INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES CENTRE VAL DE LOIRE

Module système II
Les threads

Promotion : 2020

Département : STI

3ème année

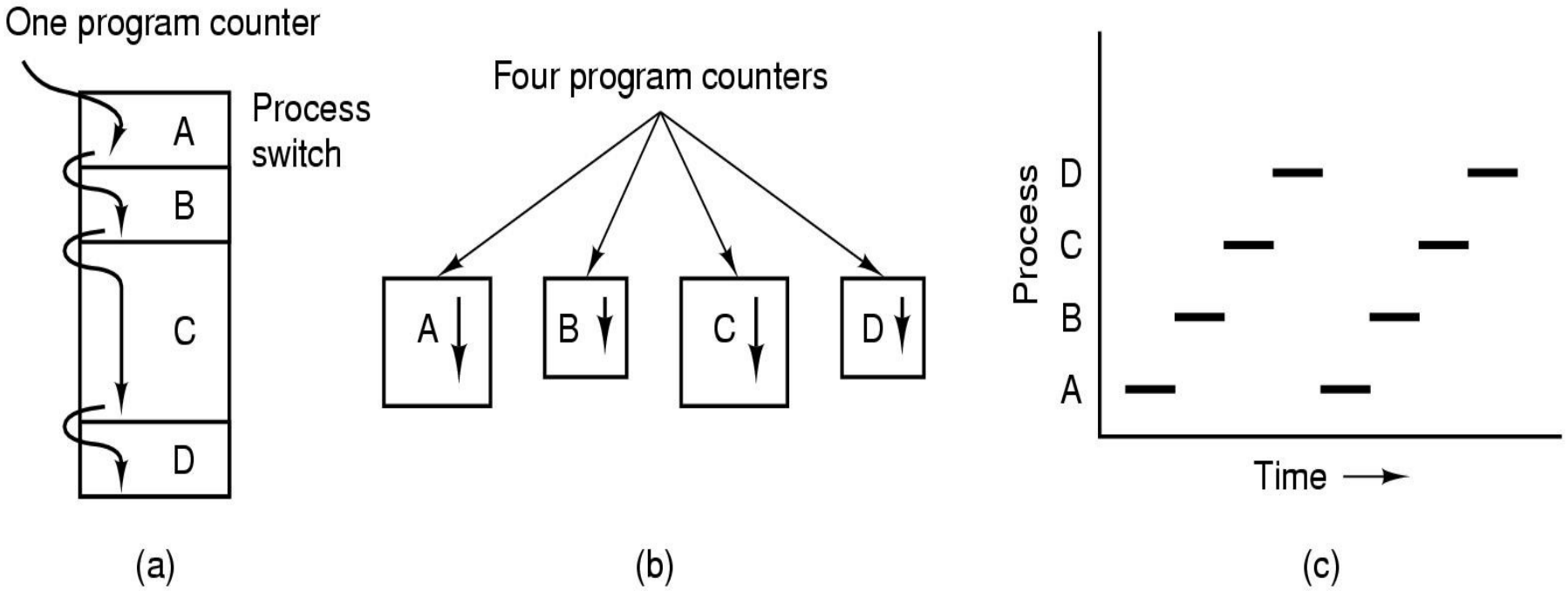
Année 2017_2018

M. Szpieg

Processus et Threads

Basé sur livres (et slides) de :
- A. TANENBAUM, Modern Operating Systems
- SILBERSCHATZ *et al.* : Operating Systems Concepts

Modèle d'exécution de processus



- Multiprogrammation de quatre processus sur un système mono processeur (pas de multi-threading)
- On obtient un modèle de 4 processus indépendants et séquentiels
- Un seul processus est actif au même instant

Création de processus

Les événements à l'origine de la création d'un processus

1. Initialisation système.
2. Exécution d'un appel système de création de processus
3. Demande de l'utilisateur pour la création d'un nouveau processus
4. Initialisation par un traitement par lot

2 et 3 :

UNIX : fork (clone du processus)

Win32 : CreateProcess(process creation + chargement du code dans le nouveau processus)

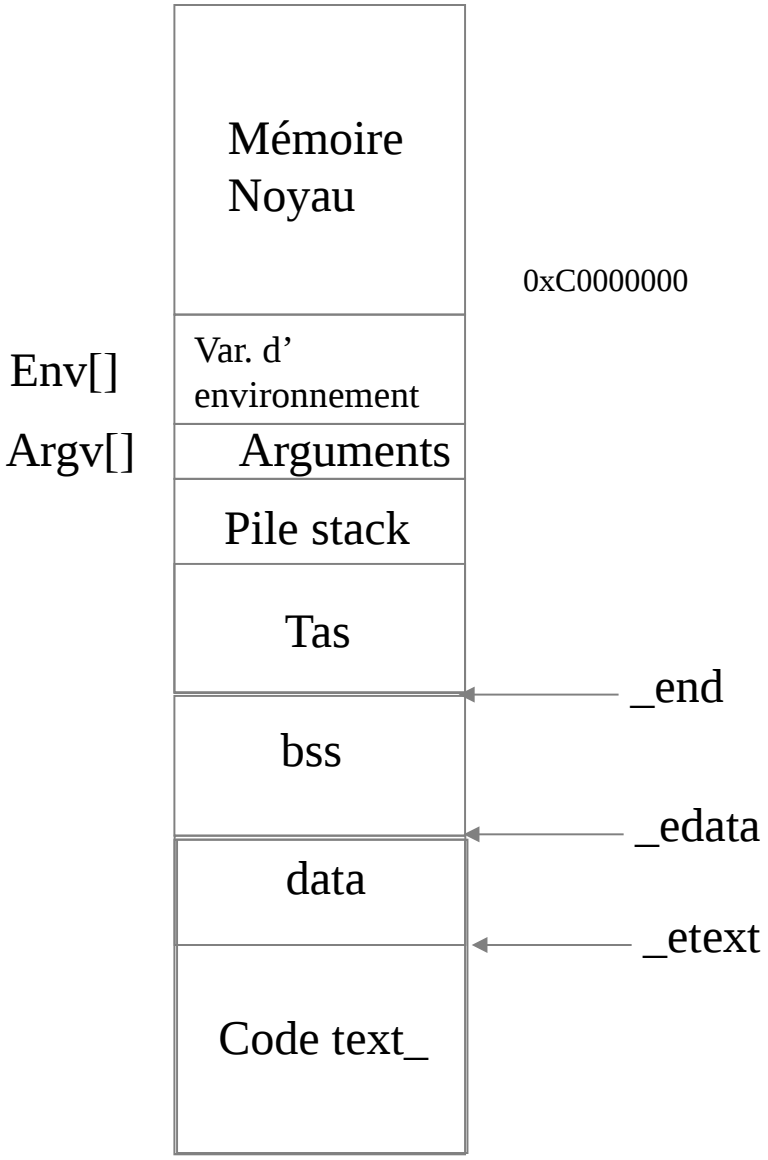
Espace d’adressage d’un processus.

Chaque processus possède son propre espace d'adressage virtualisé par le noyau (Une opération traduit ces adresses virtuelles en adresses physiques par l'intermédiaire d'un composant appelé MMU (Memory Management Unit) « voir suite du module »

Rappels :

- Code : code du prog.
- data : var. globales initialisées, constantes
- BSS (Block Started By Symbol) : zone de données non initialisées (e.g. var. globales)
- Tas (Heap) : allocations dynamiques (limité à 4Gb en 32 bits, 16 Gb en 64, etc.)
- Pile : var. locales

- *NB : dessin pas à l’échelle et représentant un espace d'adressage pour microprocesseur et OS 32 bits*



Terminaison d'un processus

Conditions qui donnent fin à un processus :

1. Sortie normale (volontaire)
2. Sortie sur erreur exit() (volontaire)
3. Erreur fatale (involontaire)
4. Arrivée d'un signal (volontaire ou involontaire)

Hiérarchie de processus

- Un processus parent crée des processus enfants, et les enfants peuvent créer leurs propres processus
- Formes de hiérarchie (arbre généalogique)
  - Sous UNIX on a la notion de groupes de processus "process group"
- Windows n'a pas tout à fait la même conception
  - Tous les processus sont créés au même niveau
  - Le parent reçoit un jeton spécial appelé « handle », qui lui permet de contrôler l'enfant. Cependant il est possible de passer ce « handle » à un autre processus, ce qui invalide la notion de hiérarchie.
  - Sous UNIX les processus ne peuvent pas renier leurs enfants. Remarque si un parent meurt un processus sous UNIX peut se rattacher au processus « init »

