Noyau d'un système d'exploitation INF2610

Chapitre 3 : Threads (Fils d'exécution)

Département de génie informatique et génie logiciel

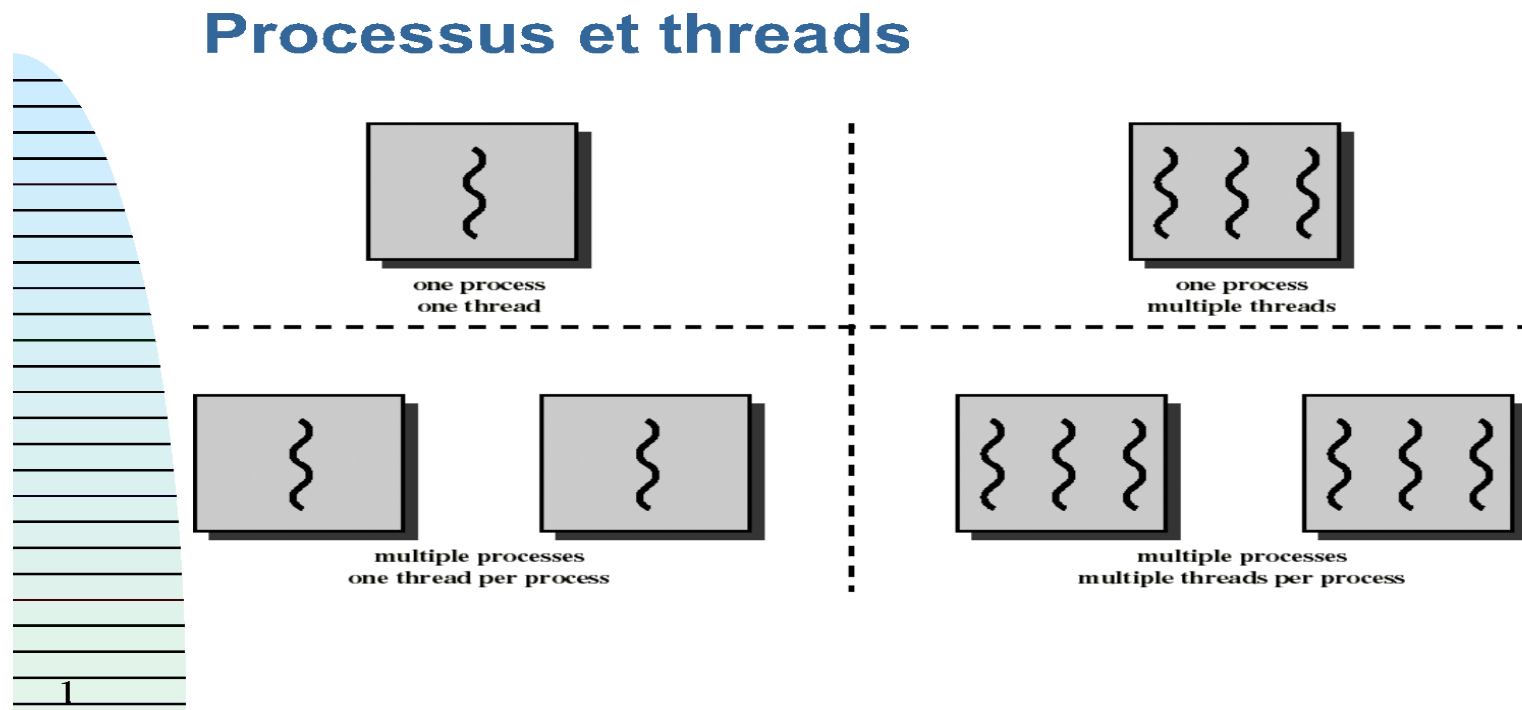
Hiver 2019

Chapitre 3 - Threads (Fils d'exécution)

- **Qu'est ce qu'un thread ?**
- **Usage des threads**
- **Threads POSIX**
  - **Création**
  - **Attente de la fin d'un fil d'exécution**
  - **Terminaison**
  - **Nettoyage à la terminaison**
- **Threads utilisateur vs threads noyau**

Qu’est ce qu’un thread ?

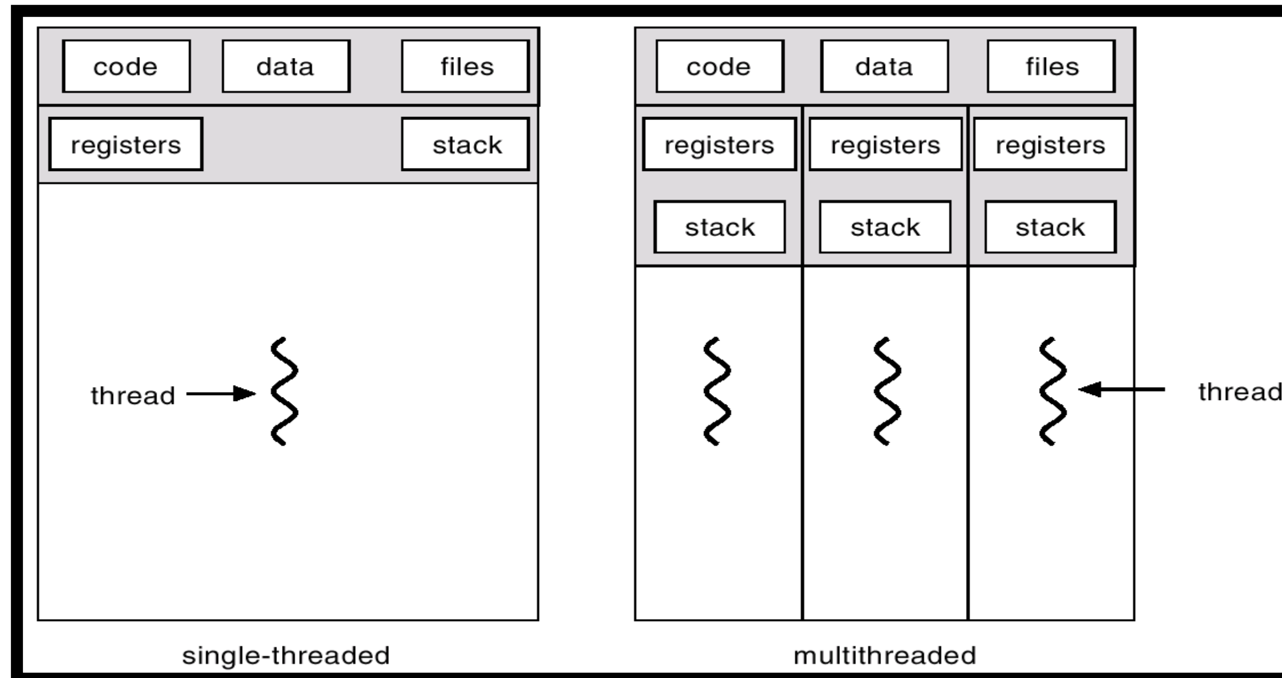
- Le modèle processus décrit précédemment est un programme qui s’exécute selon un chemin unique (compteur ordinal). On dit qu’il a un fil d'exécution ou flot de contrôle unique (single thread).
- De nombreux systèmes d’exploitation modernes offrent la possibilité d’associer à un même processus plusieurs chemins d’exécution (multithreading).



