PID du processus crée
    DWORD dwThreadId; // Id du thread crée
};
```

REMARQUE:

La plupart de ces paramètres peuvent être lus ou modifiés en cours d'exécution du processus, par des fonctions appelées GetCurrentDirectory, SetCurrentDirectory, SetEnvironment, GetEnvironment, GetEnvironmentString, etc....

V.1.4. TERMINER UN PROCESSUS-FONCTIONS ExitProcess ET TerminateProcess:

La fonction ExitProcess permet de terminer le processus qui l'appelle. Elle doit donc être placée dans le code objet du processus que l'on veut terminer:

```
VOID ExitProcess
(
    UINT fuExitCode // Code de sortie
);
```

La fonction TerminateProcess permet à un processus donné de provoquer la terminaison d'un autre processus:

```
VOID TerminateProcess
(
    HANDLE Hprocess    // handler sur le processus à terminer
    UINT  fuExitCode   // code de sortie
);
```

V.1.5.OBTENIR UN HANDLER SUR UN PROCESSUS-FONCTIONS GetCurrentHandle et GetModuleHandle:

La fonction GetCurrentProcess permet d'obtenir un handler sur le processus courant:

```
HANDLE GetCurrentProcess ( VOID );
```

La fonction OpenProcess permet d'obtenir un handler sur un processus dont on connaît le PID:

```
HANDLE WINAPI OpenProcess
(
    DWORD    dwAccessRights,    // Droits d'accès (Ex: PROCESS_ALL_ACCESS)
    BOOL     bHandleInheritance, // (TRUE/FALSE) Capacité pour les fils d'hériter du
                                // handler du père
    DWORD    dwProcessId       // PID du processus dont on veut obtenir un handler
);
```

La fonction renvoie un handler sur le processus dont le fichier .exe est passé en paramètre (ce qui permet de le manipuler à partir d'un autre processus). Le nom du processus (lpzModule) doit correspondre à la valeur de l'argument lpzImageName de la fonction de création de ce processus.

REMARQUE:

Ne pas oublier de clore le handler en fin d'utilisation (CloseHandle (Handle hProcess) ;).

V.1.6.OBTENIR LE PID D'UN PROCESSUS-FONCTIONS GetCurrentProcessId et GetProcessId:

La fonction **GetCurrentProcessId** permet d'obtenir le PID du processus courant:

```
DWORD WINAPI GetCurrentProcessId (void);
```

Elle retourne le PID du processus.

La fonction **GetProcessId** permet d'obtenir le PID d'un processus sur lequel on dispose d'un handler:

```
DWORD WINAPI GetProcessId ( HANDLE ProcessHandle );
```

La fonction retourne le PID du processus ou bien 0 en cas d'échec (La fonction GetLastError permet d'obtenir plus de renseignements sur les causes de l'échec).

V.1.7.MODIFIER LA CLASSE DE PRIORITÉ D'UN PROCESSUS-FONCTION SetPriorityClass:

```
BOOL SetPriorityClass
(
    HANDLE hProcess    // Handler sur le processus
    , DWORD fdwPriority // IDLE, NORMAL, HIGHT ou REALTIME_PRIORITY_CLASS
);
```

V.2.LES OBJETS THREADS:

V.2.1.DECLARATION D'UN THREAD:

Un thread fait référence à une section de code exécutable (fonction thread) de la forme suivante:

```
DWORD WINAPI <Nom du thread> ( LPVOID lpvThreadParam )
{
    // -----
    // --- Corps du thread ---
    // -----
}
```

Cette section doit être placée dans le code objet du processus **avant** la directive WinMain. C'est à cette section qu'est passé le contrôle au lancement du thread. Un thread, lorsqu'il existe, peut se trouver dans 3 états:

- **Eligible:** il est prêt à être lancé, mais un thread de priorité plus élevé est actif.
- **Actif:** il est en cours d'exécution
- **En attente:** Il attend l'arrivée d'un d'un événement, la libération d'une ressource, la fin d'une entrée-sortie, etc.

REMARQUES:

- Une fonction thread peut être utilisée pour créer plusieurs threads (on pourra alors utiliser le paramètre passé pour personnaliser les traitements).
- Un Thread possède un identificateur de thread, qui est un entier positif.

V.2.2.CRÉER D'UN THREAD-FONCTION CreateThread:

```
HANDLE CreateThread
(
    LPSECURITY_ATTRIBUTES    lpsa
    , DWORD                  cbStack
    , LPTHREAD_START_ROUTINE lpStartAddr    // Nom de la fonction thread à exécuter
    , LPVOID                  lpvThreadParm // paramètre
    , DWORD                   fdwCreate
    , LPDWORD                  lpThreadId    // Adresse d'un DWORD permettant de
                                           // stocker l'ID du thread
);
```

V.2.3.TERMINER UN THREAD-FONCTIONS ExitThread ET TerminateThread:

La fonction ExitThread permet de terminer le thread qui l'appelle:

```
VOID ExitThread
(
    UINT fuExitCode    // Code retour
);
```

La fonction TerminateThread permet déclencher la terminaison un thread depuis un autre thread:

```
VOID TerminateThread
(
    HANDLE    hThread    // Handler du thread à terminer
    , UINT    fuExitCode // Code retour
);
```

V.2.4.CRÉER UN HANDLER SUR LE THREAD EN COURS:

```
HANDLE GetCurrentThread ( VOID );
```

V.2.5. MODIFIER LA PRIORITE D'UN THREAD-FONCTION SetThreadPriority:

```
BOOL SetThreadPriority
(
    HANDLE          hThread          // Handler sur le thread
,   int            nPriority         // Valeur de la priorité relative
);
```

Cette fonction modifie la priorité relative du thread par rapport à son processus. Les valeurs possibles du paramètre nPriority sont:

- `THREAD_PRIORITY_LOWEST`: abaisse de 2 la priorité
- `THREAD_PRIORITY_BELOW_NORMAL`: abaisse de 1 la priorité
- `THREAD_PRIORITY_NORMAL`: fixe la priorité à celle du processus
- `THREAD_PRIORITY_ABOVE_NORMAL`: augmente de 1 la priorité
- `THREAD_PRIORITY_HIGHEST`: augmente de 2 la priorité
- `THREAD_PRIORITY_IDLE`: Priorité fixée à 1, sauf pour la classe de processus `REALTIME`, pour laquelle elle est fixée à 16.
- `THREAD_PRIORITY_TIME_CRITICAL`: Priorité fixée à 15, sauf pour la classe `REAL_TIME`, pour laquelle elle est fixée à 31.

V.2.6. OBTENIR LE PID D'UN PROCESSUS-FONCTIONS GetCurrentProcesId et GetProcessId:

La fonction `GetCurrentThreadId` permet d'obtenir l'ID du thread courant:

```
DWORD WINAPI GetCurrentThreadId(void);
```

Elle retourne l'ID du thread.

La fonction `GetThreadId` permet d'obtenir l'ID d'un thread sur lequel on dispose d'un handler:

```
DWORD WINAPI GetThreadId ( HANDLE threadHandle );
```

La fonction retourne l'ID du thread ou bien 0 en cas d'échec (La fonction `GetLastError` permet d'obtenir plus de renseignements sur les causes de l'échec).

V.3.SYNCHRONISATION DES THREADS:

V.3.1.GENERALITES:

Un thread peut être mis en attente passive d'un évènements de différentes manières, grâce à l'utilisation d'objets kernel suivants:

- Les processus
- Les threads
- Les Mutex
- Les Sémaphores
- Les Evénements.
- Les Sections critiques
- Les Fichiers
- Les Entrées de consoles
- Les Notifications de modification de fichiers

En effet, la particularité de ces objets est de posséder deux états: SIGNALE / NON_SIGNALE:

- L'état NON_SIGNALE signifie que l'objet est en cours d'utilisation (par le thread d'un autre processus, par exemple)
- L'état SIGNALE qu'il est libre.

Lorsqu'un thread invoque un objet à l'état NON_SIGNALE (par l'intermédiaire un handler de cet objet), le scheduler le place en attente passive (attente du retour à l'état SIGNALE). Par exemple, un objet PROCESSUS est NON_SIGNALE pendant son exécution. Sa fin le remet à l'état SIGNALE.

V.3.2.FONCTIONS DE SYNCHRONISATION-WaitForSingleObject ET WaitForMultipleObjects:

V.3.2.1.WaitForSingleObject:

La fonction WaitForSingleObjet, dont le prototype est:

```
DWORD WaitForSingleObject  
(  
    HANDLE      hObject          // Handler sur l'objet synchronisateur  
    , int       iCondition       // Condition d'attente (INFINITE, TIME_OUT)  
) ;
```

Place le thread appelant en attente passive si l'objet de handler hObject est à l'état non signalé. Dès que l'objet passe à l'état signalé, le thread appelant reprend son exécution. Par exemple, dans une application Console, un thread peut se mettre en attente sur un objet «entrée de console» (créé par une fonction CreateFile) pour attendre une entrée de caractères. Cependant, seuls les objets Mutex, Sémaphores et Evénements sont vraiment adaptés à la synchronisation inter-processus.

Lorsque le paramètre iCondition a pour valeur INFINITE, l'attente se prolonge jusqu'à ce que l'objet pointé par le handler hObject soit signalé. Sinon, une valeur entière positive ou nulle correspond à la valeur du time-out en millisecondes.

L'argument de retour peut prendre les valeurs suivantes:

VALEUR	SIGNIFICATION
WAIT_OBJECT_0	L'objet est passé à l'état signalé.
WAIT_TIMEOUT	Le délais de time-out est écoulé et l'objet est toujours à l'état non signalé.
WAIT_FAILED	La fonction a échoué (CallLastError permet d'avoir plus d'informations sur la cause de l'erreur).
WAIT_ABANDONED	Cas d'un objet MUTEX non libéré par un thread avant la terminaison de celui-ci.

V.3.2.2.WaitForMultipleObjects:

La fonction WaitForMultipleObjects, dont le prototype est:

```
DWORD WaitForMultipleObjects  
(  
    DWORD    nCount,           // Nombre de handlers d'objets pointés par la table  
    HANDLE   *lpHandles,      // Tableau de handlers sur des objets de différents types,  
                                // mais pas sur le même objet  
    BOOL     bWaitAll,        // Si bWaitAll = TRUE, le thread appelant ne redémarre que  
                                // si tous les objets sont passés à l'état SIGNALE  
                                // Si bWaitAll = FALSE, le thread appelant redémarre dès  
                                // qu'un des objets passe à l'état signalé.  
    DWORD    dwMilliseconds    // Valeur du time-out en millisecondes: durée limite de  
                                // l'attente après laquelle le thread appelant est relancé,  
                                // quel que soit l'état des objets.  
);
```

L'argument de retour peut prendre les valeurs suivantes:

VALEUR	SIGNIFICATION
WAIT_OBJECT_0	Tous les objets sont passés à l'état signalé.
WAIT_TIMEOUT	Le délais de time-out est écoulé un objet est toujours à l'état non signalé.
WAIT_FAILED	La fonction a échoué (CallLastError permet d'avoir plus d'informations sur la cause de l'erreur).
WAIT_ABANDONED_0	Un objet MUTEX de la liste n'est pas libéré par un thread avant la terminaison de celui-ci.

REMARQUES:

- Si l'un des objets de la liste n'est pas à l'état ouvert pour l'appelant, le comportement de la fonction est imprévisible.
- Il faut que les handlers des objets possèdent les droits d'accès pour synchronisation (voir «standard accès rights», valeur SYNCHRONISE).
- Dans le cas où bWaitAll est à la valeur FALSE, la valeur de retour de la fonction indique quel est l'objet qui a changé d'état.

V.3.3.SYNCHRONISATION PAR LES SECTIONS CRITIQUES:

DEFINITION:

Une section critique est une structure de donnée permettant de gérer un mécanisme d'exclusion mutuelle de threads appartenant au même processus.

DECLARATION:

La déclaration d'une section critique s'effectue en donnée globale du processus (avant WinMain ou main). Le schéma de programme est le suivant:

```
// Données globales du processus  
.....  
CRITICAL_SECTION g_CriticalSection; // Déclaration d'une section critique  
.....  
  
int WINAPI WinMain ( . . . )  
{  
    // Données locales  
    .....  
    // Début du processus  
    .....  
    InitializeCriticalSection ( &g_CriticalSection); // Initialisation de la section  
                                                    // critique
```

```
.....  
}
```

UTILISATION:

Dans un thread donné, une section critique s'utilise comme suit:

```
// Début du code objetde chaque thread concurrent pour l'accès:  
DWORD WINAPI ThreadName ( LPVOID lpvThreadParam )  
{  
    // Données locales  
    .....  
    // Début du thread  
    .....  
    EnterCriticalSection ( &g_CriticalSection);  
    ..... //  
    ..... // Section critique  
    ..... //  
    LeaveCriticalSection ( &g_CriticalSection);  
    Return (state );  
}
```

Le code objet placé entre EnterCriticalSection et LeaveCriticalSection constitue un objet dont on peut synchroniser l'accès:

- Lorsqu'un thread est en train d'exécuter une instruction appartenant à la section critique, cet objet est à l'état NON SIGNALE. De ce fait, tout autre thread voulant pénétrer dans la section critique sera mis en attente jusqu'à la libération de la section par le premier thread.
- Lorsqu'aucun thread n'exécute la section critique, celle-ci est à l'état SIGNALE

Le mécanisme correspond donc à une exclusion mutuelle.

V.3.4.SYNCHRONISATION PAR LES OBJETS MUTEX:

DECLARATION:

Le processus qui crée le MUTEX utilise la fonction createMutex.