Contrôle de processus

- Etats d'un processus

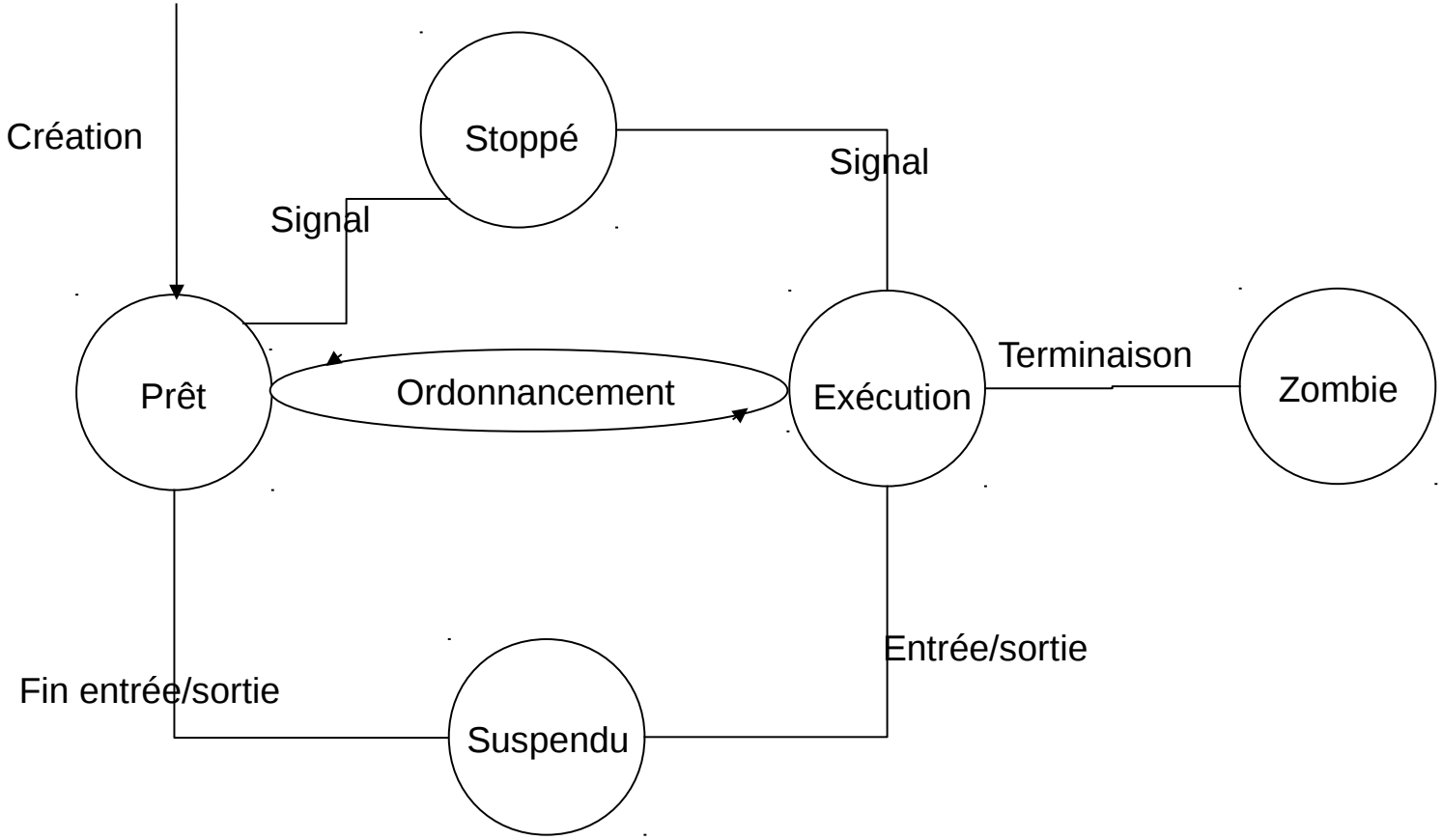


Table des processus

Process management	**Memory management**	**File management**
Registers	Pointer to text segment	Root directory
Program counter	Pointer to data segment	Working directory
Program status word	Pointer to stack segment	File descriptors
Stack pointer		User ID
Process state		Group ID
Priority		
Scheduling parameters		
Process ID		
Parent process		
Process group		
Signals		
Time when process started		
CPU time used		
Children's CPU time		
Time of next alarm		

Champs d'une entrée de la table de contrôle des processus (ou PCB : process control blocks)

Les Threads

- *Thread* = fil d'exécution
  - C'est un déroulement particulier du code du programme qui s'exécute parallèlement à d'autres entités du processus.

À la différence des processus générés par les appels systèmes fork() ou exec(), les threads sont des processus allégés (lightweight process) qui réclament peu de ressources pour le changement de contexte (registres processeur, segment de données, table des signaux etc.) puisque tout est commun.

  - Les différents threads d'un processus partagent un même espace d'adressage virtuel.

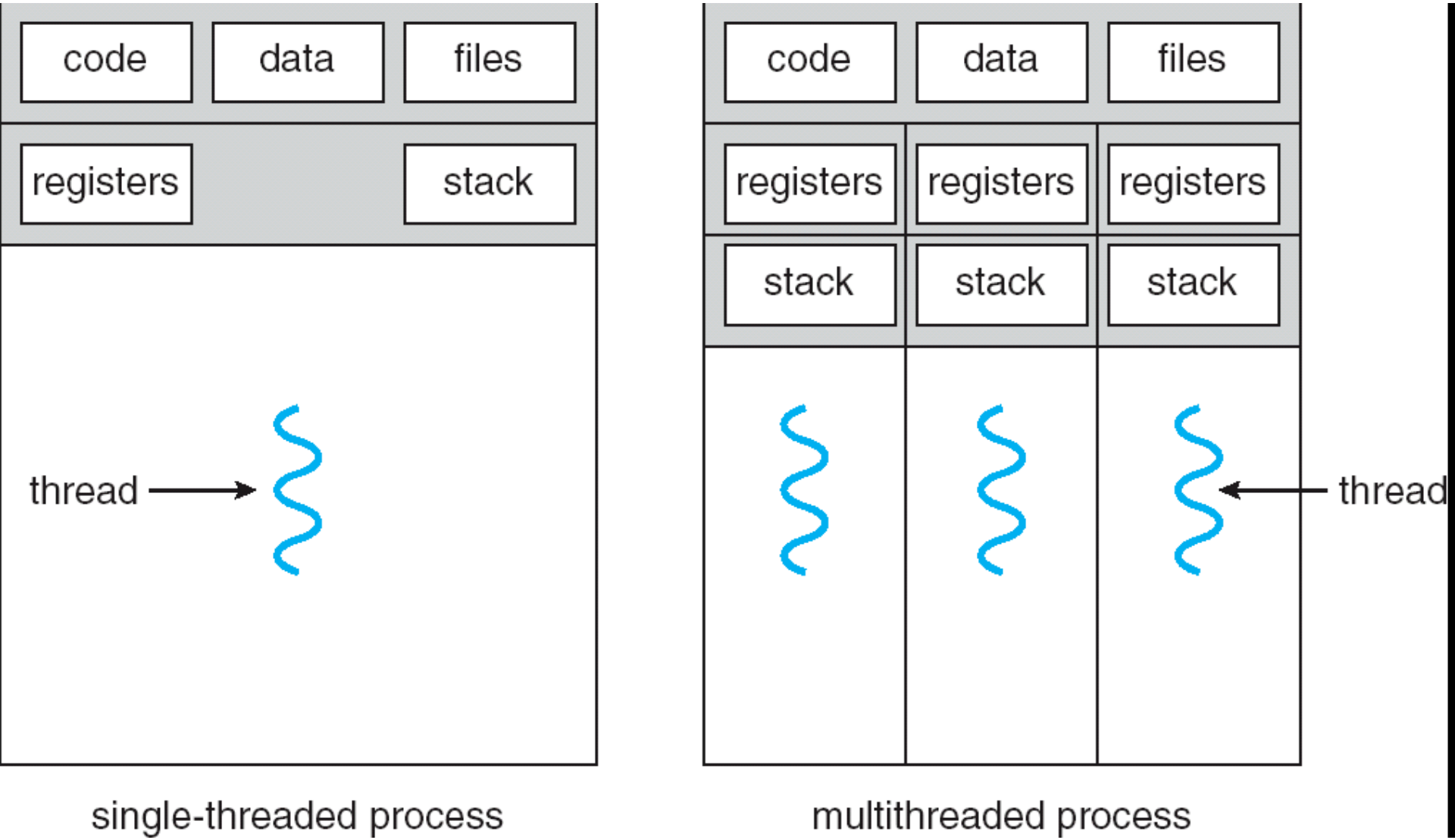
Thread et vocabulaire

Attention au vocabulaire !!! ,

On utilise parfois l'appellation « lightweight process » ou LWP pour différencier les threads utilisateur des threads noyau ce qui peut porter à confusion (voir plus loin)