Qu’est ce qu’un thread ? (2)

- Un thread est une unité d'exécution rattachée à un processus, chargée d'exécuter une partie du programme du processus.
- Un processus est vu comme étant un ensemble de ressources (espace d’adressage, fichiers, périphériques…) que ses threads (fils d'exécution) partagent.
- Lorsqu’un processus est créé, un seul fil d’exécution (thread) est associé au processus ➔ thread principal exécutant la fonction main du programme du processus.
- Ce fil principal peut en créer d’autres.
- Chaque fil a:
  - un identificateur unique
  - une pile d'exécution
  - des registres (un compteur ordinal)
  - un état…

Qu'est ce qu'un thread ? (3)



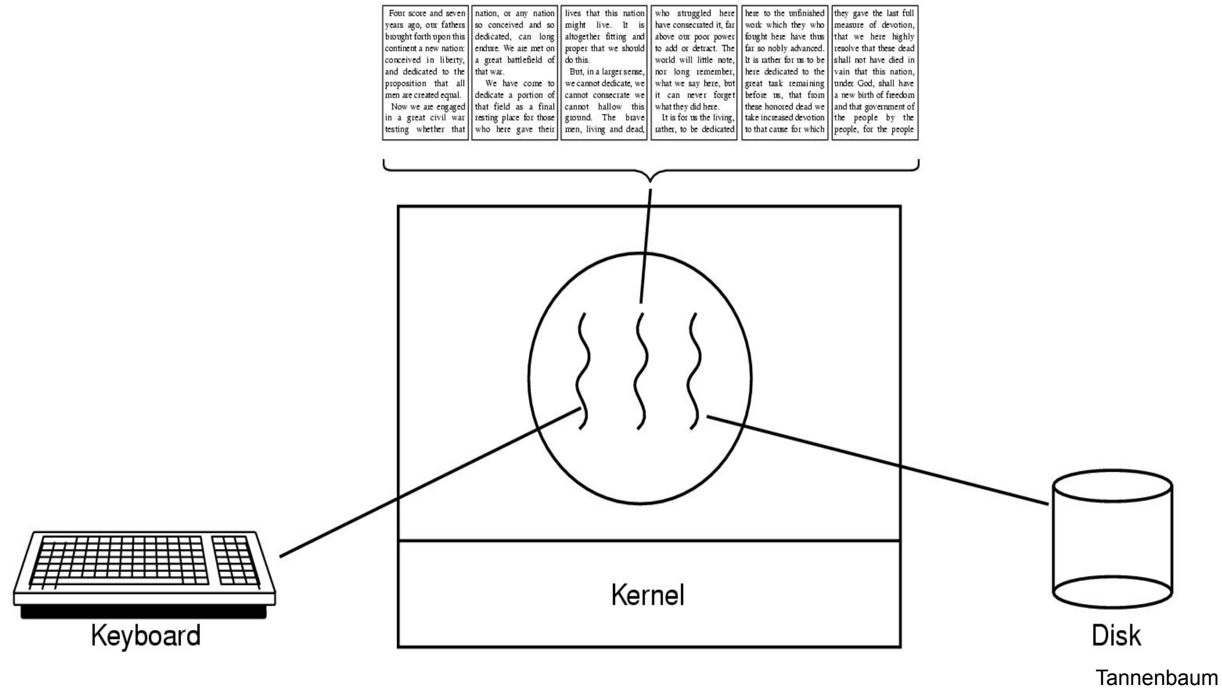
- Le multithreading permet l'exécution simultanée ou en pseudo-parallèle de plusieurs parties d'un même processus.

Qu’est ce qu’un thread ? (4)

Avantages des threads / processus

- Réactivité (le processus peut continuer à s’exécuter même si certaines de ses parties sont bloquées),
- Partage de ressources (facilite la coopération, améliore les performances),
- Économie d’espace mémoire et de temps. Il faut moins de temps pour :
  - créer, terminer un fil (sous Solaris, la création d’un processus est 30 fois plus lente que celle d’un thread),
  - Changer de contexte entre deux threads d’un même processus.