```
HANDLE CreateMutex  
(  
    LPSECURITY_ATTRIBUTE lpsa,          // Attributs de sécurité  
    , BOOL               fInitialOwner, // Nom du MUTEX, valable pour tous les  
                                     // processus.  
    , LPTSTR             lpsaMutexName, // Nom d'objet  
) ;
```

OUVERTURE:

Tout autre processus qui utilise le MUTEX obtient un handler par:

```
HANDLE OpenMutex  
(  
    , DWORD   fdwAcces  
    , BOOL    fInherit  
    , LPTSTR  lpszName          // Nom d'objet  
) ;
```

CLOTURE ET DESTRUCTION:

Un processus clot un MUTEX par la fonction:

```
HANDLE ReleaseMutex  
(  
    HANDLE hMutex  
) ;
```

Si ce processus est le créateur du MUTEX, celui-ci est détruit.

SCHEMA D'UTILISATION:

La déclaration d'un MUTEX s'effectue en donnée globale du processus (avant WinMain ou main):

```
// Début du code objetdu main:  
// Données globales du processus  
.....  
HANDLE g_hMutex;  
.....  
  
int WINAPI WinMain ( . . . )  
{  
    // Données locales  
    .....  
    // Début du processus  
    .....  
    g_hMutex = CreateMutex( .., .., «NomMutex» );  
    .....  
    // Création éventuelle d'un thread utilisateur dans le processus  
    .....  
    CloseHandle ( g_hMutex );  
}
```

Dans un thread donné, le MUTEX s'utilise comme suit:

```
// Début du code objetde chaque thread concurrent pour l'accès:  
  
DWORD WINAPI ThreadName ( LPVOID lpvThreadParam )  
{  
    // Données locales  
    .....  
    DWORD dw;  
    .....  
    // Début du processus  
    OpenMutex ( ** ,** , ** , <nom mutex> );  
    .....  
    Dw = WaitForSingleObject ( g_hMutex, INFINITE );  
    .....  
    // Section de code protégée par le MUTEX  
    .....  
    ReleaseMutex ( g_hMutex); // Repasse le Mutex à l'état SIGNALE  
    Return (state );  
}
```

V.3.5.SYNCHRONISATION PAR LES OBJETS SEMAPHORES:

GENERALITES:

Un objet sémaphore est un objet KERNEL qui peut être considéré comme une structure de données comprenant deux variables entières positives ou nulles: un décompteur et un autre entier contenant la valeur maximale de ce décompteur.

Lorsque la valeur du décompteur est plus grande que 0, le sémaphore est à l'état SIGNALE. Lorsque cette valeur est nulle, le sémaphore est à l'état NON_SIGNALE.

L'invocation d'un sémaphore par la fonction **WaitForSingleObjet** met l'appelant en attente si le sémaphore est non signalé. Dans le cas contraire, le thread continue son exécution et le décompteur est décrémenté de 1. Le sémaphore reste donc à l'état signalé tant que le décompteur n'est pas mis à 0.

L'invocation du sémaphore par la fonction **ReleaseSemaphore** incrémente d'une valeur entière le décompteur, à condition que la valeur résultante de celui-ci ne dépasse par la valeur maximale spécifiée.

Les SEMAPHORES permettent donc de gérer des accès multiples à une ressource, qui peut ainsi accepter à un instant donné l'accès de plusieurs threads (autant d'accès qu'il est spécifié dans la variable valeur maximale). Ce n'est que si le nombre maximal d'accès simultanés est atteint que le processus requérant le sémaphore est mis en attente. Ce fonctionnement est donc analogue à celui des sémaphores UNIX.

DECLARATION:

La fonction CreateSemaphore permet de créer un objet kernel SEMAPHORE:

```
HANDLE CreateSemaphore
(
    LPSECURITY_ATTRIBUTES lpsa // Pointeur sur une structure contenant les
                                // attributs de sécurité choisis ou NULL pour
                                // choisir les attributs par défaut.
    , LONG                cSemInitial // Valeur initiale du compteur d'accès
                                // ( 0 < cSemInitial <= cSemMax)
    , LONG                cSemMax // Nombre d'accès maximal
    , LPTSTR              lpszSemName // Nom d'objet (valable pour tous processus)
);
```

Lors de la création du sémaphore, l'argument cSemInitial est chargé dans le décompteur. Ceci limite (uniquement en début d'utilisation) le nombre d'accès possibles.

OUVERTURE:

Pour obtenir un handle sur un objet SEMAPHORE déjà créé, un thread utilise la fonction:

```
HANDLE OpenSemaphore
(
    DWORD fdwAcces // Nom d'objet
    , BOOL fInherit
    , LPTSTR lpszName // Nom d'objet
);
```

CLOTURE D'UN SEMAPHORE:

C'est la fonction CloseHandle sur le handler d'un sémaphore qui permet de clore l'accès à un sémaphore donné. Si cette fonction est invoquée par le thread créateur du sémaphore, celui-ci est détruit.

REQUERIR UN SEMAPHORE:

Par la fonction WaitForSingleObject appliquée au handler du sémaphore.

LIBERER UN SEMAPHORE:

La fonction ReleaseSemaphore permet d'incrémenter le décompteur d'une certaine valeur. Cette fonction, appliquée à un sémaphore NON_SIGNALE le repasse à l'état signalé, à condition que cette valeur n'amène pas le décompteur à dépasser sa valeur maximale. Sinon, l'opération est annulée et la fonction renvoie FALSE.

```
BOOL ReleaseSemaphore
(
    HANDLE hSemaphore, // Handler du sémaphore
    LONG lReleaseCount, // Valeur à ajouter au décompteur
    LPLONG lpPreviousCount // Valeur du décompteur avant l'ajout
);
```

SCHEMA D'UTILISATION:

La déclaration d'un SEMAPHORE s'effectue avant WinMain ou main:

```
// Début du code objet du main:
// Données globales du processus
. . . . .
HANDLE hSem1; // Déclaration d'un handler sur un sémaphore
```

La création du sémaphore peut être effectuée dans le thread principal du processus ou dans n'importe quel autre thread de ce processus (il est cependant recommandé de créer le sémaphore au début du thread principal):

```
// Début du code objet du processus (thread principal)
int WINAPI WinMain ( . . . )
{
    // Données locales
    . . . . .
    // Début du processus
    . . . . .
    hSem1 = CreateSemaphore( «acces», .., «NomSemaphore» );
    // Création éventuelle d'un thread utilisateur dans le processus
    . . . . .
    CloseHandle ( g_hSemaphore );
}
```

Dans un thread donné, le sémaphore s'utilise comme suit:

```
// Début du code objet de chaque thread concurrent pour l'accès:
DWORD WINAPI ThreadName ( LPVOID lpvThreadParam )
{
    // Données locales
    . . . . .
    DWORD dw;
    // Début du thread
    . . . . .
    g_hSemaphore = OpenSemaphore ( **, **, «Nom du sémaphore» );
    . . . . .
    Dw = WaitForSingleObject ( g_hSemaphore, INFINITE );
    . . . . . //
    . . . . . // Zone dont l'accès et protégé par le sémaphore
    . . . . . //
    ReleaseSemaphore ( g_hSemaphore); // Repasse le sémaphore à l'état SIGNALE
    Return (state );
}
```

V.3.6.SYNCHRONISATION PAR LES OBJETS EVENEMENTS:

GENERALITES:

Contrairement aux objets MUTEX et SEMAPHORE, les mécanismes d'ÉVÉNEMENTS ne sont pas employés pour contrôler l'accès à des données, mais pour permettre à un thread donné de signaler la survenue d'un événement à d'autres threads (appartenant ou non au même processus). Il existe des événements de réinitialisation «manuelle» qui permettent de signaler un événement à plusieurs threads et des événements de réinitialisation «automatique» qui permettent de signaler un événement à un thread particulier.

CREATION D'UN OBJET EVENEMENT:

la fonction CreateEvent permet à un processus de créer un ÉVÉNEMENT:

```
HANDLE CreateEvent
(
    LPSECURITY_ATTRIBUTE lpsa
    , BOOL fManualReset // TRUE => Etat manipulé par les fonctions
                        // SetEvent et ResetEvent
                        // FALSE => Etat géré automatiquement
    , BOOL fInitialState // TRUE si signalé, sinon FALSE
    , LPTSTR lpszEventName // Nom de l'événement, valable pour tous les
                          // processus
);
```

OUVERTURE D'UN OBJET EVENEMENT:

```
HANDLE OpenEvent
(
    DWORD fdwAcces //
    , BOOL fInherit //
```

```
, LPTSTR lpszName // Nom d'objet  
);
```

MANIPULATION DE L'ETAT D'UN EVENEMENT:

Lorsqu'un événement est déclaré "Manuel" (fManualReset = true), son état SIGNALE/NON SIGNALE est manipulé à l'aide de deux fonctions:

```
BOOL SetEvent ( Handle hEvent ); // Passe l'événement à l'état signalé  
BOOL ResetEvent ( Handle hEvent ) // le ramène à l'état non signalé.
```

Les fonctions WaitForSingleObject et WaitForMultipleObjects effectuent la remise à l'état non signalé des événements automatiques, mais pas des manuels, pour lesquels il faut utiliser ResetEvent.

CLOTURE D'UN EVENEMENT:

C'est la fonction CloseHandle sur le handler d'un événement qui permet de clore l'accès à un événement donné. Si cette fonction est invoquée par le thread créateur, l'événement est détruit.

SCHEMA D'UTILISATION:

La déclaration d'un EVENEMENT s'effectue en donnée globale du processus (avant WinMain ou main):

```
// Début du code objet du main:  
// Données globales du processus  
.....  
HANDLE g_hEvent;  
.....  
  
int WINAPI WinMain ( . . . )  
{  
// Données locales  
.....  
// Début du processus  
.....  
g_hEvent = CreateEvent( .., .., «NomEvent» );  
.....  
// Création éventuelle d'un thread utilisateur dans le processus  
.....  
CloseHandle ( g_hEvent );  
}
```

Dans un thread donné, la synchronisation sur événement s'utilise comme suit:

```
//Début du code objet de chaque thread concurrent pour l'accès:  
DWORD WINAPI ThreadName ( LPVOID lpvThreadParam )  
{  
// Données locales  
.....  
DWORD dw;  
.....  
// Début du processus  
.....  
Dw = WaitForSingleObject ( g_hEvent, INFINITE );  
..... //  
..... // Zone dont l'accès est synchronisé  
..... //  
ResetEvent ( g_hEvent); // Repasse l'événement à l'état NON-SIGNALE (s'il  
// s'agit d'un événement manuel  
Return (state );  
}
```

V.4.PARTAGE DE DONNEES ENTRE PROCESSUS:

V.4.1.GENERALITES:

Nous avons vu précédemment que chaque processus WINDOWS s'exécute dans un espace mémoire qui lui est réservé. L'accès d'autres processus à cet espace est interdit par le système et provoque une alarme. Il est pourtant indispensable, dans une application comprenant plusieurs processeurs, de pouvoir échanger des données entre ceux-ci. De ce fait, les systèmes d'exploitation mettent à disposition des utilisateurs différents mécanismes permettant à des processus situés sur une même machine ou sur des machines différentes d'échanger des données.

V.4.2.TRANSMISSION DE DONNEES PAR LE MECANISME DES SOCKETS:

Le mécanisme des sockets, permet d'échanger des messages entre processus d'une même machine ou de machines différentes, en utilisant les couches réseau. Ces messages peuvent évidemment être utilisés pour échanger des données.

Les avantages de cette solution sont:

- Sa portabilité: les socket sont supportés par la plupart des systèmes d'exploitation.
- La possibilité de traiter le cas de processus distants sur le réseau (systèmes répartis) de la même manière que le cas de processus locaux.

Les inconvénients sont:

- La durée des échanges, dûe a la traversée des couches réseau du système d'exploitation. Cette durée peut aller de la milliseconde à quelques dizaines de millisecondes quand le volume de données est important.
- Le manque de déterminisme de cette durée (surtout si l'on travaille en mode connecté)
- Une mise en oeuvre assez lourde, même dans le cas d'échanges de petits volumes de données.

V.4.3.PARTAGE DE DONNEES PAR LES FICHIERS:

Plusieurs processus peuvent avoir accès en lecture et écriture au contenu d'un même fichier: il suffit que chacun d'eux ouvre un accès sur ce fichier (fonction open). Un fichier peut donc être utilisé pour échanger des données au moins entre des processus s'exécutant sur une même machine.

Si l'on dispose de mécanismes de partage de fichiers entre machines (par exemple, le Network File System des O.S. UNIX-Linux), il est également possible d'échanger des données entre processus distants avec ce procédé.

L'avantages de cette solution est sa portabilité (elle n'exige l'ajout d'aucun mécanisme spécifique).

Les inconvénients sont:

- La durée de l'opération (l'accès aux données d'un fichier nécessite des délais se chiffrant en dizaines de millisecondes)
- La lourde de la mise en oeuvre, même dans le cas d'échanges de petits volumes de données.

V.4.4.PARTAGE DE DONNEES PAR DES ZONES MEMOIRES PARTAGEES:

V.4.4.1.MECANISME DE PROTECTION MEMOIRE INTER-PROCESSUS:

En général, les codes objets des processeurs informatiques possèdent des instructions capables d'accéder à la totalité de la mémoire des machines, par l'intermédiaire de **pointeurs**. De ce fait, en lui-même, l'exécutable d'un processus quelconque est capable d'accéder à n'importe quelle zone de la mémoire, y compris l'espace alloué à un autre processus.

Cependant, les processeurs intègrent également la notion de **mode d'exécution**: il existe au moins deux modes d'exécution: le mode **système** et le mode **utilisateur**. Un code exécutable s'exécute d'après l'un ou l'autre de ces modes.

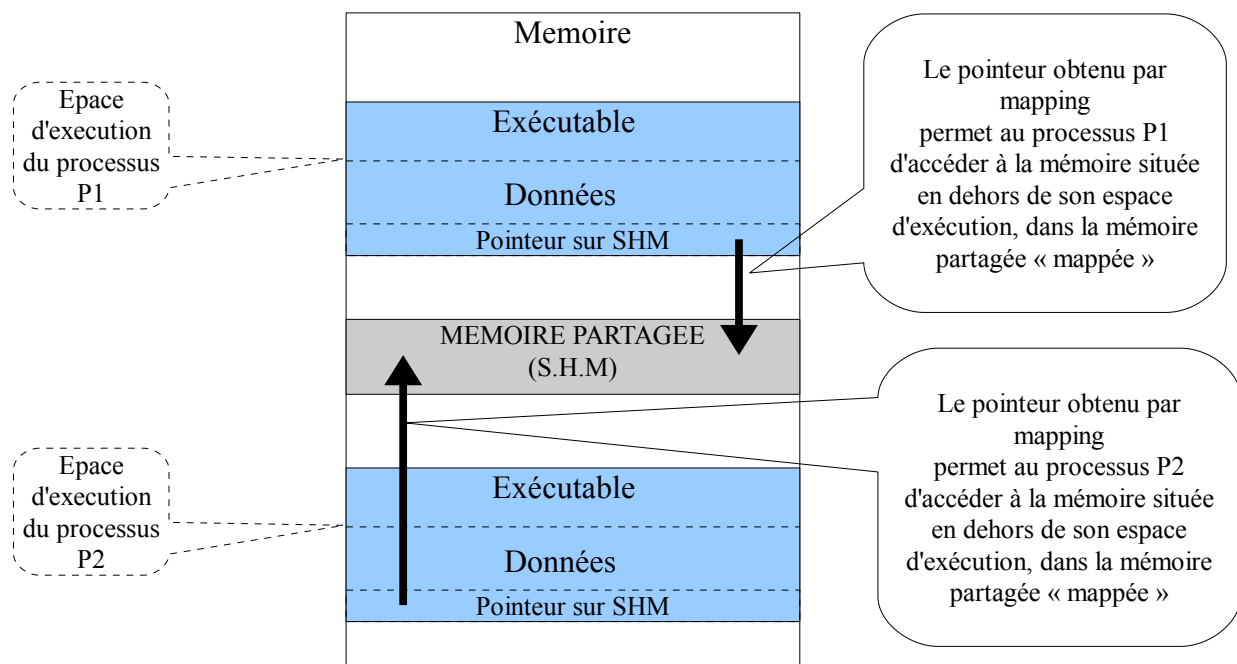
Le **mode système** est celui dans lequel s'exécute le système d'exploitation. En mode système, l'accès du code exécutable n'est pas limité. En revanche, en **mode utilisateur**, l'accès d'un code exécutable donné est contrôlé par un mécanisme matériel de **protection mémoire**. C'est ce mécanisme qui limite l'accès de l'exécutable d'un processus à l'espace d'exécution de ce processus.

En fait, à la création d'un processus utilisateur, le système alloue à celui-ci une zone mémoire, matérialisée par une adresse de début et une adresse de fin. Lorsqu'un le processus s'exécute, ces deux adresses sont chargées dans les registres du mécanisme de protection mémoire. A chaque fois que le code exécutable du processus demande un accès à la mémoire, le mécanisme de protection mémoire compare l'adresse demandée au contenu de ces registres. Si cette adresse est en dehors des limites, l'accès échoue et une alarme système est émise.

V.4.4.2.MECANISME DE PARTAGE MEMOIRE INTER-PROCESSUS:

Appelé **SHARED MEMORY** en anglais (souvent abrégé en SHM dans les documentations), ce procédé consiste à allouer, en dehors des espaces réservés de chaque processus, des zones mémoires dont l'accès est géré par le système d'exploitation. Celui-ci peut accorder à des processus utilisateurs l'accès direct au contenu de la mémoire contenue dans cette zone en leur fournissant des **POINTEURS** d'accès capables d'outrepasser la protection mémoire. Cette opération est appelée **MAPPING** en anglais.

*N.D.A: A ma connaissance, ce terme n'a pas d'équivalent satisfaisant en français. La traduction littérale (cartographie) n'est pas vraiment adaptée. Le terme mapping est souvent employé en informatique pour désigner le **rattachement** (d'une entité matérielle ou logicielle) à une **adresse mémoire** (par exemple, les ports de communication avec le matériel sont «mappés» en mémoire, ce qui veut dire qu'on y accède par une adresse mémoire, **comme s'il s'agissait d'octets situés à cette adresse**).*



Les avantages du procédé sont:

- La rapidité: l'accès se fait grâce à un simple adressage indirect utilisant le pointeur fourni. La durée d'un accès est donc la même que dans le cas où le processus accède à son propre espace d'exécution (quelques nano-secondes).
- La simplicité de mise en œuvre.

Les inconvénients du procédé sont:

- Sa limitation à des processus locaux.
- Sa portabilité relativement faible, chaque système d'exploitation implémentant ses propres mécanismes, assez différents les uns des autres.

V.4.4.3.LES OBJETS FILE MAPPING DE WINDOWS:**REMARQUE PRELIMINAIRE:**

WIN32 implémente les mémoires partagées à l'aide d'objets KERNEL appelés FILE MAPPING. En fait, ces objets permettent de partager des données par l'intermédiaire de fichiers ou de zones situées dans la mémoire paginée (paging file). On ne peut donc parler de SHARED MEMORY que dans le dernier cas, et encore, dans la limite ou la zone partagée se trouve effectivement en mémoire.

CREATION D'UN OBJET FILE MAPPING:

La fonction CreateFileMapping permet **soit de créer** un objet File Mapping non existant, **soit d'ouvrir** un File Mapping existant.

```
HANDLE WINAPI CreateFileMapping
(
    HANDLE                hFile,
    LPSECURITY_ATTRIBUTES lpAttributes,
    DWORD                 flProtect,
    DWORD                 dwMaximumSizeHigh,
    DWORD                 dwMaximumSizeLow,
    LPCTSTR               lpName
);
```

hFile	- Si hfile = INVALID_HANDLE_VALUE, la zone partagée est prise dans la mémoire paginée. - Si hFile est un handler sur un fichier, c'est le contenu de ce fichier qui sera partagé.
lpAttributes	Pointe sur une structure SECURITY_ATTRIBUTES qui définit les modalités d'héritage du handler retourné. Sinon, positionner la valeur à NULL
FlProtect	Définit les permissions demandées par l'appelant pour l'accès à la zone partagée (PAGE_EXECUTE-READ, PAGE_EXECUTE_READWRITE, PAGE_READWRITE, etc...)
dwMaximumSizeHigh	Rang du premier entier long par rapport au début de la zone partagée (ou 0 si hfile est un handler de fichier)
dwMaximumSizeLow	Rang du dernier entier long par rapport au début de la zone partagée (ou 0 si hfile est un handler de fichier). Si le précédent argument vaut 0 et hFile vaut INVALID_HANDLE_VALUE, la valeur est égale à la taille de la zone en octets.
lpName	Pointeur sur le nom alloué à l'objet (Si NULL, l'objet est dit «unnamed»)

- Si l'objet est créé, la fonction retourne un handler sur cet objet.
- Si l'objet existait déjà, un handler sur cet objet est retourné. La fonction GetLastError, appelée à la suite, retourne alors ERROR_ALREADY_EXISTS.
- Si la création ou l'ouverture ont échoués, la fonction retourne NULL.

OBTENTION D'UN POINTEUR SUR LA ZONE PARTAGEE-OUVERTURE D'UNE VUE:

Lorsqu'un processus a créé un objet FILE MAPPING ou bien a ouvert un objet existant, il peut obtenir un pointeur sur la zone partageable grâce à la fonction MapViewOfFile. Cette fonction permet d'obtenir une «vue» sur le fichier ou la zone partagée:

```
LPVOID WINAPI MapViewOfFile
(
    HANDLE                hFileMappingObject,
    DWORD                 dwDesiredAccess,
    DWORD                 dwFileOffsetHigh,
    DWORD                 dwFileOffsetLow,
    SIZE_T                dwNumberOfBytesToMap
);
```

);

hFileMappingObject	Handler sur l'objet File Mapping
dwDesiredAccess	Permet de spécifier le type d'accès qu'on veut autoriser au processus (FILE_MAP_ALL_ACCESS, FILE_MAP_READ, FILE_MAP_WRITE, FILE_MAP_COPY, FILE_MAP_EXECUTE)
dwFileOffsetHigh	Début de la vue par rapport au début de la zone partagée
dwFileOffsetLow	Fin de la vue par rapport au début de la zone partagée
dwNumberOfBytesToMap	Nombre de bytes à mapper, ou 0 si l'on mappe toute la zone partageable.

- Si la fonction réussit, elle retourne un **pointeur sur l'adresse de début de la vue**. C'est ce pointeur qui permettra d'accéder à la zone partagée, dans les limites fixées par les arguments dwFileOffsetHigh et dwFileOffsetLow.
- Si elle échoue, elle retourne NULL.

FERMETURE D'UNE VUE:

La fonction UnmapViewOfFile permet de refermer une vue existante et de désallouer le pointeur obtenu:

```
BOOL WINAPI UnmapViewOfFile  
(  
    LPCVOID lpBaseAddress;  
);
```

L'argument lpBaseAddress est le pointeur retourné par l'appel à la fonction MapViewOfFile. La fonction retourne 0 en cas d'erreur.

EXEMPLE D'ACCES A LA ZONE PARTAGEE:

Grâce au pointeur retourné par la fonction MapViewOfFile, l'appelant va pouvoir accéder aux données de la zone partage grâce à des adressages indirects:

Exemple: Ecriture de 100 octets dans la zone partagée

```
char MonBuffer[100];  
  
HandlerSHM = CreateFileMapping ( INVALID_HANDLE_VALUE, NULL,  
                                PAGE_READWRITE, 0, TAILLE_MEMOIRE, lpNomSHM );  
if ( HandlerSHM != NULL )  
{  
    lpBuf = MapViewOfFile ( HandlerSHM, FILE_MAP_ALL_ACCESS, 0, 0, TAILLE_MEMOIRE );  
    CopyMemory ( (void *) lpBuf, (void *) MonBuffer, 100 );  
    UnmapViewOfFile( (void *) lpBuf );  
    CloseHandle (HandlerSHM);  
}
```

VI.EXEMPLES DE CODES

VI.1.CREATION D'UN PROCESSUS:

Le code objet suivant crée un processus fils en utilisant la fonction «CreateProcess» et lance en thread principal le logiciel «calculatrice» de Windows (la calculatrice s'affiche donc sur l'écran). Puis, il attend la fin de ce processus fils, grâce à la fonction WaitForSingleObjet. En attendant cet événement, il fait progresser et affiche la valeur d'un compteur, au rythme d'une unité par seconde. Lorsque l'on arrête l'application calculatrice (par un clic sur la croix en haut et à gauche de sa fenêtre), l'objet kernel correspondant au processus fils passe à l'état signalé et le compte-rendu de la fonction WaitForSingleObjet passe à la valeur WAIT_OBJECT_0, ce qui termine la boucle d'attente. Le logiciel père affiche alors "Terminaison du processus fils detectee".

```
#include <windows.h>
#include <stdio.h>

int WINAPI WinMain ( HINSTANCE hThisInstance, HINSTANCE hPrevInstance, LPSTR lpszArgument,
                    int nCmdShow)
{
// DONNEES LOCALES
    char          Buffer[100];
    BOOL          CR;
    DWORD        Ct, Cr;
    STARTUPINFO  StartInfo;
    PROCESS_INFORMATION ProcessInfo;

// DEBUT
    // Initialisation minimale de la structure STARTUPINFO définissant la
    // fenêtre principale du processus à créer
    ZeroMemory( &StartInfo, sizeof(StartInfo) );
    StartInfo.cb = sizeof(StartInfo);

    // Initialisation de la structure PROCESS_INFORMATION retournée
    // par l'appel à CreateProcess
    ZeroMemory( &ProcessInfo, sizeof(ProcessInfo) );

    // Creation du processus fils.
    CR = CreateProcess
        (
            NULL,          // Nom du fichier executable.
            "c:\\windows\\System32\\calc.exe", // Ligne de commande lançant l'executable
                                //calculatrice
            NULL,          // Attribut de sécurité par défaut pour le processus
            NULL,          // Attribut de sécurité par défaut pour le thread
            FALSE,        // Pas d'héritage des handles par le processus fils
            0,             // Par défaut, la classe de priorité est NORMAL_PRIORITY_CLASS
            NULL,          // Par défaut, le nouveau processus utilise l'environnement du parent
            NULL,          // Par défaut, le nouveau processus utilise le même lecteur et le même
                                // repertoire que le parent.
            &StartInfo,    // Pointeur sur structure _STARTUPINFO définissant les
                                // caractéristiques de la fenêtre d'affichage principale du nouveau
                                // processus (si celui-ci en définit une)
            &ProcessInfo  // Pointeur vers une structure _PROCESS_INFORMATION contenant des
                                // informations relatives au processus (disponibles en retour de la
                                // fonction CreateProcess ). les membres hProcess et hThread de cette
                                // structure contiennent des handles sur le process et le thread
                                // principal
        );

    // SI ( La creation a echoué ) ALORS
    if ( !CR )
    {
        // Afficher l'erreur et terminer le processus
        printf( "Erreur CreateProcess (%ld).\n", GetLastError() );
        return (-1);
    }
    // SINON
    else
    {
```

```
// Signaler le lancement du processus fils
printf( "Creation et lancement processus fils reussis\n" );
// FINSI
}

// TANT QUE ( L'objet processus fils n'est pas signalé ) FAIRE
Ct = 0;
Cr = WAIT_TIMEOUT;
while ( Cr == WAIT_TIMEOUT )
{
    // Imprimer un message d'attente de fin du processus fils
    printf ( "Attente fin du processus fils: %ld s\n", Ct );

    // Incrementer le compteur d'attente
    Ct++;

    // Attendre que le processus fils soit signalé ou qu'une erreur survienne
    Cr = WaitForSingleObject( ProcessInfo.hProcess, 1000 );
// FINFAIRE
}

// SI ( On est sortie de l'attente sur objet signalé ) ALORS
if ( Cr == WAIT_OBJECT_0 )
{
    // Le signaler
    printf( "Terminaison du processus fils detectee.\n" );
}
// SINON
else
{
    // Signaler une erreur
    printf ( "Echec de la fonction d'attente ( compte-rendu = %d )\n", GetLastError() );
// FINSI
}

// Attente de la commande de terminaison du logiciel père
printf( "Pour quitter, frapper une touche...\n" );
gets ( Buffer );

// Clôture des handlers du fils.
CloseHandle( ProcessInfo.hProcess );
CloseHandle( ProcessInfo.hThread );
// FIN
return 0;
}
```

VI.2.TERMINAISON D'UN PROCESSUS:

Le code objet suivant permet alternativement de créer un processus, puis de le détruire en utilisant son PID.