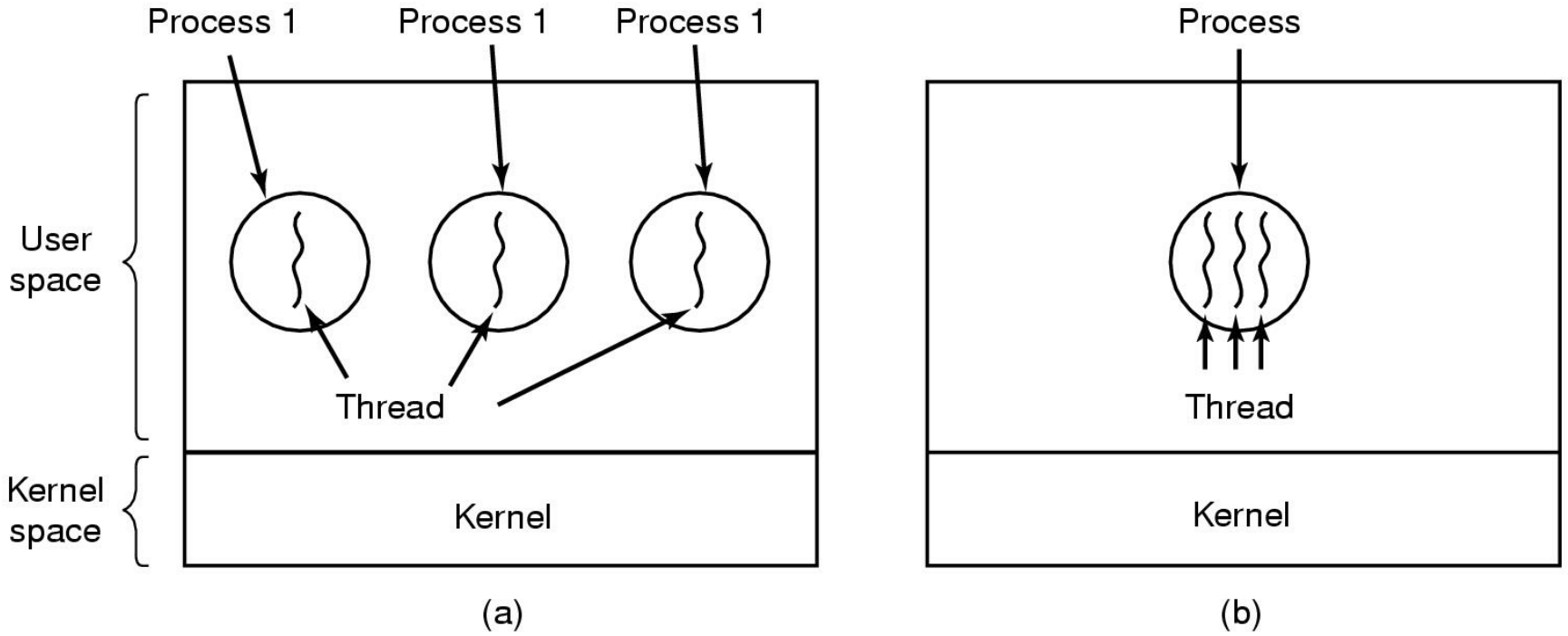
Le terme thread (traduction flux d'instructions de code) se retrouve dans d'autres contextes, par exemple on parle de processeur multithreadé, dans ce cas le multi-threading est la capacité pour un seul processeur physique d'exécuter plus d'une instruction à la fois, on est dans un contexte de thread matériel.

Threads principe



Processus au sens premier

Processus et thread



Un processus qui n'utilise pas la création de thread peut être considéré comme un thread

a) Trois processus

b) Un processus contenant trois threads

Thread vs processus

- Les threads d’une même application partagent les mêmes données statiques et dynamiques contrairement aux processus affiliés par fork() ou exec().
- Par contre chacun d’entre-eux possède son propre contexte d’exécution (registre processeur, compteur d’instruction (ou ordinal) et pile)
- Communication entre threads plus simple qu’entre processus.
- Les différents threads d’une application, partageant un même espace d ’adressage, nécessitent une synchronisation (voir plus loin).
- Les threads sont intéressants pour les applications assurant plusieurs tâches en parallèle.

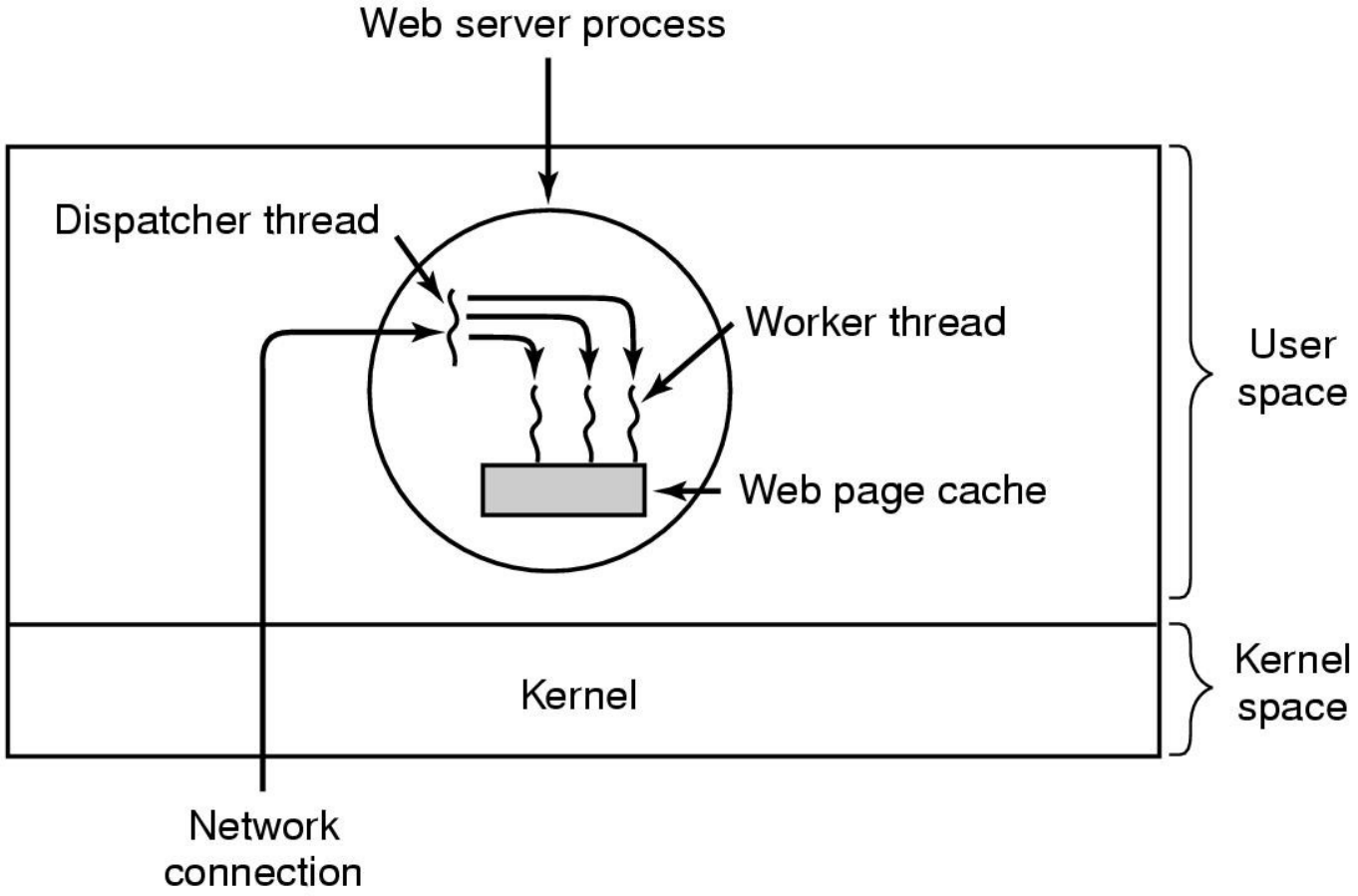
Partages entre threads d'un processus

Partagé entre threads	Propre à chaque thread
Espace mémoire virtuel	Pointeur d'instruction
Variables globales	Registres
Descripteurs de fichiers ouverts	Pile
Processus fils	Etats
Signaux en attente et gestionnaires de signaux	

