Noyau d'un système d'exploitation
INF2610

Chapitre 3 : Threads (Fils d'exécution)

Département de génie informatique et génie logiciel

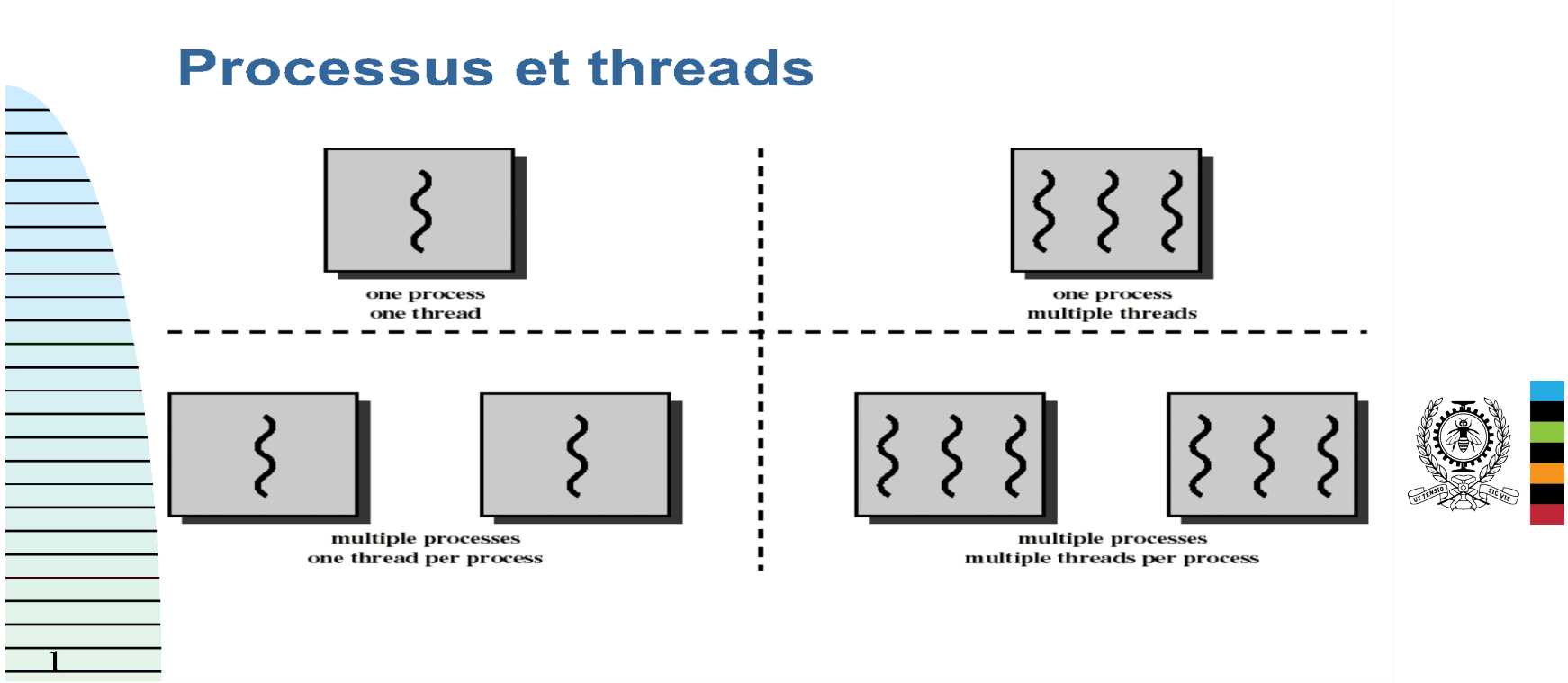
POLYTECHNIQUE MONTRÉAL
AFFILIÉE À L'UNIVERSITÉ DE MONTRÉAL

Chapitre 3 - Threads (Fils d'exécution)

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

Qu'est ce qu'un thread ?