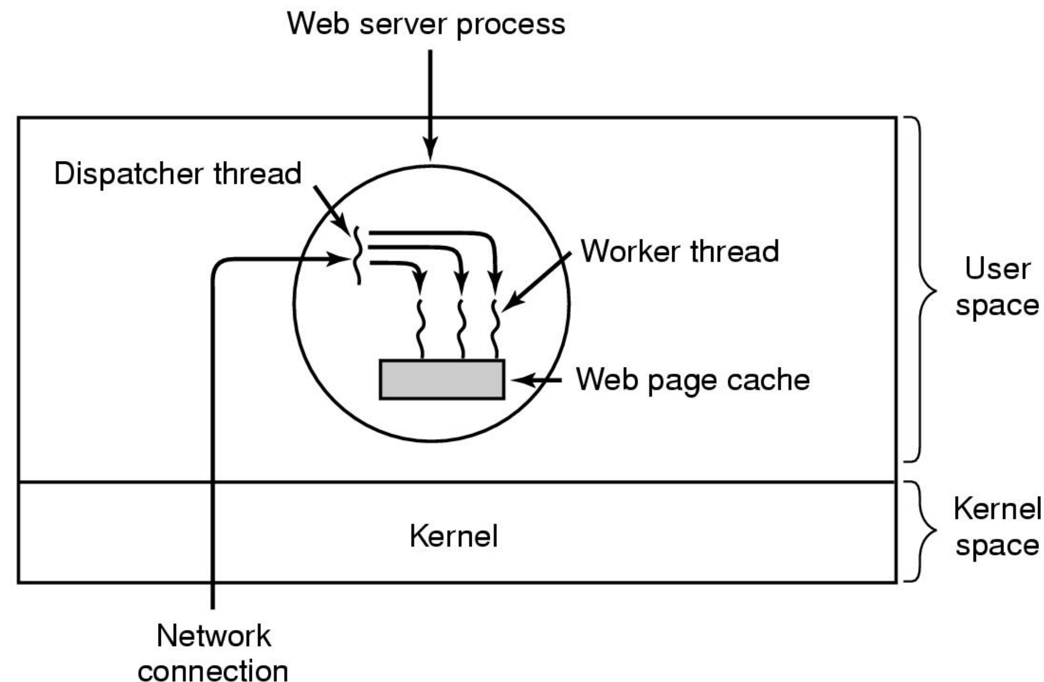
Usage des threads : Traitement de texte



Tannenbaum

- Un thread pour interagir avec l'utilisateur,
- Un thread pour reformater en arrière plan,
- Un thread pour sauvegarder périodiquement le document

Usage des threads (2) : Serveur Web




```
while(TRUE) {
  get_next_request(&buf);
  handoff_work(&buf);
}
```

```
while(TRUE) {
  wait_for_work(&buf);
  look_for_page_in_cache(&buf, &page);
  if(page_not_in_cache(&page)
    read_page_from_disk(&buf, &page);
  return_page(&page);
}
```

Tannenbaum

Threads POSIX (librairie pthread)

- L'objectif premier des Pthreads est la portabilité (disponibles sous Solaris, Linux, Windows XP…)

Appel	Description
pthread_create	Créer un nouveau fil d'exécution
pthread_exit	Terminer le fil d'exécution appelant
pthread_join	Attendre la fin d'un autre fil d'exécution
pthread_mutex_init	Créer un mutex
pthread_mutex_destroy	Détruire un mutex
pthread_mutex_lock	Verrouiller un mutex
pthread_mutex_unlock	Relâcher un mutex
pthread_cond_init	Créer une variable de condition
pthread_cond_destroy	Détruire une variable de condition
pthread_cond_wait	Attendre après une variable de condition
pthread_cond_signal	Signaler une variable de condition

Threads POSIX (2) : pthread_create

```
#include <pthread.h>
int pthread_create(
        pthread_t *tid,   // sert à récupérer le numéro d'identification du thread créé.
        const pthread_attr_t *attr,
        // sert à préciser les attributs du pthread (taille de la pile, priorité….)
        //attr = NULL pour les attributs par défaut
        void * (*func) (void*),  // func est la fonction à exécuter par le thread
        void *arg);    // arg est le paramètre de la fonction.
```

• **pthread_create** crée un thread. Elle retourne 0, en cas de succès et une valeur non nulle en cas d'erreur.

void pthread_join(pthread_t tid, void **status);

• **pthread_join** met l'appelant en attente de la fin du thread tid. Le paramètre **status** sert à récupérer l'état de terminaison du thread tid.

• **pthread_t pthread_self(void)** retourne le tid du thread appelant.

Threads POSIX (3) : pthread_exit

void pthread_exit(void * status);

- Termine l'exécution du thread appelant.

- Si le thread n'est pas détaché, le tid du thread et l'état de terminaison sont sauvegardés pour les communiquer au thread qui effectuera pthread_join sur tid.

- Un thread détaché (par la fonction **pthread_detach(pthread_t tid)**) a pour effet de le rendre indépendant (pas de valeur de retour attendue).

- Un thread peut annuler (terminer l'exécution) d'un autre thread du même processus en appelant **pthread_cancel**. Cependant, les ressources utilisées (fichiers, allocations dynamiques, verrous, etc) ne sont pas libérées.

- Il est possible de spécifier une ou plusieurs fonctions de nettoyage à exécuter à la terminaison du thread (**pthread_cleanup_push()** et **pthread_cleanup_pop()**).

Threads POSIX (4) : Exemple 1

```
// programme threads1.c : partage de variables avec accès concurrents
#include <unistd.h> //pour sleep
#include <stdio.h>  // pour printf
#include <pthread.h>

int glob=0;
pthread_spinlock_t *lock;
void* inc_dec(void * x)
{
	int i, pas = *(int *) x;
	for(i=0; i<100000; i++)
		glob = glob + pas ;
	printf("ici inc_dec(%d), glob = %d\n", pas, glob);
	pthread_exit(NULL);
}
```

Threads POSIX (5) : Exemple 1

```
// programme threads1.c : partage de variables avec accès concurrents
int main( )
{       pthread_t tid1, tid2;
        const int un=1, moinsun=-1;
        printf("ici main, glob = %d\n", glob);

        // creation de deux threads
        if ( pthread_create(&tid1, NULL, inc_dec, (void*) &un) != 0)    return -1;
        printf("ici main: creation du thread inc_dec(%d) avec succes\n",  th1);

        if ( pthread_create(&tid2, NULL, inc_dec, (void*) &moinsun) != 0)   return -1;
        printf("ici main: creation du thread inc_dec(%d) avec succes\n", th2);

        // attente de la fin des threads
        pthread_join(tid1,NULL);
        pthread_join(tid2,NULL);
        printf("ici main : fin des threads, glob = %d \n",glob);
        return 0;
}
```

Threads POSIX (6) : Exemple 1

Thread principal

Variable globale glob=0

pthread_create

pthread_create

inc_dec 1

inc_dec -1

Accès concurrents à glob

pthread_join
pthread_join

return 0

Threads POSIX (7) : Exemple 1

```
jupiter$ gcc threads1.c -o threads1 -pthread
jupiter$ ./threads1
ici main, glob = 0
ici main: creation du thread inc_dec(1) avec succes
ici main: creation du thread inc_dec(-1) avec succes
ici inc_dec(1), glob = 99980
ici inc_dec(-1), glob = 32024
ici main : fin des threads, glob = 32024


jupiter$ ./threads1
ici main, glob = 0
ici main: creation du thread inc_dec(1) avec succes
ici main: creation du thread inc_dec(-1) avec succes
ici inc_dec(-1), glob = -24116
ici inc_dec(1), glob = 100000
ici main : fin des threads, glob = 100000
jupiter$
```