Bénéfices

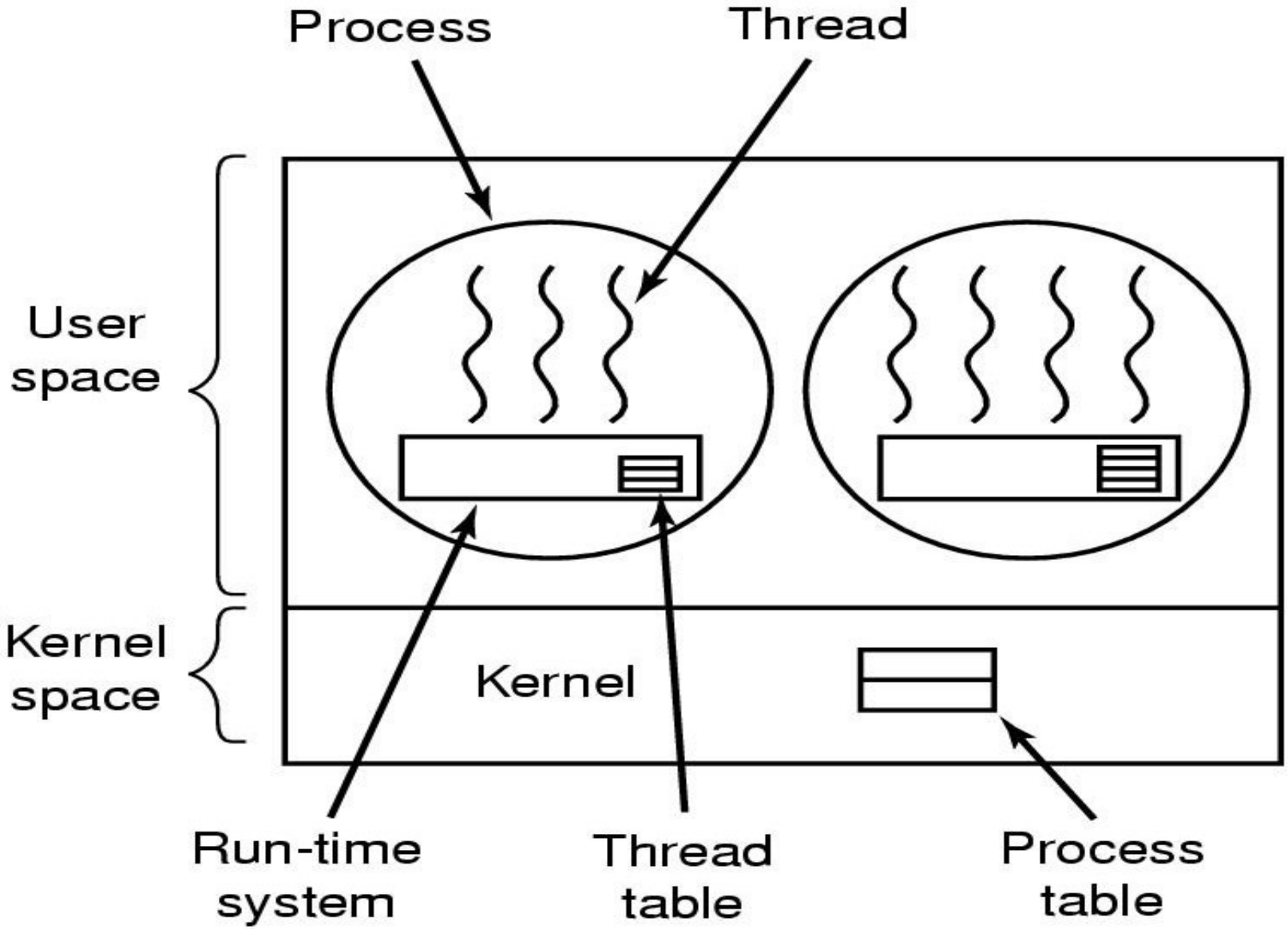
- Rapidité
- Partage des ressources
- Économie mémoire
- Utilisation de système d'exploitation multitraitement symétrique SMP qui comporte une architecture matérielle multiprocesseur ou multi cœurs où deux ou plusieurs processeurs identiques sont connectés à une mémoire principale partagée et sont gérés par un système d'exploitation unique.

Exemple d'utilisation de threads



Processus serveur web multi-threadé

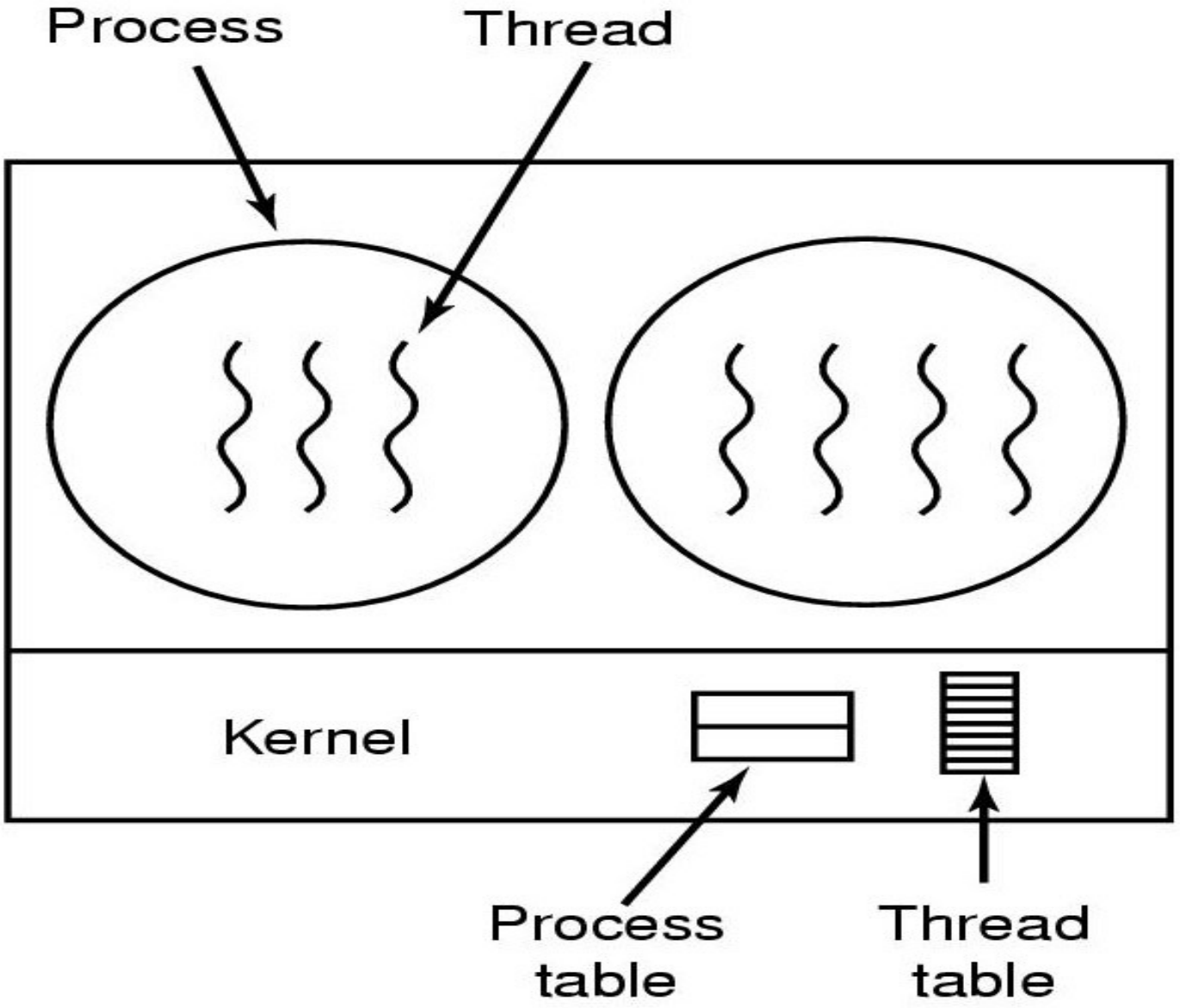
Implémentation des threads dans l'espace utilisateur