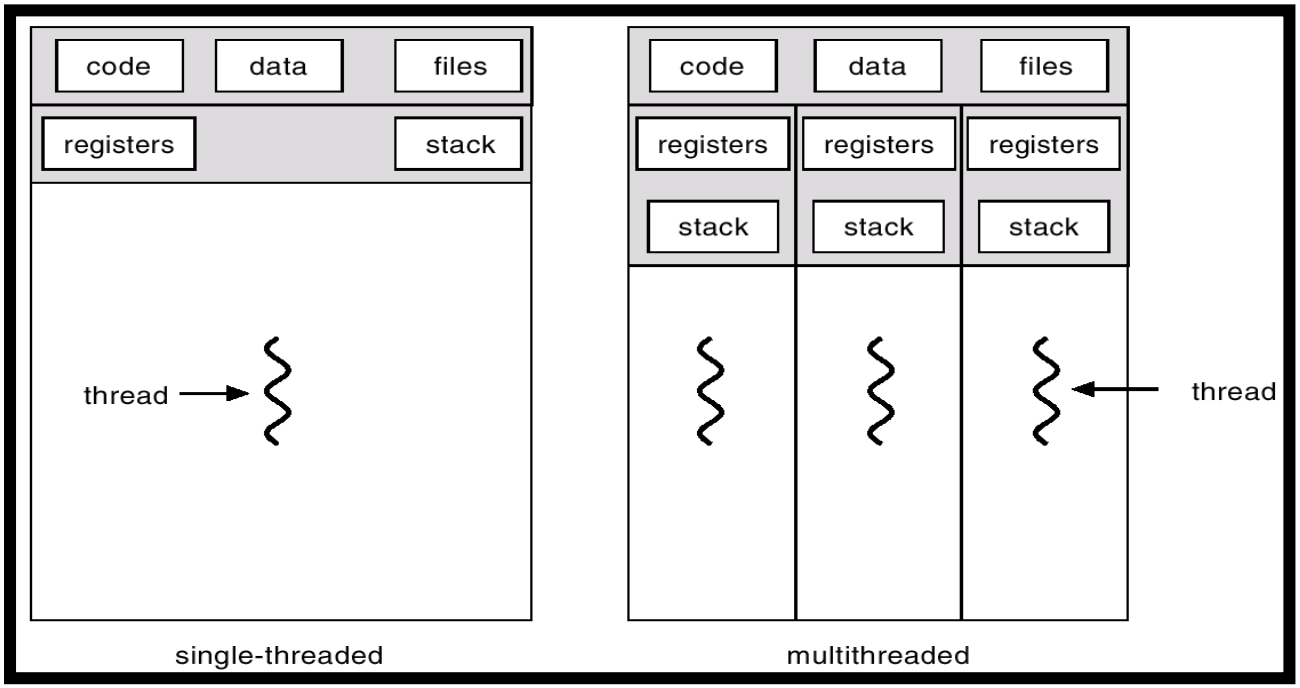
- Le modèle processus décrit précédemment est un programme qui s'exécute selon un chemin unique (compteur ordinal). On dit qu'il a un fil d'exécution ou flot de contrôle unique (single thread).
- De nombreux systèmes d'exploitation modernes offrent la possibilité d'associer à un même processus plusieurs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

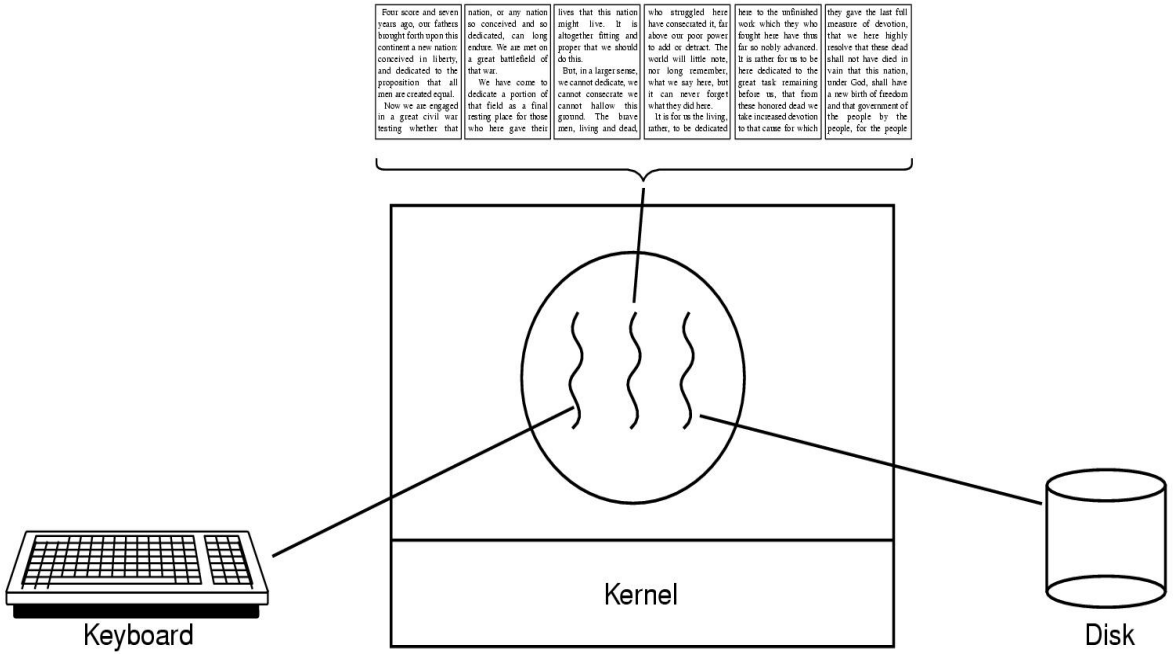
Avantages des threads / processus

- 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.

Comment mesurer le temps d'exécution d'un appel système ?