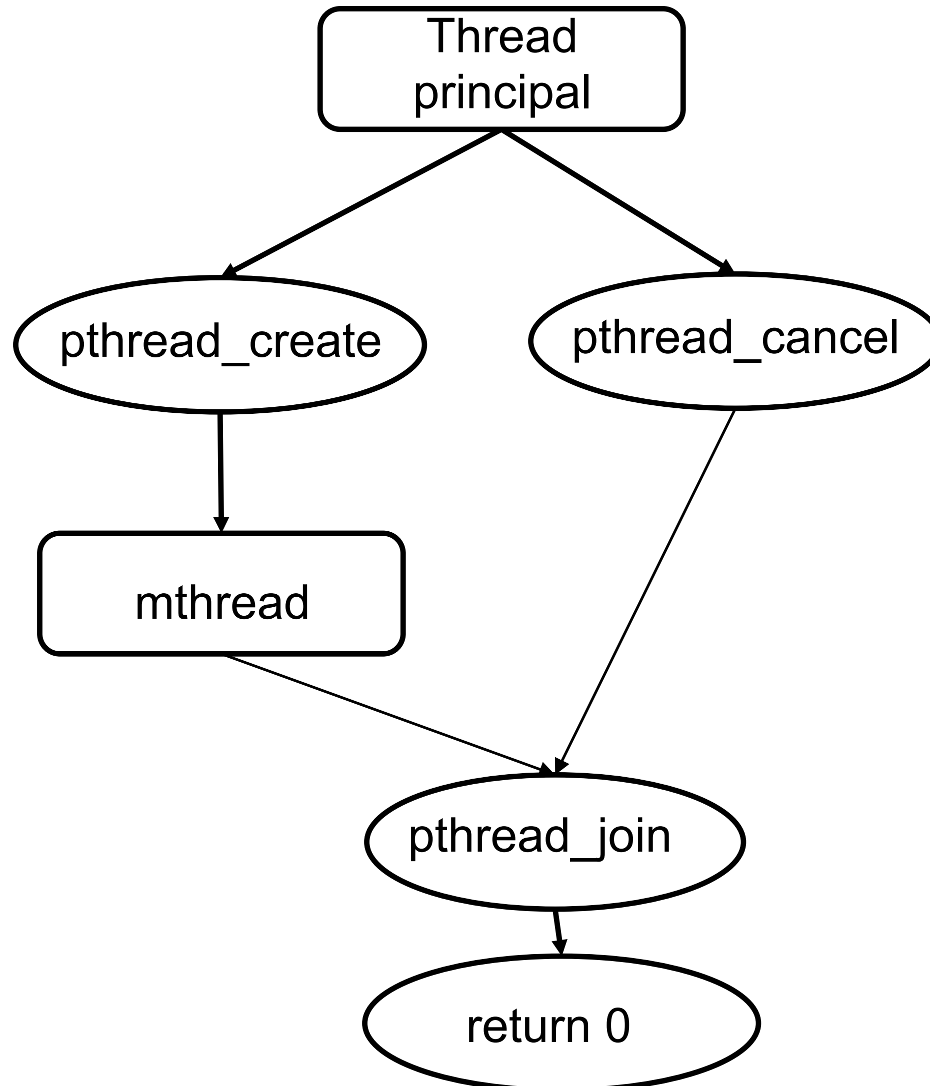
Threads POSIX (8) : Exemple 2

```
// threads2.c :  arrêt forcée d'un thread et macros de nettoyage
#include <pthread.h>
#include <unistd.h>
#include <stdio.h>

void mfclose (void *arg)  { printf("fclose\n"); fclose((FILE *)arg); }
void munlink(void * arg)  { printf("unlink\n"); unlink((char*)arg); }
void* mthread(void*inutilise);

int main()
{    pthread_t th;
     int* pstatus;
     if(pthread_create(&th,NULL,mthread,NULL) )
          { perror("Erreur dans la création du thread"); return 1;}
     sleep(2);
     pthread_cancel(th); // pour forcer la terminaison du thread
     pthread_join(th, (void**) &pstatus); // attendre la fin du thread

     if(pstatus == PTHREAD_CANCELED)  printf("Terminaison forcee du thread \n");
     else if(pstatus== NULL) printf("Terminaison normale du thread avec NULL\n");
       else printf("Terminaison normale du thread avec  %s.\n", pstatus);
    return 0;
}
```

Threads POSIX (9) : Exemple 2

```
void *mthread(void *inutilise)
{
   FILE* fichier; int i;
    setvbuf(stdout, (char *) NULL, _IONBF, 0);
    if ((fichier = fopen("sortie.txt","w")) == NULL)
             perror("Erreur dans l'ouverture du fichier");
    else {  // pour une terminaison propre
                pthread_cleanup_push(munlink, (void *) "sortie.txt");
                pthread_cleanup_push(mfclose, (void *) fichier);

                for (i=0; i<50; i++){
                        fprintf(fichier,"%d",i);
                        fprintf(stdout,"%d sur 50 ",i);
                        usleep(50000);
                }
    fprintf(stdout,"fin\n");
    pthread_cleanup_pop(1); // exécute fclose
    pthread_cleanup_pop(1); // exécute unlink
   }
   return (void*) "succes";
}
```

Threads POSIX (10) : Exemple 2



Arrêt forcée du thread mthread

Threads POSIX (11) : Exemple 2

```
jupiter$ $ gcc threads2.c -o threads2 -pthread
jupiter$ ./threads2
0 sur 50 1 sur 50 2 sur 50 3 sur 50 4 sur 50 5 sur 50 6 sur 50 7 sur 50 8
sur 50 9 sur 50 10 sur 50 11 sur 50 12 sur 50 13 sur 50 14 sur 50 15 sur
50 16 sur 50 17 sur 50 18 sur 50 19 sur 50 20 sur 50 21 sur 50 22 sur 50
23 sur 50 24 sur 50 25 sur 50 26 sur 50 27 sur 50 28 sur 50 29 sur 50 30
sur 50 31 sur 50 32 sur 50 33 sur 50 34 sur 50 35 sur 50 36 sur 50 37 sur
50 38 sur 50 39 sur 50 fclose
unlink
Terminaison forcee du thread
jupiter$
```