Les threads au niveau utilisateur

- POSIX Pthreads
- C-threads (MACH system)
- Solaris threads

Gestion des threads au niveau noyau



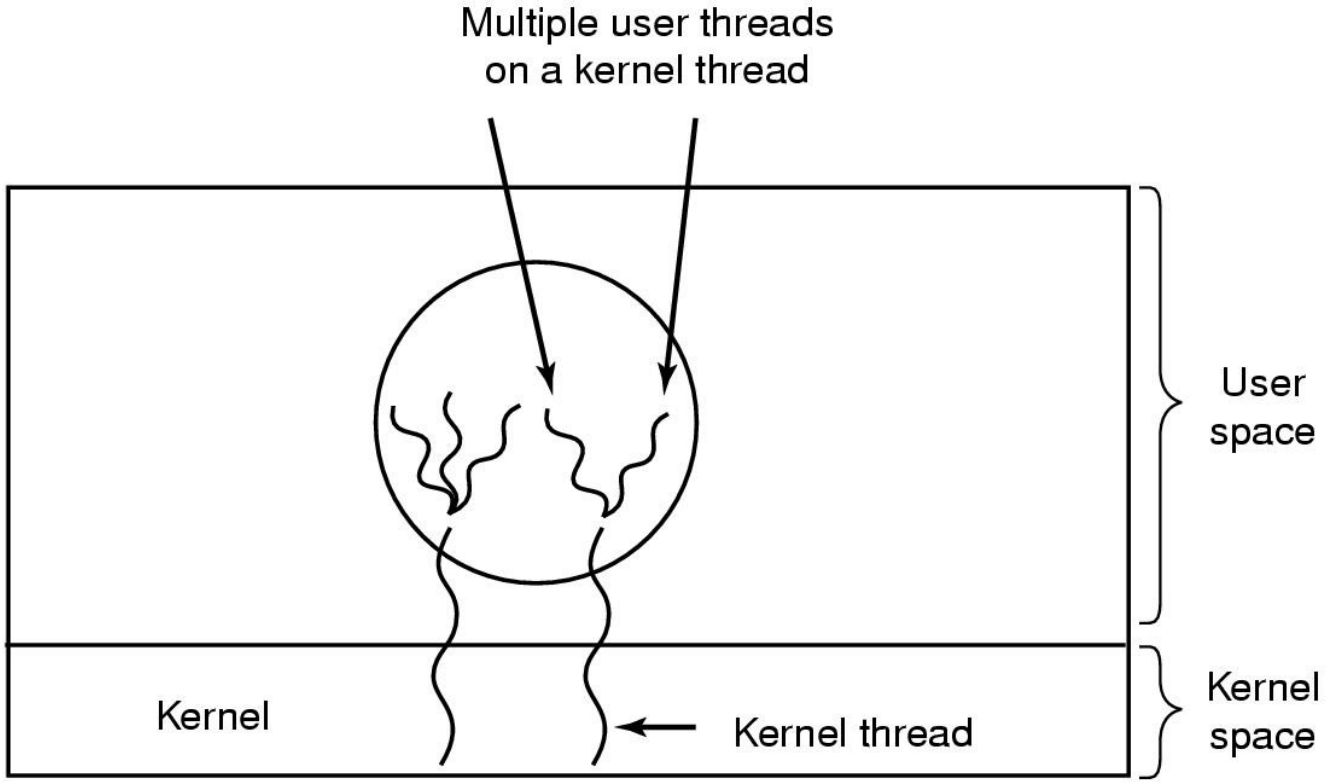
Threads au niveau Kernel

- Windows 95/98/NT/2000/(Vista)
- Solaris
- Tru64 UNIX
- BeOS
- Linux
- ...

User Threads vs Kernel Threads

	Implémentation noyau	Implémentation utilisateur
Respect de la norme POSIX 1.c	Pas prévu dans toutes les versions du noyau	portable sur UNIX pas de modification du noyau
Création d'un Thread	appel système	pas d'appel système, coût plus faible en ressources
Commutation entre Threads	Géré par le noyau	Commutation bibliothèque Pas de changement de contexte
Ordonnancement	Chaque thread dispose de temps cpu donné par le noyau	Les thread se partagent le temps cpu du processus d'accueil
Priorité des tâches	Les threads peuvent avoir des priorités différentes	Priorité inférieure au processus d'accueil
Parallélisme	Le noyau peut répartir les threads sur différents processeurs, si présents	Pas de parallélisation car un seul processus

Implémentation hybride



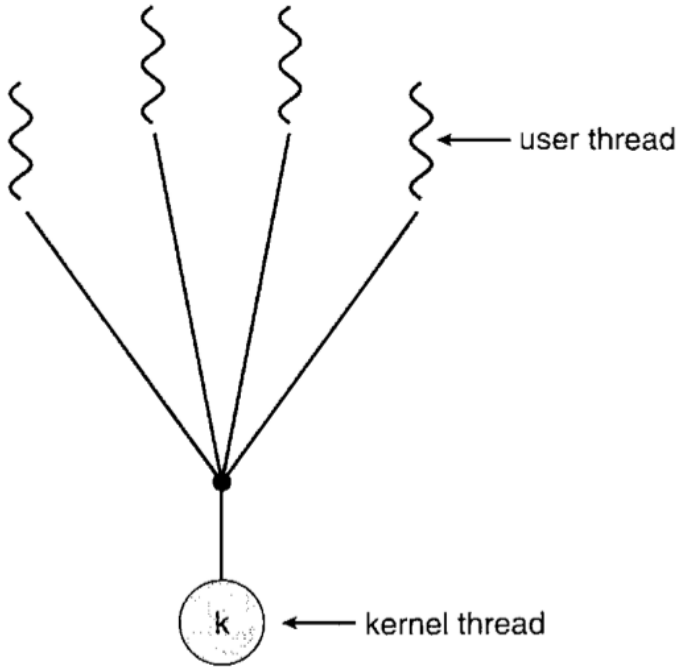
Multiplexage des threads au niveau utilisateur avec des threads au niveau noyau

Modèles de multi-threading

- Many-to-One
- One-to-One
- Many-to-Many

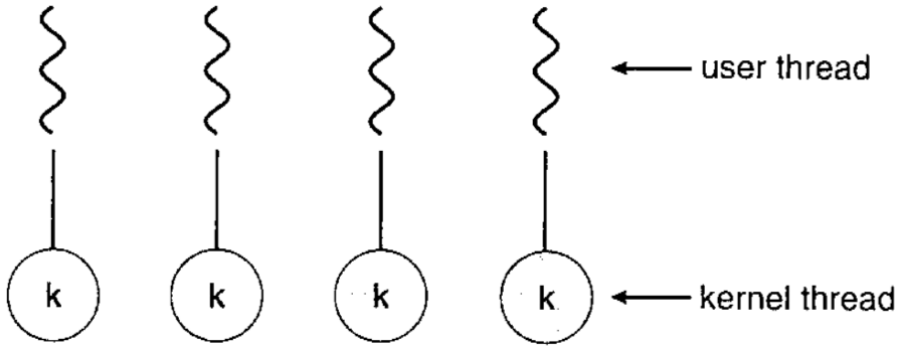
Multithreading – Many-to-One

- Gestion du thread par la librairie mode user (efficace)
- Processus complètement bloqué si un thread fait un appel système bloquant
- Un seul process kernel pour tous les threads empêche de les paralléliser sur une machine multi-proc.
- Ex. : Green threads (Solaris), GNU Portable threads



Multithreading – One-to-One

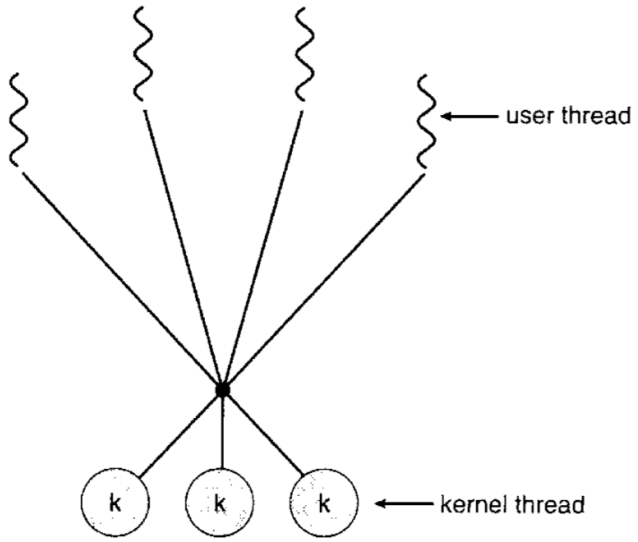
- Augmente la concurrence puisque tous les threads s'exécutent en même temps



- Plus de pb d'appel système
- Parallélisation possible sur multiprocesseurs
- Pb : coût de création de thread kernel => limitation du nombre sur tous les OS le supportant
- Ex. : Linux, tous les Windows >= 95, Solaris >9

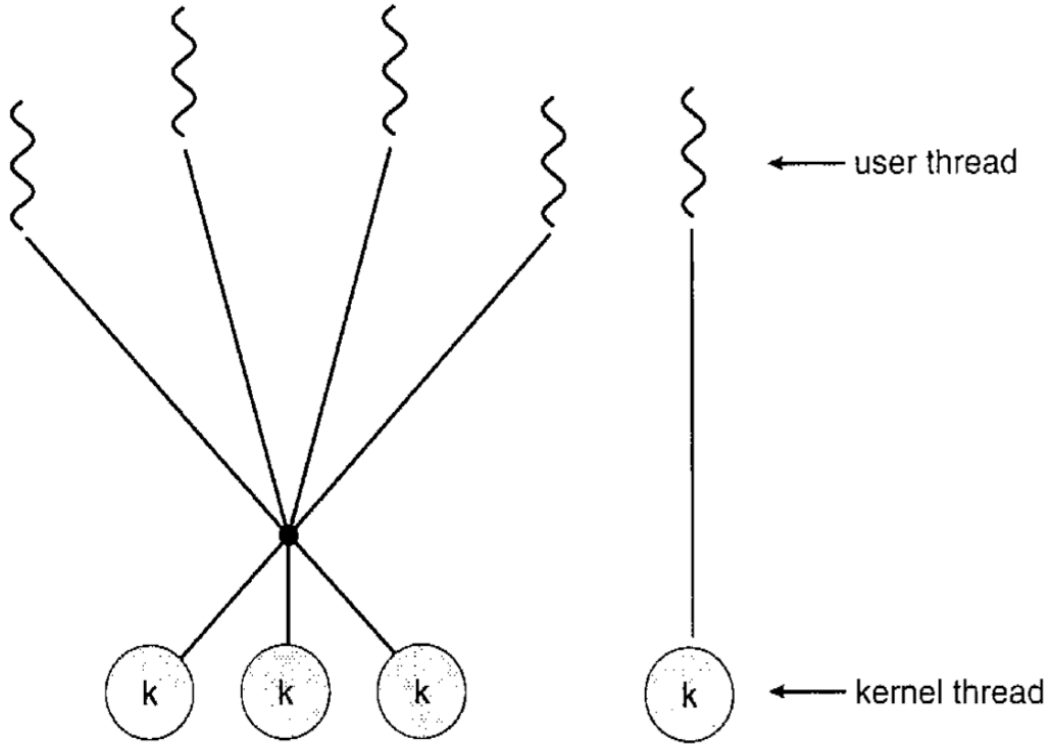
Multithreading – Many-to-Many

- Multiplexe de nombreux user threads en quelques kernel threads
- On peut considérer que chaque kernel thread représente un cœur virtuel auquel on affecte un certain nombre de threads user
- Le nombre peut dépendre de l' application et/ou du système (un processus peut obtenir plus de threads kernel si elle lancée sur une machine multiprocesseurs).
- Compromis entre les 2 précédents (plusieurs threads, exécution parallèle , appel système bloquant => réveil d'un autre thread)