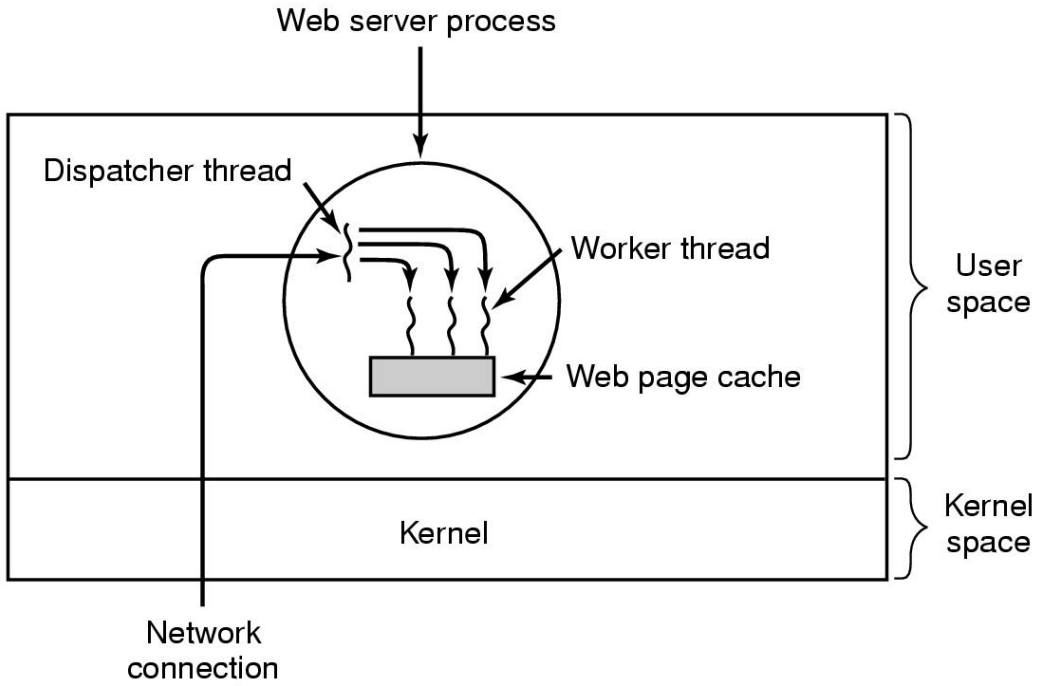
Richard L. Oliver and Patricia J. Teller and Ward P. McGregor, "**ACCURATE MEASUREMENT OF SYSTEM CALL SERVICE TIMES FOR TRACE-DRIVEN SIMULATION OF MEMORY HIERARCHY DESIGNS**".