Threads POSIX (12) : Exemple 2

Résultat obtenu en commentant la ligne :
pthread_cancel(th); // pour forcer la terminaison du thread

```
jupiter$ $ gcc threads2.c -o threads2 -pthread
jupiter$ ./threads2
./threads2
0 sur 50 1 sur 50 2 sur 50 3 sur 50 4 sur 50 5 sur 50 6 sur 50 7 sur 50 8
sur 50 9 sur 50 10 sur 50 11 sur 50 12 sur 50 13 sur 50 14 sur 50 15 sur
50 16 sur 50 17 sur 50 18 sur 50 19 sur 50 20 sur 50 21 sur 50 22 sur 50
23 sur 50 24 sur 50 25 sur 50 26 sur 50 27 sur 50 28 sur 50 29 sur 50 30
sur 50 31 sur 50 32 sur 50 33 sur 50 34 sur 50 35 sur 50 36 sur 50 37 sur
50 38 sur 50 39 sur 50 40 sur 50 41 sur 50 42 sur 50 43 sur 50 44 sur 50
45 sur 50 46 sur 50 47 sur 50 48 sur 50 49 sur 50 fin
fclose
unlink
Terminaison normale du thread avec  succes.
jupiter$
```

Threads POSIX (13) : Exemple 3

```
// threads3.c  : (thread A || thread B) ; thread C
#include <pthread.h>
#include <unistd.h>
#include <stdio.h>
void afficher(char lettre)
{       int i,j;
        for (j=1; j<100; j++)
        {       for (i=1; i < 10000000; i++);
                printf("%c",lettre);
                fflush(stdout);
        }
}
Void *thread(void *plettre)
{       char lettre = *(char *) plettre;
        afficher(lettre);
        printf(“\n Fin du thread %c\n”,lettre);
       pthread_exit(NULL);
}
```

Threads POSIX (14) : Exemple 3

```
// threads3.c : (thread A || thread B) ; thread C
int main()
{
        pthread_t thA, thB, thC;

        printf("Creation des threads A et B\n");
        pthread_create(&thA, NULL, thread, (void *)"A");
        pthread_create(&thB, NULL, thread, (void*)"B");

        //attendre la fin des threads A et B
        printf("Le thread principal attend la fin des threads A et B\n");
        pthread_join(thA,NULL);
        pthread_join(thB,NULL);

        printf("\n Creation du thread C\n");
        pthread_create(&thC, NULL, thread, (void*)"C");
        printf("Le thread principal attend la fin du thread C\n");
        pthread_join(thC,NULL);
        _exit(0);
}
```

Threads POSIX (15) : Exemple 3

Thread principal

pthread_create

pthread_create

thread A

thread B

(thread A || thread B) ; thread C

_exit(0)

pthread_join

pthread_join
pthread_join

pthread_create

thread C

Threads POSIX (16) : Exemple 3

jupiter$ ./threads3
Creation des threads A et B
Le thread principal attend la fin des threads A et B
BAABABABABABABABABABABABABABABABABABABABABABABABABABABABABAB
ABABABABABABABABABABABABABABABABABABABABABABABABABABABABABAB
ABABABABABABABABABABABABABABABABABABABABABABABABABABABABABAB
ABABABABABABABABABABABABABABABABABABABA
Fin du thread A
B
Fin du thread B

Creation du thread C
Le thread principal attend la fin du thread C
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
Fin du thread C
jupiter$

Threads POSIX (17) : Exemple 3

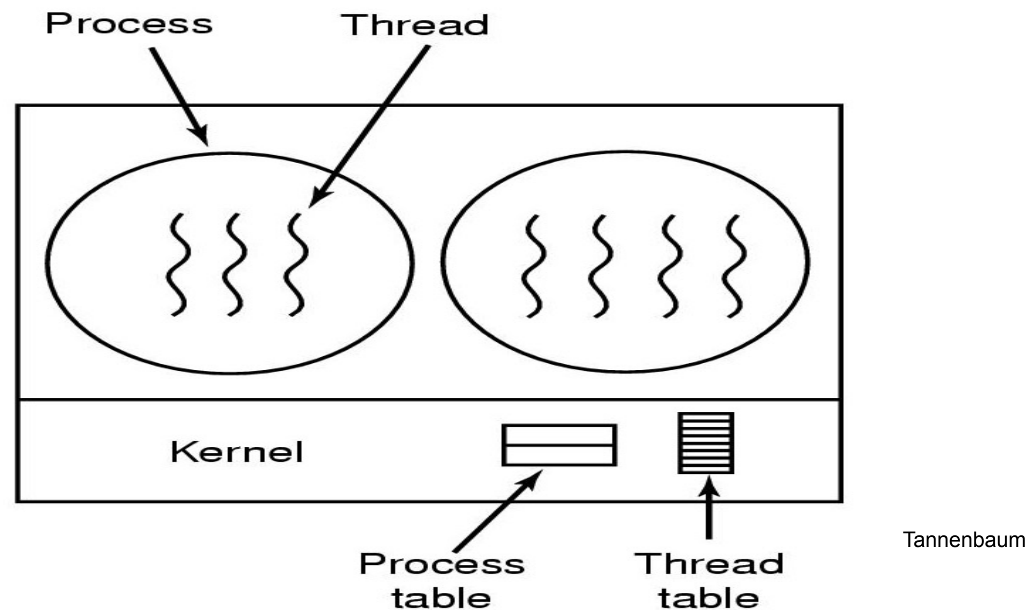
jupiter$ ./threads3
Creation des threads A et B
Le thread principal attend la fin des threads A et B
ABABBABABABABABABABABABABABABABABABABABABABABABABABABA
BABABABABABABABABABABABABABABABABABABABABABABABABABABA
BABABABABABABABABABABABABABABABABABABABABABABABABABABA
BABABABABABABABABABABABABABABABABAB
 Fin du thread B
A
 Fin du thread A

 Creation du thread C
Le thread principal attend la fin du thread C
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
 Fin du thread C
jupiter$

Threads noyau vs threads utilisateur

Threads noyau :

- Tous les systèmes d'exploitation de la famille Unix ainsi que Windows supportent les threads ➔ implémentation des threads au niveau noyau.