Multithreading – Many-to-Many: Modèle à 2 niveaux

- Many-to-many permet à un thread utilisateur à se lier à un thread noyau :
  - IRIX,
  - HP-UX,
  - Tru64 UNIX
  - Solaris <9



Exemple d'implémentation de threads

- Contexte de l'exemple :
  - un processus crée un thread de sommation (1 à *i* (argv))
  - Il attend que ce thread lui retourne le résultat de la sommation
  - Puis il affiche le résultat reçu

Exemple – POSIX Pthreads

Librairie Pthread : User level ou Kernel level

1 – Processus main() : déclarations

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

int sum; /* this data is shared by the thread(s) */
void *runner(void *param); /* the thread */

int main(int argc, char *argv[])
{
   pthread_t tid; /* the thread identifier */
   pthread_attr_t attr; /* set of thread attributes */

   if (argc != 2) {
      fprintf(stderr,"usage: a.out <integer value>\n");
      return -1;
   }
   if (atoi(argv[1]) < 0) {
      fprintf(stderr,"%d must be >= 0\n",atoi(argv[1]));
      return -1;
   }
```

Exemple – POSIX Pthreads

2 – Processus main() : création du thread

```
    /* get the default attributes */
    pthread_attr_init(&attr);
    /* create the thread */
    pthread_create(&tid,&attr,runner,argv[1]);
    /* wait for the thread to exit */
    pthread_join(tid,NULL);

    printf("sum = %d\n",sum);
}
```

Exemple – POSIX Pthreads

3 – Fonction runner() : code du thread de calcul

```
/* The thread will begin control in this function */
void *runner(void *param)
{
   int i, upper = atoi(param);
   sum = 0;

   for (i = 1; i <= upper; i++)
      sum += i;

   pthread_exit(0);
}
```

Exemple – Win32 threads

Librairie de threads Win32 : kernel level

1 - Déclarations

```
#include <windows.h>
#include <stdio.h>
DWORD Sum; /* data is shared by the thread(s) */
/* the thread runs in this separate function */

DWORD WINAPI Summation(LPVOID Param)
{
   DWORD Upper = *(DWORD*)Param;
   for (DWORD i = 0; i <= Upper; i++)
      Sum += i;
   return 0;
}
```

Exemple – Win32 threads

2 - Main

```
int main(int argc, char *argv[])
{
   DWORD ThreadId;
   HANDLE ThreadHandle;
   int Param;
   /* perform some basic error checking */
   if (argc != 2) {
      fprintf(stderr,"An integer parameter is required\n");
      return -1;
   }
   Param = atoi(argv[1]);
   if (Param < 0) {
      fprintf(stderr,"An integer >= 0 is required\n");
      return -1;
   }
```

Exemples – Win32 threads

3 – Création du thread

```
// create the thread
ThreadHandle = CreateThread(
   NULL, // default security attributes
   0, // default stack size
   Summation, // thread function
   &Param, // parameter to thread function
   0, // default creation flags
   &ThreadId); // returns the thread identifier

if (ThreadHandle != NULL) {
   // now wait for the thread to finish
   WaitForSingleObject(ThreadHandle,INFINITE);

   // close the thread handle
   CloseHandle(ThreadHandle);

   printf("sum = %d\n",Sum);
}
}
```

Threads POSIX : Détails d’implémentation

- C’est l ’appel système clone() qui permet, sous Linux, de générer un thread
  <linux/sched.h>
  int __clone (int (*fonction) (void *arg),void *pile, int attributs,void *arg);
- En général on utilise **plutôt** les fonctions de la bibliothèque Linux « pthread » et non pas directement l ’appel système clone() d’où (#include <pthread.h>).
- Attention lors de la compilation ajouter la constante _REENTRANT sur la ligne de commande de la compilation et l ’option -lpthread pour l’éditeur de lien

Création de thread

- int pthread_create (pthread_t * thread, pthread_attr_t * attributs, void * (* fonction) (void * argument), void * argument);
  - pthread_t * thread : pointeur sur la routine formant le contenu du thread. Il sert d'identifiant (comme le pid pour un processus).
  - pthread_attr_t * attributs : attributs donnés au thread
    - detashthread :
      - PTHREAD_CREATE_JOIGNABLE : Attend la consultation de la valeur de retour par un autre thread avant de se terminer.
      - PTHREAD_CREATE_DETACHED : Terminaison du thread sans attendre la consultation d 'un autre thread, pas de valeur de retour.
    - stackaddr et stacksize configuration de la pile
    - schedpolicy (SCHED_OTHER,SCHED_RR,SCHED_FIFO) , schedparam (contient la priorité du thread) : règle l'ordonnancement attention TR seulement !! voir plus loin.
    - inheritsched (PTHREAD_EXPLICIT_SCHED, PTHREAD_INHERIT_SCHED): indique si l'on donne les règles d 'ordonnancement du thread de manière explicite ou si celui-ci hérite des règles de son créateur.
    - Si ce pointeur est égal à NULL alors héritage des attributs de son créateur

Création de thread

- Ces attributs se manipulent avec les fonctions suivantes :
  - pthread_attr_init : initialisation de la variable contenant les attributs
  - pthread_attr_setXXX() ou pthread_attr_getXXX() pour modifier ou obtenir les valeurs d ’attributs et où XXX est le nom de l ’attribut.
    Exemples : pthread_attr_getschedparam, pthread_attr_setschedpolicy …
- void * (* fonction) (void * argument) : pointeur sur la fonction principale du thread (= « main » du thread.
- void * argument : pointeur passé en argument à la fonction principale du thread.
- Le nombre de threads est limité par la constante symbolique PTHREAD_THREADS_MAX (1024 avec la bibliothèque LinuxThread).

Création de thread

Remarque sur le paramètre « schedparam » :

Seules les politiques d'ordonnancement « temps réel » utilisent ce paramètre :