```
#include <windows.h>
#include <stdio.h>

int WINAPI WinMain ( HINSTANCE hThisInstance, HINSTANCE hPrevInstance, LPSTR lpszArgument,
                    int nCmdShow)
{
    char          Buffer[100];
    BOOL          CR;
    int           PID;
    HANDLE        ProcessHandle;
    DWORD         ExitCode;
    STARTUPINFO  StartInfo;
    PROCESS_INFORMATION ProcessInfo;

    PID = -1;
    // TANT QUE ( le programme n'est pas stoppé ) FAIRE
    while ( 1==1 )
    {
        printf ( "Creer un processus? -->Taper C\n
                Terminer un processus? -->Taper T\n
                Quitter l'application? -->Taper Q\n" );
        gets ( Buffer );

        // SI ( on veut creer un processus ) ALORS
        if ( Buffer[0] == 'C' )
        {
            // SI ( Un processus est déjà crée ) ALORS
            if ( PID != -1 )
```

```
{
    printf ( "Detruire d'abord le processus cree.\n" );
}
// SINON
else
{
    // Initialisation minimale de la structure STARTUPINFO définissant la
    // fenêtre principale du processus à créer
    ZeroMemory( &StartInfo, sizeof(StartInfo) );
    StartInfo.cb = sizeof(StartInfo);

    // Initialisation de la structure PROCESS_INFORMATION retournée
    // par l'appel à CreateProcess
    ZeroMemory( &ProcessInfo, sizeof(ProcessInfo) );

    // Creation du processus fils.
    CR = CreateProcess ( NULL, "c:\\windows\\System32\\calc.exe", NULL, NULL,
        FALSE, 0, NULL, NULL,&StartInfo, &ProcessInfo );
    // SI ( La creation a echoué ) ALORS
    if ( !CR )
    {
        // Afficher l'erreur et terminer le processus
        printf( "Erreur CreateProcess (%ld).\n", GetLastError() );
        return (-1);
    }
    // FINSI
}

// Recuperation du PID du processus créé dans la structure PROCESS_INFORMATION
PID = ProcessInfo.dwProcessId;

// FINSI
}
}
// SINON SI ( on veut terminer un processus ) ALORS
else if ( Buffer[0] == 'T' )
{
    // SI ( Aucun processus n'est crée ) ALORS
    if ( PID == -1 )
    {
        printf ( "Aucun processus n'est cree!\n" );
    }
    // SINON
    else
    {
        // Obtenir un handler sur le processus
        ProcessHandle = OpenProcess ( PROCESS_ALL_ACCESS, FALSE, PID );

        // SI ( le handler est obtenu ) ALORS
        if ( ProcessHandle )
        {
            // Terminer le processus
            TerminateProcess ( ProcessHandle, ExitCode );
            CR = false;
        }
        // SINON
        else
        {
            // Signaler l'erreur
            printf( "Le processus de PID %d n'existe pas.\n", PID );
        }
        // FINSI
    }
    PID = -1;

    // FINSI
}
}
// SINON
else if ( Buffer[0] == 'Q' )
{
    // Terminer le programme
    break;
    // FINSI
}
// FINFAIRE
}
// FIN
}
```

VI.3.GESTION DES THREADS:

Dans le code objet suivant, le thread principal du processus crée un événement appelé «Dizaine» qui va être associé au passage à la dizaine d'un compteur. Puis il crée un thread qui est associé à la fonction Compteur. Et se met en attente de l'événement «Dizaine».

La fonction du thread Compteur incrémente un compteur (CT) toutes les 500 ms. D'autre part, lorsque la valeur du compteur correspond à une dizaine ronde, le thread signale l'événement Dizaine.

Quand le thread principal détecte le passage de l'événement «Dizaine» à l'état signalé, il affiche la valeur courante du compteur, puis il teste la valeur de celui-ci. Si elle est égale à 50, il stoppe le thread et se termine lui-même:

```
#include <windows.h>
#include <stdio.h>

// DONNEES COMMUNES AUX DEUX THREADS
long CT;
HANDLE EventHandle;

// THREAD SECONDAIRE Compteur
DWORD WINAPI Compteur ( LPVOID lpvThreadParam )
{
// DONNEES LOCALES

// DEBUT
CT = 0;

// TANT QUE ( Le thread n'est pas stoppe par le parent ) FAIRE
while ( 1==1 )
{
// Afficher le compteur
printf ( "Thread Compteur: %ld\n", CT );

// Incrémenter le compteur
CT++;

// Si une dizaine est atteinte, signaler l'événement "Dizaine"
if ( CT%10 == 0 ) SetEvent ( EventHandle );

// Attendre 500 ms
Sleep (500);
// FINFAIRE
}
// FIN
}

// THREAD PRINCIPAL DU PROCESSUS
int WINAPI WinMain ( HINSTANCE hThisInstance, HINSTANCE hPrevInstance, LPSTR lpszArgument, int nCmdShow)
{
// DONNEES LOCALES
char Buffer[100];
bool CR;
HANDLE ThreadHandle;
DWORD ThreadPID;
int R;
DWORD Dw;

// DEBUT
// Creation d'un objet événement "Dizaine"
EventHandle = CreateEvent ( NULL, TRUE, FALSE, "Dizaine" );

// Creation du thread Compteur
ThreadHandle = CreateThread ( NULL, 0, Compteur, 0, 0, (DWORD*) &ThreadPID );

// TANT QUE (le processus n'est pas terminé ) FAIRE
while ( 1==1 )
{
// Attendre l'événement "Dizaine"
Dw = WaitForSingleObject ( EventHandle, INFINITE );

// Réinitialiser l'événement dizaine
ResetEvent (EventHandle);

// Imprimer la valeur du compteur sur réception de l'événement
printf ( "Thread principal: evenement \"Dizaine\" signale, valeur compteur = %ld\n", CT );
```

```
// Au bout de 25 secondes de comptage, terminer le thread Compteur
// et terminer la boucle d'attente
if ( CT >= 50 )
{
    TerminateThread ( ThreadHandle, R );
    printf ( "Terminaison du thread compteur\n" );
    CloseHandle (ThreadHandle);
    CloseHandle (EventHandle);
    Sleep ( 5000 );
    break;
}
// FINFAIRE
}
// FIN
}
```

VI.4.PARTAGE DE DONNEES ENTRE PROCESSUS (OBJET FILE_MAPPING):

```
#include <windows.h>
#include <stdio.h>
#include <conio.h>
#include <tchar.h>

#define TAILLE_MEMOIRE 1000

// Nom de l'objet FILE_MAPPING
char NomSHM[50] = "Global\\MaMemoirepartagee";