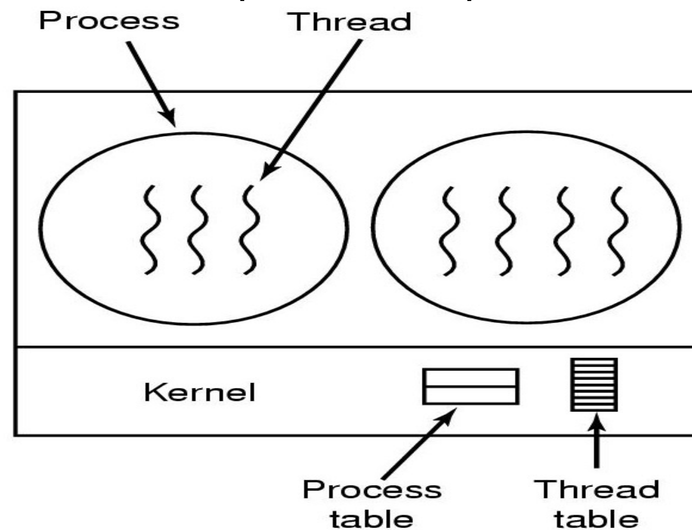
Tannenbaum

- Lorsqu'un processus est créé, il est composé d'un seul thread noyau qui exécute la fonction ***main***. Ce thread, appelé le thread principal du processus, peut ensuite créer d'autres threads noyau (ou encore des threads utilisateur).

Threads noyau vs threads utilisateur (2)

Threads noyau :

- Le noyau alloue du temps CPU aux threads noyau et non pas au processus
  - ➔ Si un thread noyau d'un processus se bloque suite à un appel système, un autre thread noyau du même processus peut être élu et donc exécuté.



Tannenbaum

- Sous Linux, l'appel système ***clone*** permet de créer, notamment, des threads noyau.

- Sous Linux, les threads Posix sont gérés au niveau noyau car la création d'un thread POSIX se traduit par la création d'un thread noyau en appelant ***clone***.

Threads noyau vs threads utilisateur (3)

Threads noyau : Sous Linux, threads POSIX = threads noyau

Exemple 4 :

```
// threads4.c : threads POSIX = threads noyau sous Linux
#include <unistd.h> //pour sleep
#include <stdio.h>  // pour printf
#include <pthread.h>
#include <sys/types.h>
#include <sys/syscall.h>

void* thread1(void * nused)
{        printf("ici thread1 %d du processus %d\n", syscall(SYS_gettid), getpid());
       pthread_exit(NULL);
}

void* thread2(void * nused)
{       printf("ici thread2 %d du processus %d\n",syscall(SYS_gettid), getpid());
       pthread_exit(NULL);
}
```

Threads noyau vs threads utilisateur (4)

Threads noyau : Sous Linux, threads POSIX = threads noyau

Exemple 4 :

```
// threads4.c : threads POSIX = threads noyau sous Linux
int main( )
{       pthread_t tid1, tid2;
        // creation de deux threads
        if ( pthread_create(&tid1, NULL, thread1, NULL) != 0)    return -1;
        if ( pthread_create(&tid2, NULL, thread2, NULL) != 0)   return -1;
        // attente de la fin des threads
        pthread_join(tid1,NULL);
        pthread_join(tid2,NULL);
        printf("ici thread principal %d du processus %d \n",syscall(SYS_gettid), getpid());
        return 0;
}
```

Threads noyau vs threads utilisateur (5)

Threads noyau : Sous Linux, threads POSIX = threads noyau

Exemple 4 :
jupiter$ gcc threads4.c -o threads4 -pthread
jupiter$ strace -f -e trace=clone -o output.txt ./threads4
ici thread2 **28583** du processus **28581**
ici thread1 **28582** du processus **28581**
ici thread principal **28581** du processus **28581**

jupiter$ cat output.txt
28581 clone(child_stack=0x7fd2b4fdffb0, flags=CLONE_VM|CLONE_FS|CLONE_FILES|
CLONE_SIGHAND|CLONE_THREAD|CLONE_SYSVSEM|CLONE_SETTLS|
CLONE_PARENT_SETTID|CLONE_CHILD_CLEARTID, parent_tidptr=0x7fd2b4fe09d0,
tls=0x7fd2b4fe0700, child_tidptr=0x7fd2b4fe09d0) **= 28582**
28581 clone(child_stack=0x7fd2b47defb0, flags=CLONE_VM|CLONE_FS|CLONE_FILES|
CLONE_SIGHAND|CLONE_THREAD|CLONE_SYSVSEM|CLONE_SETTLS|
CLONE_PARENT_SETTID|CLONE_CHILD_CLEARTID, parent_tidptr=0x7fd2b47df9d0,
tls=0x7fd2b47df700, child_tidptr=0x7fd2b47df9d0) **= 28583**
28583 +++ exited with 0 +++
28582 +++ exited with 0 +++
28581 +++ exited with 0 +++
jupiter$

Threads noyau vs threads utilisateur (6)

Threads noyau :

•Linux ne fait pas de distinction entre les processus et les fils d'exécution qui sont communément appelés tâches.

•Il supporte les appels système **fork** et **vfork** de **POSIX** mais il a aussi son propre appel système pour créer des tâches : **clone**.

•L'appel système **vfork** fonctionne comme **fork** sauf que le processus parent est bloqué jusqu'à ce que le processus créé se termine ou se transforme (exec()).

•L'appel système **clone** permet de spécifier les ressources à partager (espace d'adressage, fichiers, signaux, etc.) entre les tâches créatrice et créée.

Flag	Activé	Non activé
CLONE_VM	Partagé l'espace d'adressage	Ne pas partager
CLONE_FS	Partager umask, /, CWD	Ne pas partager
CLONE_FILES	Partager la table des descripteurs de fichier	Dupliquer cette table
CLONE_SIGHAND	Partager la table des gestionnaires de signaux	Dupliquer cette table

Threads utilisateur vs threads noyau (7)

Threads utilisateur :

- Les threads utilisateur sont implantés dans une librairie (au niveau utilisateur) qui fournit un support pour les gérer (exemple, la librairie GNU portable threads https://www.gnu.org/software/pth/)

- Lorsqu'un processus est créé, il a déjà un thread noyau qui exécute la fonction main du processus.