```
-->chrt -m
SCHED_OTHER priorité min/max     : 0/0
SCHED_FIFO priorité min/max      : 1/99
SCHED_RR priorité min/max        : 1/99
```

Voir « nice() » pour « sched_other »

Fin d’un thread

- Deux manières de terminer un thread :
  1. La fonction principale du thread se termine
  2. Utilisation de la fonction « void pthread_exit(void* retour); »

  /!\ => Ne pas utiliser la fonction ’exit()’ qui mettrait fin à l’ensemble de l’application.
- Lorsque cette fonction ‘pthread_exit’ est appelée les fonctions de libération de ressources sont exécutées (voir plus loin cleanup_push)
- Récupération de la valeur de retour du thread
  1. La fonction « int pthread_join (pthread_t thread, void ** retour); permet de récupérer par un autre thread la valeur de retour.
     Le thread utilisant cette fonction est suspendu dans l’attente de la valeur retour. Si retour est un entier, attention au transtypage.

Fin d’un thread (suite)

- Si la valeur de retour n ’est pas utile, la fonction « int pthread_detach (pthread_t thread); permet au thread de se terminer et de libérer la place mémoire sans attendre que sa valeur de retour soit récupérée par un autre thread.

- Un thread peut en annuler un autre, dans certaines conditions (analogie avec un signal de type SIGKILL pour un processus), dans ce cas la valeur « retour » est égale à PTHREAD_CANCEL. Attention à ne pas utiliser la valeur numérique correspondante à d’autres fins.
- Voir 1_pthread_create.c et 2_pthread_join.c (TD)

Récupération de son identifiant

- La fonction pthread_t pthread_self(void);
  permet à un thread de récupérer son propre identifiant (analogie getpid()).
- La fonction pthread_equal permet de comparer deux identifiants.

Attribut d’un thread (1)

- On regroupe les attributs d’un thread dans un type opaque pthread_attr_t. On active ces attributs au moment de la création du thread grâce au deuxième argument de la fonction « pthread_create() ».
- Dans un premier temps il faut faire appel à la fonction
  int pthread_attr_init (pthread_attr_t *attributs);
  qui permet d’initialiser les attributs et la zone mémoire qui les contient.
- Ensuite, on utilise, pthread_attr_setXXX() ou pthread_attr_getXXX() pour modifier ou obtenir les valeurs d ’attributs où XXX est le nom de l ’attribut.
  Exemples : pthread_attr_getschedparam, pthread_attr_setschedpolicy …

Attribut d’un thread (2)

- L’instance attribut de type pthread_attr_t peut être détruite par la fonction « int pthread_attr_destroy (pthread_attr_t * attributs) »
- Cette fonction libère les ressources utilisées par attributs néanmoins cela n’affecte pas les threads créés précédemment et ayant utilisé cette variable.
- On ne peut pas modifier un attribut avec les fonctions du type pthread_attr_setXXX() une fois le thread créé, il faut donc effectuer ces réglages avant l’appel à la fonction pthread_create(); (sauf attribut particulier).

Attribut d’un thread (3)

- Néanmoins, la fonction pthread_detach() permet de modifier l’attribut « detachstate » de manière dynamique.
- Avant la création du thread cet attribut se manipule avec les fonctions suivantes :
  - int pthread_attr_getdetachstate (const pthread_attr_t *attributs, int * valeur);
  - int pthread_attr_setdetachstate (const pthread_attr_t *attributs, int * valeur);
- Où « valeur » peut être égale à PTHREAD_CREATE_JOINABLE, valeur par défaut, ou PTHREAD_CREATE_DETACHED.

Attribut d'un thread – la pile

- Les attributs stackaddr et stacksize permettent de configurer la taille de la pile utilisée par un thread.
- En effet, chaque thread a sa propre pile pour s'exécuter.
- Les constantes symboliques
  _POSIX_THREAD_ATTR_STACKSIZE
  _POSIX_THREAD_ATTR_STACKADDR
  PTHREAD_ATTR_STACKMIN
  donnent des informations sur la pile. Néanmoins on y fera appel que pour des threads bien particuliers, dans le cas général, on se contentera des valeurs par défaut.
- Les fonctions associées sont :
  - pthread_attr_getstackaddr(), pthread_attr_setstackaddr()
  - pthread_attr_getstacksize(), pthread_attr_setstacksize()

Attribut d'un thread – scheduling

- Les attributs schedpolicy,schedparam et inheritsched concernent l'ordonnancement.
- Ils sont disponibles **uniquement** si la constante symbolique _POSIX_THREAD_SCHEDULING est égale à vrai ds unistd.h
- Les fonctions :
  - int pthread_attr_getschedpolicy(const pthread_attr_t *attributs, int *valeur);
  - int pthread_attr_getschedpolicy(const pthread_attr_t *attributs, int valeur);
    permettent de manipuler ces attributs.
- Valeur peut prendre les valeurs SCHED_OTHER, SCHD_RR, SCHED_FIFO.
- On peut aussi modifier de manière dynamique ces comportements avec la fonction pthread_setschedparam();

Attribut d’un thread – scheduling

- L’ attribut inheritsched indique si le thread dispose de son propre ordonnancement ou si il en a hérité du père (schedparam et schedpolicy **sont alors ignorés**).
- Les valeurs de cet attribut peuvent être PTHREAD_EXPLICIT_SCHED ordonnancement spécifique ou PTHREAD_INHERIT_SCHED héritage du père

Attribut scope

int pthread_attr_setscope(pthread_attr_t *attr, int scope);

Définit comment sont interprétés les paramètres d'ordonnancement pour le nouveau thread. Une seule valeur est supportée dans LinuxThreads : PTHREAD_SCOPE_SYSTEM, qui signifie que tous les threads sont en compétition avec tous les processus en cours d'exécution pour le temps processeur.

L'autre valeur spécifiée par le standard, PTHREAD_SCOPE_PROCESS, signifie que les threads ne sont en compétition qu'avec les autres threads du même processus.

PTHREAD_SCOPE_PROCESS n'est pas supportée dans LinuxThreads(au jour où j'écris ces lignes 2016)

Annulation d’un thread

- Un thread peut en annuler un autre si celui-ci l ’accepte
- Équivalent de la fonction kill(SIGTERM,..) des processus classiques
  int pthread_cancel (pthread_t thread);
  - ceci enclenche pthread_exit et n’est possible que si le thread cible l’accepte
    « int pthread_setcancelstate (int etat_annulation, int * ancien_etat);
    avec int etat_annulation pouvant valoir soit
    - PTHREAD_CANCEL_ENABLE
    - PTHREAD_CANCEL_DISABLE
- Il existe un mécanisme de retardement de l ’annulation
  int pthread_setcanceltype (int type_annulation, int * ancien_type);