Usage des threads : Traitement de texte



- 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);
}
```

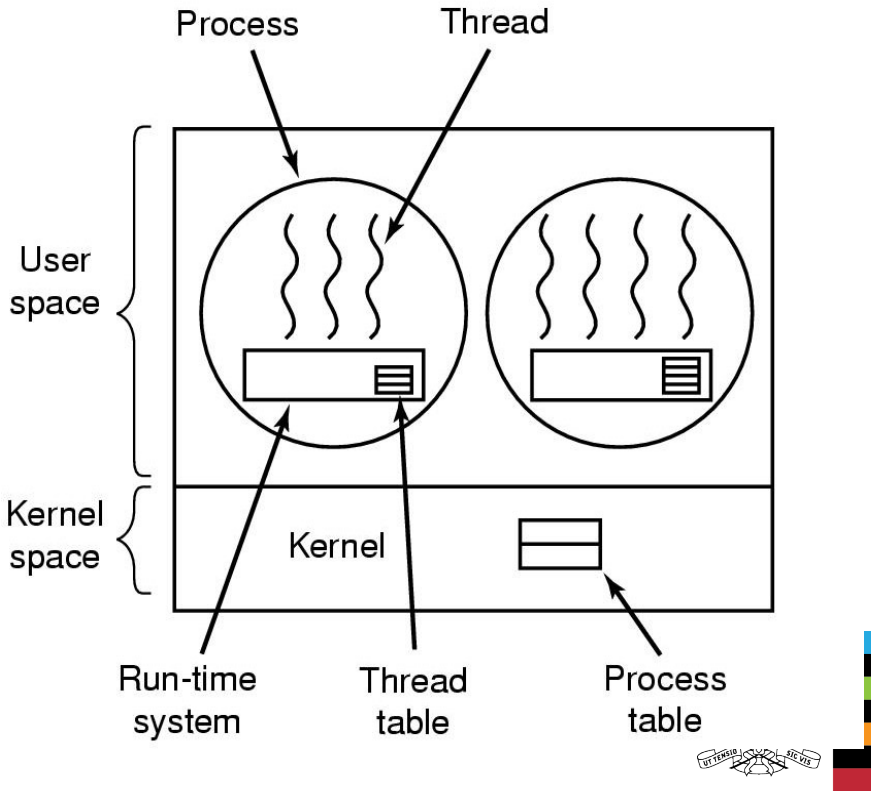
```
while(TRUE) {
  wait_for_work(&buf);
  look_for_page_in_cache(&buf, &page);
  if(page_not_in_cache(&page)
    read_page_from_disk(&buf, &page);
  return_page(&page);
}
```

Threads utilisateur

- Les threads utilisateur sont implantés dans une bibliothèque (au niveau utilisateur) qui fournit un support pour les gérer. Ils ne sont pas gérés par le noyau.

- Le noyau gère, par exemple, les processus et ne se préoccupe pas de l'existence des threads utilisateur (modèle plusieurs-à-un).

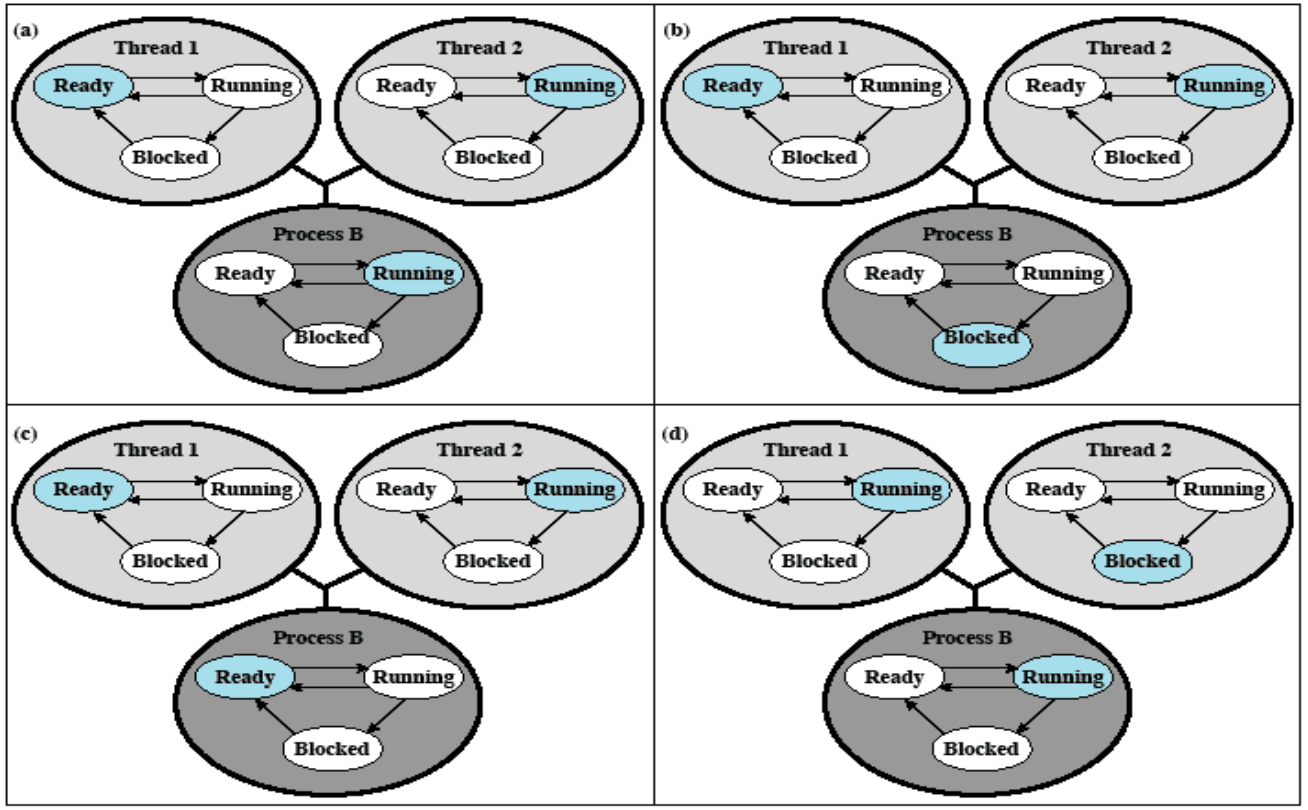
- Lorsque le noyau alloue le processeur à un processus, le temps d'allocation du processeur est réparti entre les différents threads du processus (cette répartition n'est pas gérée par le noyau).



=> Deux niveaux d'ordonnancement :

- Processus (par le noyau)
- threads de chaque processus (par la bibliothèque).

Threads utilisateur (2)



Colored state
is current state

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

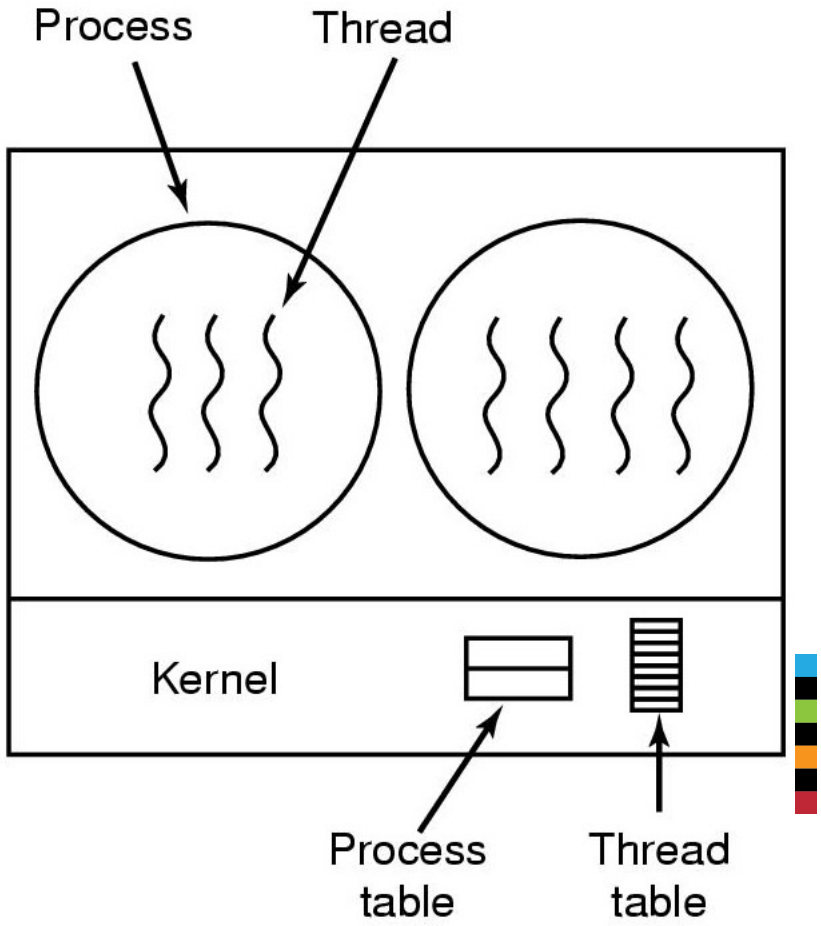
William Stallings (OS book, Seventh Edition)

Threads utilisateur (3)

- Les threads utilisateur sont généralement créés, et gérés rapidement (en mode utilisateur).
- Ils facilitent la portabilité (comparativement aux autres implémentations)
- Inconvénients :
  - À tout instant, au plus un fil par processus est en cours d'exécution. Cette implémentation n'est pas intéressante pour des systèmes multiprocesseurs.
  - Si un thread d'un processus se bloque, tout le processus est bloqué. Pour pallier cet inconvénient, certaines librairies transforment les appels système bloquants en appels système non bloquants.