- Si ce thread noyau veut créer des threads utilisateur de la librairie pth, il doit d'abord appeler la fonction d'initialisation pth_init qui va, notamment, créer un ordonnanceur dont le rôle est de répartir le temps CPU alloué au thread noyau entre tous les threads utilisateur rattachés au thread noyau. Cet ordonnanceur est non préemptif.

➔ Deux niveaux d'ordonnancement :

- noyau (threads noyau) et
- utilisateur (threads utilisateur)

Threads utilisateur vs threads noyau (8)

Threads utilisateur :

- L'ordonnanceur noyau est appelé, si le thread en cours d'exécution effectue un appel système qui va bloquer, stopper ou terminer son exécution, libérer le processeur, ou encore suite à une interruption.

- L'ordonnanceur au niveau utilisateur est appelé, si le thread en cours d'exécution effectue un appel à une fonction de la librairie qui va bloquer, stopper ou terminer son exécution, libérer le processeur (ex. pth_sleep, pth_join).

Threads utilisateur vs threads noyau (9)

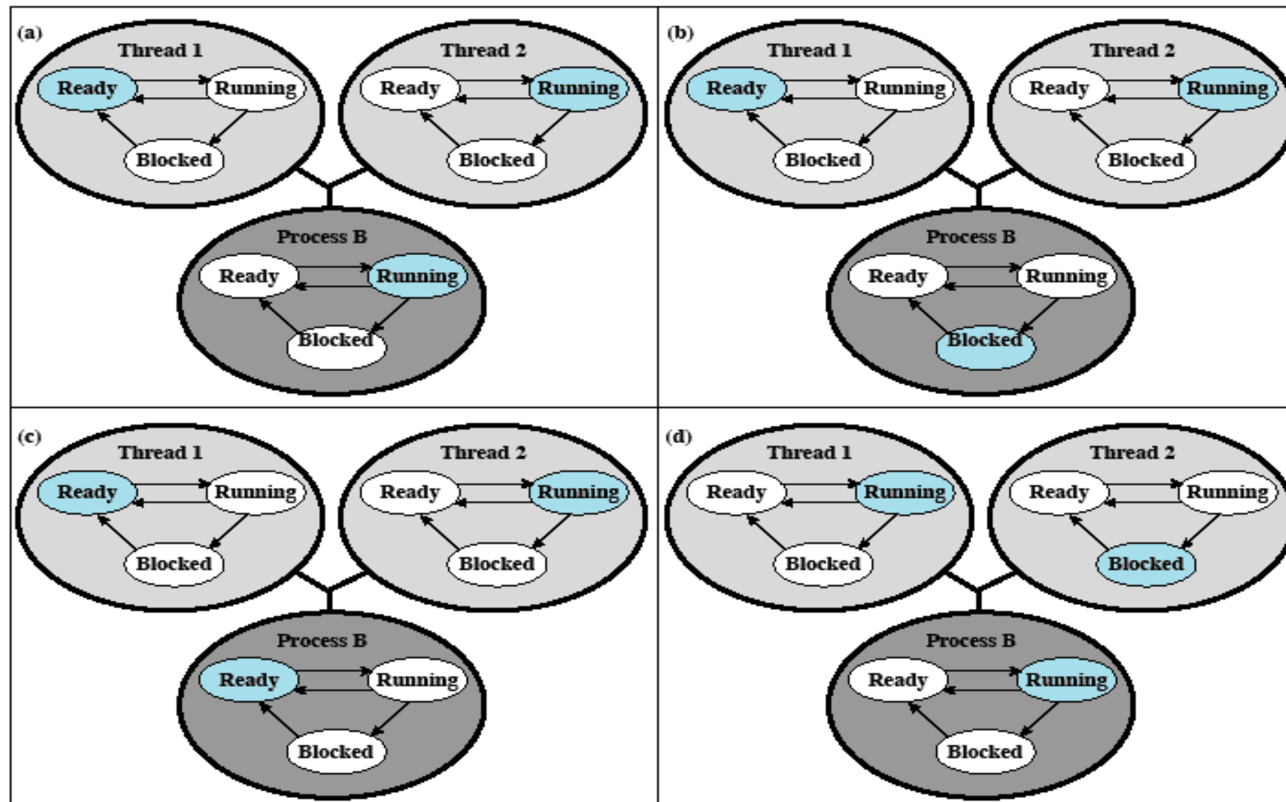


Figure 4.7 Examples of the Relationships Between User-Level Thread States and Process States

William Stallings (OS book, Seventh Edition)

Threads utilisateur vs threads noyau (10)

Threads utilisateur :

- Les threads utilisateur sont généralement créés, et gérés plus rapidement (en mode utilisateur) que les threads noyau.
- Ils sont plus portables.
- Par contre, si un processus a **n** threads noyau, il ne peut y avoir plus de **n** threads du processus qui s'exécutent en concurrence (ou en parallèle) ➔ pas intéressante pour des systèmes multiprocesseurs.
- Si un thread d'un processus se bloque suite à un appel système, tous les threads utilisateur rattachés au thread noyau ne pourront pas s'exécuter.

Threads utilisateur vs threads noyau (11) : Exemple 5

```
// thread5.c
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/wait.h>
static unsigned long long a = 0;
#define MAX 1000000000
void *count(void *arg) ;
int main( ) {
        int p, i;
        for (i = 0; i < 2; i++) {
        if ((p = fork()) < 0) return 1;
        if (p == 0) {   count(&a);
          printf("child %d a=%llu\n", getpid(), a);
          return 0;
         }
        }
        for (i = 0; i < 2; i++) { wait(NULL);}
        printf("parent %d a=%llu \n", getpid(), a);
        return 0;
}
```

```
void *count(void *arg) {
  volatile unsigned long long*var = (unsigned long long*) arg;
  volatile unsigned long long i;
  for (i = 0; i < MAX; i++)
        *var = *var + 1;
  return NULL;
}
```

```
jupiter$ gcc threads5.c -o threads5
jupiter$ time ./threads5
child 28921 a=1000000000
child 28922 a=1000000000
parent 28920 a=0

real    0m2,354s
user    0m4,683s
sys     0m0,003s
```