  avec type_annulation pouvant prendre les valeurs
  - PTHREAD_CANCEL_DEFERRED se termine lors d’un point annulation
  - PTHREAD_CANCEL_ASYNCHRONOUS
    arrêt dès réception de la requête (sans garantie)

Annulation d’un thread

- Les points d’annulations
  - Soit un appel système répertorié
    - close(), open(), pause() …
  - soit la fonction « void pthread_testcancel(void) » que l ’on répartit dans le programme
  - soit pthread_join()
  - soit pthread_cond_wait() et pthread_cond_timewait() qui permettent d ’attendre l ’arrivée d ’une condition avant l’utilisation d ’une ressource.
  - Points d’annulation implicites :
    - pthread_join(3)
    - pthread_cond_wait(3)
    - pthread_cond_timedwait(3)
    - pthread_testcancel(3)
    - sem_wait(3)
    - sigwait(3)

Envoi d’un signal à un thread

- pthread_kill
  - int pthread_kill(pthread_t thread, int sig);
  - Le thread qui envoie le signal est dans le même processus que le thread cible.

Libération des ressources utilisées par un thread

- Mise dans une pile de la fonction de libération de ressources.
  void pthread_cleanup_push (void (* fonction) void *argument), void *argument);
  premier argument, adresse de la routine à exécuter, deuxième argument élément passé à cette routine.
- Mise en route de cette fonction de libération de la ressource
  void pthread_cleanup_pop (int execution_routine);
  - si execution_routine est nul alors la fonction est retirée de la pile sans exécution
  - sinon la fonction est retirée de la pile avec exécution.

Libération des ressources utilisées par un thread

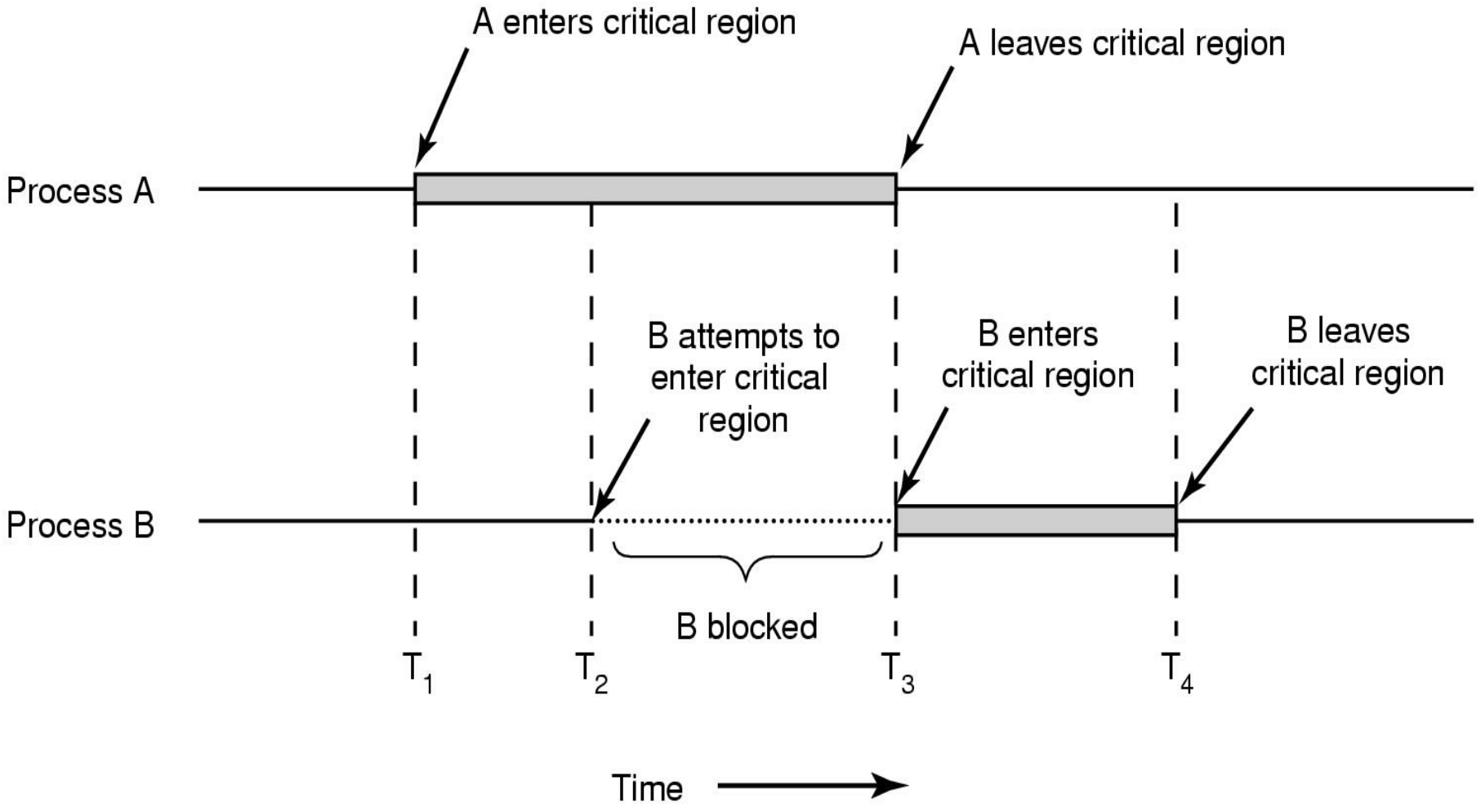
- Exemple
  - FILE * fp;
    fp= fopen (« monfichier », « r»);
    pthread_cleanup_push (fclose,fp);
  - char * buffer;
    buffer=malloc(BUFSIZE);
     pthread_cleanup_push (free,buffer);
  - ……..
  - Pthread_cleanup_pop (1); /* free(buffer); */
  - Pthread_cleanup_pop (1); /* close(fp); */

Synchronisation des accès aux ressources (Mutex)

- Les threads partagent des ressources communes il faut donc synchroniser les accès.
- Le mécanisme de protection utilisé est appelé MUTEX (mutuelle exclusion) de type pthread_mutex_t pouvant prendre deux états :
  - disponible ou verrouillé.
- L'initialisation du « mutex » peut se faire de manière statique ou dynamique
  - pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER
  - int pthread_mutex_init (pthread_mutex_t *mutex, const pthread_mutexattr_t *attributs);

    EXEMPLE : pthread_mutex_t mutex;
    pthread_mutexattr_t mutexattr;
    initialisation de mutex attribut;
    if ((mutex =malloc (sizeof (pthread_mutex_t)) == null) return (-1);
    pthread_mutex_init (&mutex, &mutexattr);
- La destruction du mutex se fait avec : pthread_mutex_destroy

Mutex et région critique



Synchronisation des accès aux ressources (Mutex)

- La fonction de verrouillage est :
  int pthread_mutex_lock (pthread_mutex_t * mutex);
  - si le mutex est libre, il se verrouille alors et un appel à cette fonction devient bloquant pour un autre thread.
  - Attention si le thread ayant verrouillé un mutex recommence l ’appel alors le système est bloqué.
- Fonction de déverrouillage :
  int pthread_mutex_unlock (pthread_mutex_t * mutex);

Synchronisation des accès aux ressources (Variable condition)

- Autre technique de synchronisation, les variables condition.
  - Si un thread doit attendre le déblocage d'un mutex ou l'arrivée d'un événement venant d'un autre thread, alors on emploie des variables condition.
  - Initialisation d 'une variable condition :
    - pthread_cond_t condition = PTHREAD_COND_INITIALIZER.
  - libération d 'une variable condition
    - int pthread_cond_destroy (pthread_cond_t * condition);

Synchronisation des accès aux ressources (Variable condition)

- Déroulement des étapes de mise en place de la variable condition:
  - thread qui attend la condition :
    - initialise la variable condition et le mutex
    - bloque le mutex
    - appelle la fonction pthread_cond_wait()
      - débloque le mutex
      - attente de la condition
      - arrivée de la condition, bloque le mutex, revient au thread.
    - Libère le mutex
  - thread réalisant la condition
    - réalise la condition
    - bloque le mutex lié à la condition
    - appel de pthread_cond_signal() ou pthread_cond_broadcast pour informer que la condition est remplie
    - débloque le mutex

Synchronisation des accès aux ressources (les sémaphores)

- Ne pas confondre les sémaphores IPC systemV (préfixe des fonctions sem) avec ceux de la norme POSIX.1b (préfixe des fonctions sem_)
- Ils servent à limiter la portion critique de code.
- int sem_init (sem_t * semaphore, int partage, unsigned int valeur);
  - premier argument : le sémaphore
  - deuxième argument : indique si le sémaphore est partagé au delà de l ’application
  - troisième argument : valeur initiale du compteur du sémaphore.
  - La portion de code critique ne peut être atteinte que si le compteur est supérieur à zéro
- la fonction sem_wait() permet l ’attente en utilisant un compteur strictement positif puis décrémente ce compteur en entrant.
- La fonction sem_post() permet d ’incrémenter le compteur en sortant de la portion critique.
- int sem_getvalue(sem_t * semaphore, int *valeur); permet de consulter le compteur

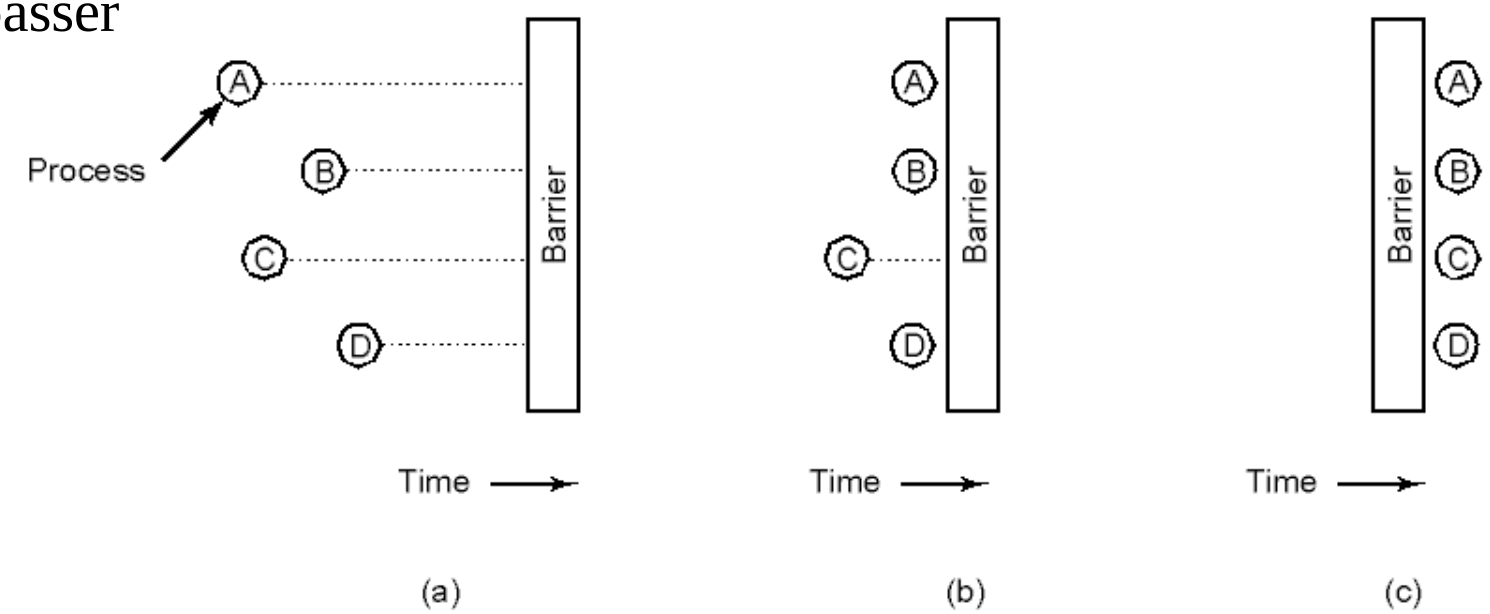
Synchronisation des accès aux ressources (les barrières)

Le but d'une barrière est de synchroniser plusieurs threads afin que l'ensemble des threads attend le dernier arrivé.

a) les threads avancent vers la barrière

b) Les threads A,B et D sont arrivés au niveau de la barrière et sont bloqués car le thread C n'est toujours pas arrivé.

c) Le thread C arrive, ils sont donc tous là, la barrière se lève et les laisse passer



Synchronisation des accès aux ressources (les barrières)

- Initialisation de la barrière (remarque count indique le nombre de threads nécessaires au niveau de la barrière pour qu'elle se lève)

```
int pthread_barrier_init(pthread_barrier_t *barrier,
pthread_barrierattr_t *barrier_attr, unsigned int count);
```

```
pthread_barrier_t barrier =
PTHREAD_BARRIER_INITIALIZER(count);
```

- Attente de la levée de la barrière

```
int pthread_barrier_wait(pthread_barrier_t *barrier);
```