Threads noyau

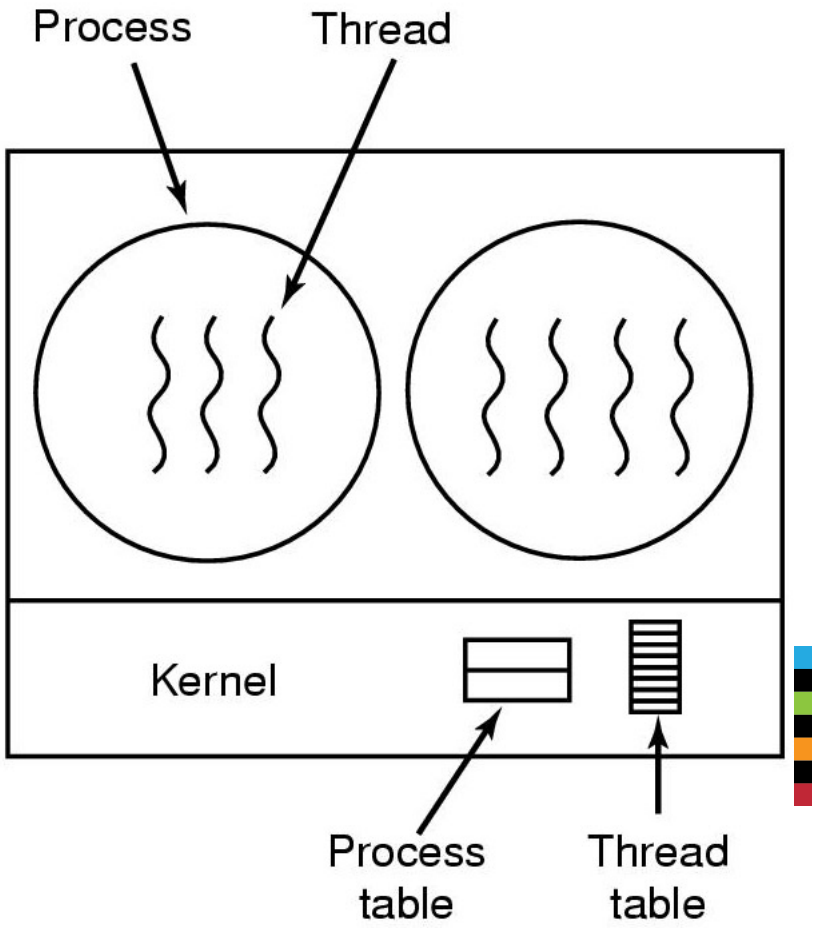
- Les threads noyau sont directement supportés par le système d'exploitation.
- Le système d'exploitation se charge de leur gestion. Du temps CPU est alloué à chaque thread.
- Si un thread d'un processus est bloqué, un autre thread du même processus peut être élu par le noyau.
- Cette implémentation est plus intéressante pour les systèmes multiprocesseurs.
- Un processus peut ajuster les niveaux de priorité de ses threads. Par exemple, un processus peut améliorer son interactivité en assignant:
  - une forte priorité à un thread qui traite les requêtes des utilisateurs et
  - une plus faible priorité aux autres.



Threads noyau (2)

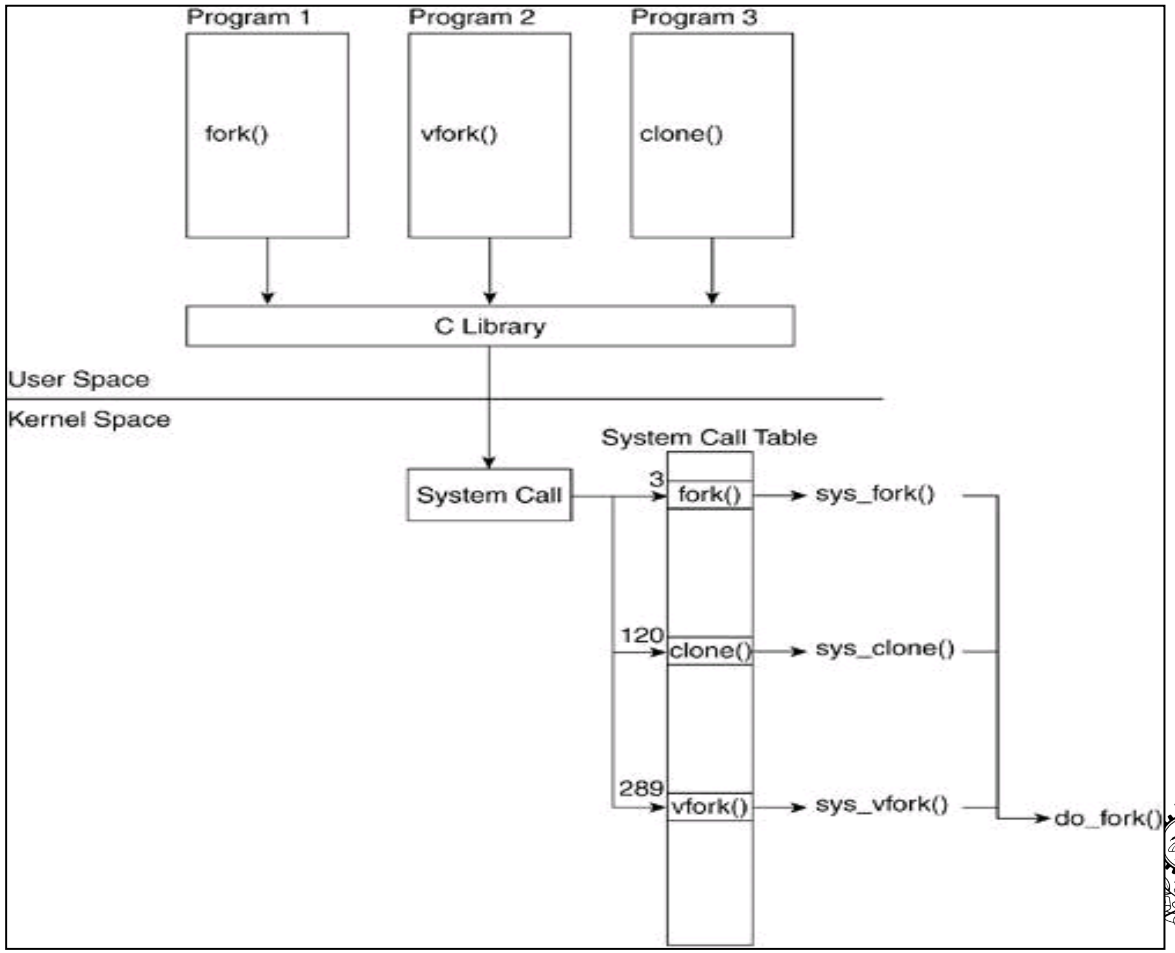
Inconvénients :

- Performance (gestion plus coûteuse)
- Les programmes utilisant les fils d'exécution noyau sont moins portables que ceux qui utilisent des fils en mode utilisateur.



Threads noyau (3) : Linux

- Linux ne fait pas de distinction entre les processus et les fils d'exécution qui sont communément appelés tâches.
- Il implémente le modèle multiflot un-à-un.
- La création de tâches est réalisée au moyen de l'appel système fork, vfork ou clone.



Threads noyau (4) : Linux

- vfork() fonctionne comme fork() sauf que le processus parent est bloqué jusqu'à ce que le processus créé appelle exit() ou exec().

- Clone() permet de spécifier les ressources à partager (espace d'adressage, fichiers, signaux..) entre les tâches créatrice et créée.

Flag	Activé	Non activé
CLONE_VM	Nouveau fil	Nouveau processus
CLONE_FS	Partager umask, /, CWD	Ne pas partager
CLONE_FILES	Partager descripteurs de fichiers	Copier les fd