Threads utilisateur vs threads noyau (12) : Exemple 6

```
//threads6.c
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>
#include <sys/syscall.h>
```

```
void *count(void *arg) {
  volatile unsigned long long*var = (unsigned long long*) arg;
  volatile unsigned long long i;
  for (i = 0; i < MAX; i++)
       *var = *var + 1;
  return NULL;
}
```

```
static unsigned long long a = 0;
#define MAX 1000000000
void *count(void *arg) ;

int main( ) {
     pthread_t t1, t2;
     pthread_create(&t1, NULL, count, &a);
     pthread_create(&t2, NULL, count, &a);
     pthread_join(t1, NULL);
     pthread_join(t2, NULL);
     printf("pid=%d,  a=%llu\n", getpid(), a);
        return 0;
}
```

```
jupiter$ gcc thread6.c –o thread6 -lpthread
jupiter$ time ./threads6
pid=14646,  a=996904208

real      0m6,449s
user      0m12,778s
sys       0m0,001s
jupiter$
```

```
jupiter$ time ./threads6
pid=17277, a=1024717878

real      0m5,535s
User      0m10,059s
sys       0m0,005s
jupiter$
```

Threads utilisateur vs threads noyau (13) : Exemple 7

```
// threads7.c
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <pth.h>
static unsigned long long a = 0;
#define MAX 1000000000
void *count(void *arg);

int main(int argc, char **argv) {
	pth_init();
	pth_t t1, t2;
	t1 = pth_spawn(PTH_ATTR_DEFAULT, count, &a);
	t2 = pth_spawn(PTH_ATTR_DEFAULT, count, &a);
	pth_join(t1, NULL);
	pth_join(t2, NULL);
	printf("a=%llu\n", a);
	return EXIT_SUCCESS;
}
```

```
void *count(void *arg) {
  volatile unsigned long long*var = (unsigned long long*) arg;
  volatile unsigned long long i;
  for (i = 0; i < MAX; i++)
      *var = *var + 1;
  return NULL;
}
```

```
jupiter$ gcc thread7.c –o thread7 –lpth
jupiter$ time ./thread7
a=2000000000

real  0m4.679s
user  0m4.660s
sys   0m0.000s
jupiter$
```

Threads utilisateur vs threads noyau (13) : Exemple 7

```
// threads7.c
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <pth.h>
static unsigned long long a = 0;
#define MAX 1000000000
void *count(void *arg);

int main(int argc, char **argv) {
        pth_init();
        pth_t t1, t2;
        t1 = pth_spawn(PTH_ATTR_DEFAULT, count, &a);
        t2 = pth_spawn(PTH_ATTR_DEFAULT, count, &a);
        pth_join(t1, NULL);
        pth_join(t2, NULL);
        printf("a=%llu\n", a);
        return EXIT_SUCCESS;
}
```

```
void *count(void *arg) {
  volatile unsigned long long*var = (unsigned long long*) arg;
  volatile unsigned long long i;
  for (i = 0; i < MAX; i++)
        *var = *var + 1;
  return NULL;
}
```

```
jupiter$ strace -f -e trace=clone ./threads7
a=2000000000
+++ exited with 0 +++
```

```
jupiter$ gcc thread7.c –o thread7 –lpth
jupiter$ time ./thread7
a=2000000000

real  0m4.679s
user  0m4.660s
sys   0m0.000s
jupiter$
```

Processus, threads noyau, threads utilisateur (4) : Exemple 3

État bloqué

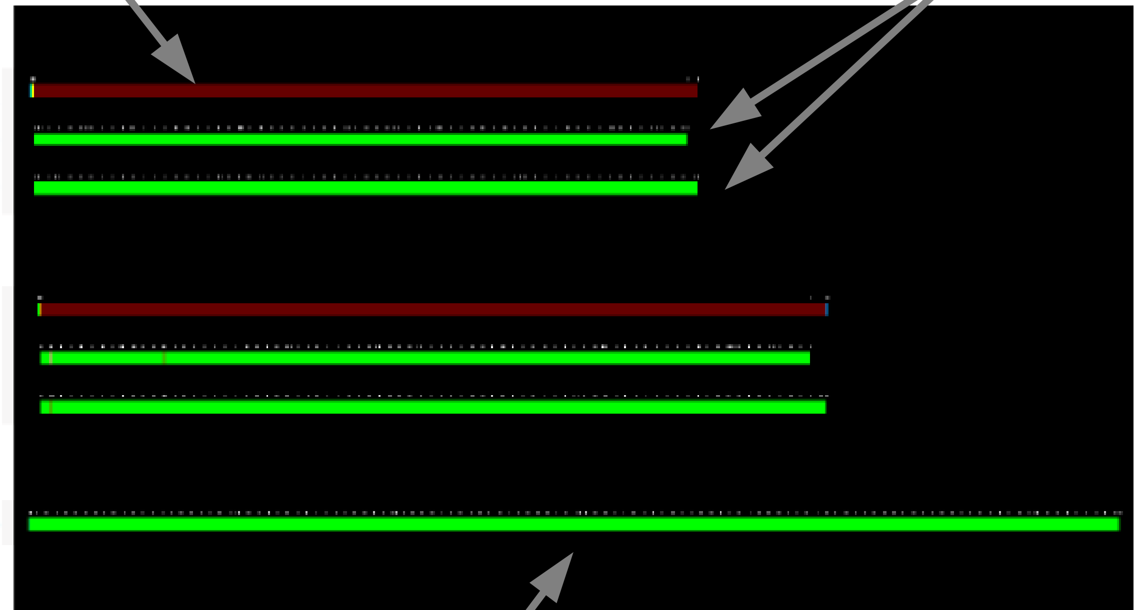
Exécution parallèle

Processus principal 17459 crée de 2 processus fils 17460 et 17462.

Processus (thread) principal 17476 crée de 2 threads noyau 17477 et 17478

Processus principal crée de 2 threads utilisateur.

PID	TGID	PPID
17459	17459	17452
17460	17460	17459
17461	17461	17459
17476	17476	17469
17477	17476	17476
17478	17476	17476
17493	17493	17486



Un seul fil d'exécution est créé, exécution sérielle

Lectures suggérées

2.2 THREADS et 10.3 PROCESSES IN LINUX
Modern operating Systems, 4nd edition, Andrew S. Tanenbaum, publié par Pearson Education, Prentice-Hall, 2016. **Livre disponible dans le dossier Slides Aut 2018 du site moodle du cours.**