// PROCEDURE PRINCIPALE
int main()
{
    // DONNEES LOCALES
    HANDLE    HandlerSHM;
    char      *Buffer;
    char      Buff[100];

    // DEBUT
    // Tenter de créer la mémoire partagée (objet FILE_MAPPING) ou bien l'ouvrir s'il est déjà créé.
    HandlerSHM = CreateFileMapping
    (
        INVALID_HANDLE_VALUE, // Crée une mémoire partagée en mémoire (sinon, il faut
        // un handler sur le fichier partagé)
        NULL, // Pointeur sur une structure SECURITY_ATTRIBUTES
        PAGE_READWRITE, // File d'indicateurs définissant les permissions d'accès
        // (PAGE_READONLY, PAGE_EXECUTE_READ, etc...)
        0, // Taille maximale de l'objet (limite supérieure)
        TAILLE_MEMOIRE, // Taille maximale de l'objet (limite inférieure)
        NomSHM // Nom de l'objet mémoire partagée
    );

    // SI ( La création a échoué )
    if ( HandlerSHM == NULL)
    {
        // Le signaler
        printf ( "Echec de la creation de la mémoire partagee (ERREUR = %d).\n", GetLastError() );
        return -1;
    }
    // FINSI
}

// Tenter de créer une vue sur la mémoire partagée
Buffer = (char *) MapViewOfFile
(
    HandlerSHM, // handler sur la SHM
    FILE_MAP_ALL_ACCESS, // Ouverture en lecture/ecriture
    0, //
    0, // La vue comprend toute la zone partagée
    0 //
);

// SI ( la création de la vue a échoué )
if ( Buffer == NULL)
{
    // Le signaler et terminer le logiciel
    printf ( "impossible de créer une vue ( ERREUR = %d).\n", GetLastError());
    CloseHandle( HandlerSHM);
    return 1;
}
// FINSI
}
```

```
// TANT QUE ( On ne termine pas le programme ) FAIRE
while ( TRUE )
{
    // Saisir une commande (Lire, ecrire ou terminer)
    printf ( "E =>Ecrire un message, L => Lire un message Q => Terminer:\n" );
    gets ( Buff );

    // SI ( On veut terminer le logiciel ) ALORS interrompre la boucle
    if ( Buff[0] == 'Q' ) break;

    // SINON SI ( On veut écrire en zone partagée ) ALORS
    else if ( Buff[0] == 'E' )
    {
        // Saisir le message a ecrire
        ZeroMemory ( Buff, sizeof (Buff) );
        printf ( "Saisir le message:\n" );
        gets ( Buff );

        // Copier le message saisi dans la zone partagée
        CopyMemory ( (void *) Buffer, (void *) Buff, sizeof(Buff) );
    }
    // SINON SI ( On veut lire en zone partagée ) ALORS
    else if ( Buff[0] == 'L' )
    {
        // Copier la zone partagée dans une zone locale
        CopyMemory ( (void *) Buff, (void *) Buffer, sizeof(Buff) );

        // Afficher le message lu
        printf ( "Message reçu= %s\n", Buff );
    }
    // FINSI
}
// FINFAIRE
}

// Fermer la vue sur la mémoire partagée
UnmapViewOfFile( (void *) Buffer);

// Fermer l'objet FILE_MAPPING
CloseHandle(HandlerSHM);

// FIN
return 0;
}
```

VI.5.UTILISATION D'UN TIMER

Bien que cela ne fasse pas vraiment partie de l'objet de cet ouvrage (traitement des processus), nous donnons ici un exemple de traitement du timer courant fourni avec WINDOWS. En effet, dans une application employant des processus parallèles, l'usage d'un timer est souvent nécessaire.

Cet exemple permet de mettre en évidence la granulométrie très faible du timer standard (environ 16 ms). Il est à noter que cette valeur correspond à la norme POSIX 4 et qu'elle est la même pour les timers standards de LINUX.

Cette définition semble bien trop grossière pour la plupart des exigences en matière d'applications temps réel: il faudra alors avoir recours à des timers dits «à hautes performances» possédant des granulométries bien plus fines, souvent fournis avec les distributions de DELPHI ou de BORLAND C++. Le principe du traitement reste sensiblement le même.

```
#include <windows.h>
#include <stdio.h>
#define IDT_TEST 100

SYSTEMTIME St;
float Date;
float AncienneDate;

//-----
// Fonction gestionnaire de timer.
// Cette fonction est appelée par la procédure par défaut de traitement des événements
// de la fenêtre principale (dans le cas WM_TIMER).
//-----
VOID CALLBACK Proc( HWND hwnd, UINT message, UINT idTimer, DWORD dwTime )
{
    // DEBUT
    // Lire la date courante
    GetSystemTime ( &St );

    // La convertir en flottant et en secondes
```

```
Date = St.wHour*3600 + St.wMinute*60 + St.wSecond + ( (float) St.wMilliseconds/1000);

// SI (On dispose d'une ancienne date) ALORS
if ( AncienneDate >= 0 )
{
    // Afficher la difference entre nouvelle date et ancienne date
    printf ( "-%6.3lf s\n", Date-AncienneDate );
// FINSI
}

// Prendre la nouvelle date pour ancienne date
AncienneDate = Date;
// FIN
}

// Procedure principale
int WINAPI WinMain ( HINSTANCE hThisInstance, HINSTANCE hPrevInstance, LPSTR lpstCmdLine, int
nCmdShow )
{
// DONNEES LOCALES
    UINT    uiCrTimer;
    MSG     msg;

// DEBUT
    // Init. ancienne date (pour ne pas afficher la première durée)
    AncienneDate = -1;

    // SI ( Le passage du processus en priorité temps réel a échoué ) ALORS
    if ( !SetPriorityClass ( GetCurrentProcess(), REALTIME_PRIORITY_CLASS ) )
    {
        // Le signaler
        printf ( "echec changement de classe de priorité du processus\n" );
// FINSI
    }

    // Initialisation du timer
    // REMARQUE: si aucune fonction TIMERPROC n'était spécifiée (valeur NULL), il faudrait inclure
    // les traitements correspondant au timer dans une procédure de fenêtre (cas WM_TIMER)
    uiCrTimer = SetTimer ( NULL, IDT_TEST, 500, (TIMERPROC) Proc );

    // SI ( L'initialisation du timer a échoué ) ALORS
    if ( uiCrTimer == 0 )
    {
        // Le signaler et terminer l'application
        printf ( "Echec set timer\n" );
        Sleep ( 5000 );
        return (-1);
// FINSI
    }

    // Cette boucle est obligatoire même s'il n'y a pas de fenêtre déclarée, pour que
    // la procédure par défaut de traitement des événements soit activée: c'est elle qui
    // appelle le gestionnaire de timer

    // TANT QUE ( Le processus est actif ) FAIRE
    while ( TRUE )
    {
        // Detecter les événements
        if ( !GetMessage (&msg, (HWND) NULL, 0, 0) ) break;
        TranslateMessage(&msg);
        DispatchMessage(&msg);
// FINFAIRE
    }

    // Supprimer le timer
    KillTimer ( NULL, IDT_TEST );

// FIN
    return (0);
}
```