CLONE_SIGHAND	Partager réaction aux signaux	Copier la table
CLONE_PID	Conserver le PID	thread obtient son propre PID

Threads POSIX

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

Appel	Description
pthread_create	Créer un nouveau fil d'exécution
pthread_exit	Terminer le fil appelant
pthread_join	Attendre la fin d'un autre fil
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 condition
pthread_cond_destroy	Détruire une condition
pthread_cond_wait	Attendre après une condition
pthread_cond_signal	Signaler une condition

Threads POSIX (2) : pthread_create

- **int pthread_create(**

  pthread_t *tid,
  // sert à récupérer le TID du pthread 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*),
  // est la fonction à exécuter par le pthread

  void *arg);
  //le paramètre de la fonction.

- L'appel retourne 0, en cas de succès. En cas d'erreur, il retourne une valeur non nulle identifiant l'erreur.

Threads POSIX (3) : pthread_join et pthread_self

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

- Attend la fin d'un thread. L'équivalent de waitpid des processus sauf qu'on doit spécifier le tid du thread à attendre.
- status sert à récupérer la valeur de retour et l'état de terminaison.

pthread_t pthread_self(void);

- Retourne le TID du thread appelant.

Threads POSIX (4) : pthread_exit

void pthread_exit(void * status);

- Termine l'exécution du thread.
- 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.
- Un thread détaché (par la fonction **pthread_detach(pthread_t tid)**) a pour effet de le rendre indépendant de celui qui l'a créé (pas de valeur de retour attendue).
- Un thread peut annuler ou terminer l'exécution d'un autre 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 (5) : Exemple 1

```
// programme exemple2.c; partage de variables
#include <unistd.h> //pour sleep
#include <pthread.h>
#include <stdio.h>
int glob=0;
void* decrement(void * x)
{
      glob = glob - 1 ;
       printf("ici decrement[%d], glob = %d\n", *((int*) x), glob);
       pthread_exit(NULL);
}

void* increment (void * x)
{
       int *num=x;
       glob = glob + 1;
       printf("ici increment[%d], glob = %d\n", *num, glob);
       pthread_exit(NULL);
}
```

Threads POSIX (6) : Exemple 1

```
int main( )
{       pthread_t tid1, tid2;
        int th1=1, th2=2;
        printf("ici main[%d], glob = %d\n", getpid(),glob);
        //creation de deux threads pour increment et decrement
        if ( pthread_create(&tid1, NULL, increment, &th1) != 0)    return -1;
        printf("ici main: creation du thread[%d] avec succes\n",  th1);
        if ( pthread_create(&tid2, NULL, decrement, &th2) != 0)   return -1;
        printf("ici main: creation du thread [%d] avec succes\n", th2);
        pthread_join(tid1,NULL); // attendre la fin du thread tid1
        pthread_join(tid2,NULL);
        printf("ici main : fin des threads, glob = %d \n",glob);
        return 0;
}
```

```
-bash-3.2$ gcc exemple2.c -lpthread –o exemple
-bash-3.2$ ./exemple
ici main[26533], glob = 0
ici main: creation du thread[1] avec succes
ici main: creation du thread [2] avec succes
ici increment[1], glob = 1
ici decrement[2], glob = 0
ici main : fin des threads, glob = 0
```

Threads POSIX (7) : Exemple 1'

Les instructions glob = glob-1; et glob = glob+1; sont remplacées par respectivement :

```
int r = glob;
for(int j=0; j<1000000; j++);
 r= r – 1; glob = r;
```

et

```
int r = glob;
for(int j=0; j<1000000; j++);
r= r + 1; glob = r;
```

```
-bash-3.2$ gcc exemple2_1.c -lpthread –o exemple2_1
-bash-3.2$ ./exemple2_1
ici main[46618], glob = 0
ici main: creation du thread[1] avec succes
ici main: creation du thread [2] avec succes
ici increment[1], glob = 1
ici decrement[2], glob = -1
ici main : fin des threads, glob = -1
```

Threads POSIX (8) : Exemple 2

```
// exemple4.c Annulation d'un thread
#include <pthread.h>
#include <unistd.h>
#include <stdio.h>

void _fclose (void *arg)
{    printf("fclose\n"); fclose((FILE *)arg);}

void _unlink(void * arg)
{    printf("unlink\n"); unlink((char*)arg);}

void*mon_thread(void*inutilise);

int main()
{   pthread_t th;
    if(pthread_create(&th,NULL,mon_thread,NULL) )
          perror("Erreur dans la création du thread");
    sleep(2);
    pthread_cancel(th); // annulation
    pthread_join(th,NULL);
    return 0;
}
```

Threads POSIX (9) : Exemple 2

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

            for (i=0; i<1000; i++){
                  fprintf(un_fichier,"%d",i);
                  fprintf(stdout,"%d sur 1000 ",i);
                  usleep(50000);
            }
    fprintf(stdout,"fin\n");
    pthread_cleanup_pop(1); // exécute fclose
    pthread_cleanup_pop(1); // exécute unlink
   }
   return null;
}
```

Threads POSIX (10) : Exemple 2

```
-bash-3.2$ gcc exemple4.c -lpthread –o exemple4
-bash-3.2$ ./exemple4
0 sur 1000 1 sur 1000 2 sur 1000 3 sur 1000 4 sur 1000 5 sur
1000 6 sur 1000 7 sur 1000 8 sur 1000 9 sur 1000 10 sur 1000
11 sur 1000 12 sur 1000 13 sur 1000 14 sur 1000 15 sur 1000
16 sur 1000 17 sur 1000 18 sur 1000 19 sur 1000 20 sur 1000
21 sur 1000 22 sur 1000 23 sur 1000 24 sur 1000 25 sur 1000
26 sur 1000 27 sur 1000 28 sur 1000 29 sur 1000 30 sur 1000
31 sur 1000 32 sur 1000 33 sur 1000 34 sur 1000 35 sur 1000
36 sur 1000 37 sur 1000 38 sur 1000 39 sur 1000 fclose
unlink
```

Processus, threads noyau (pthreads), threads utilisateur (pth) : Exemple 3

```
#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("a=%llu \n", 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$ g++ thread_process.cpp –o thread_process
jupiter$ time ./thread_process
child 10629 a=1000000000
child 10630 a=1000000000
a=0

real  0m2.507s
user 0m4.900s
sys   0m0.040s
jupiter$
```

Processus, threads noyau, threads utilisateur (2) : Exemple 3

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>
static unsigned long long a = 0;
#define MAX 1000000000
void *count(void *arg) ;

int main( ) {
        int p, i;
        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;
}
```

```
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$ g++ thread_pthread.cpp –o thread_pthread
jupiter$ time ./thread_pthread
pid=10686 a=987341287

real  0m3.791s
user 0m7.240s
sys   0m0.040s
jupiter$
```

```
jupiter$ time ./thread_pthread
pid=10695 a=1019958670

real  0m3.740s
user 0m7.423s
sys   0m0.040s
jupiter$
```

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

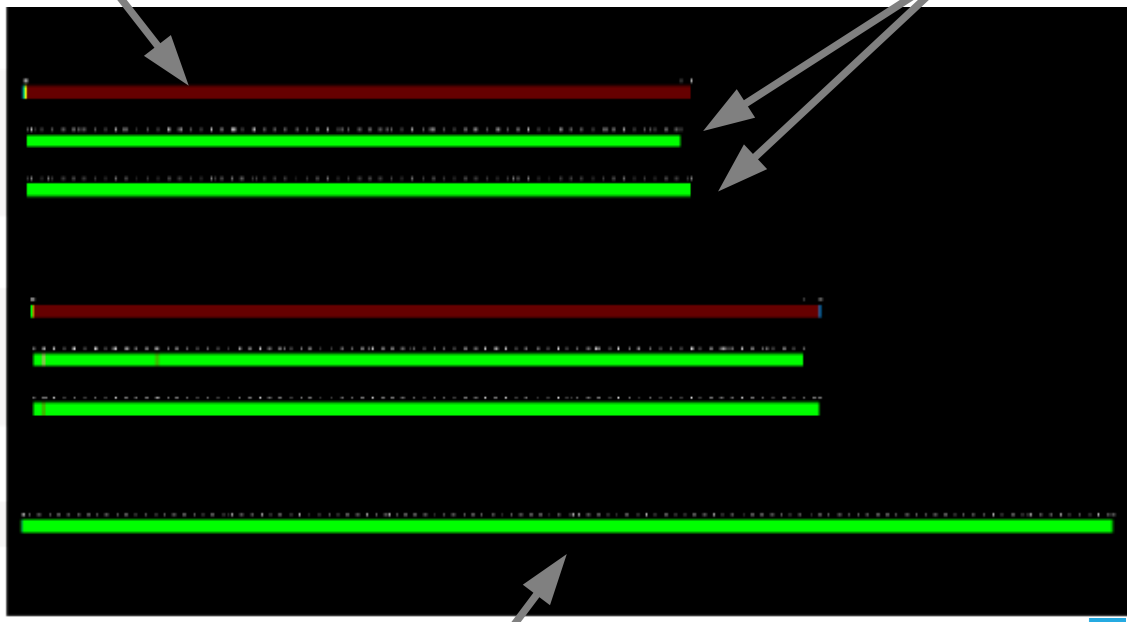
```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <pth.h>
static unsigned long long a = 0ULL;
#define MAX 1000000000
void *count(void *arg);

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$ g++ thread_userspace.cpp –o thread_userspace –lpth
jupiter$ time ./thread_userspace
a=2000000000

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

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

Commande	PID	TGID	PPID
./module2/thread_process	17459	17459	17452
./module2/thread_process	17460	17460	17459
./module2/thread_process	17461	17461	17459
./module2/thread_pthread	17476	17476	17469
./module2/thread_pthread	17477	17476	17476
./module2/thread_pthread	17478	17476	17476
/module2/thread_userspace	17493	17493	17486



Lectures suggérées

- Notes de cours: Chapitre 4
(http://www.groupes.polymtl.ca/inf2610/documentation/notes/chap4.pdf)

- Chapitre 4 (pp 55- 82)
M. Mitchell, J. Oldham, A. Samuel - Programmation Avancée sous Linux-
Traduction : Sébastien Le Ray (2001) **Livre disponible dans le dossier Slides Automne 2017 du site moodle du cours.**

Threads POSIX : Exemple 4

Annexe

```
// exemple1.c  partage de descripteurs de fichiers
#define _REENTRANT
#include <pthread.h>
#include <unistd.h>
#include <stdio.h>
void afficher(int n, char lettre)
{           int i,j;
            for (j=1; j<n; j++)
            {           for (i=1; i < 10000000; i++);
                        printf("%c",lettre);
                        fflush(stdout);
            }
}

void *threadA(void *inutilise)
{           afficher(100,'A');
            printf("\n Fin du thread A\n");
            fflush(stdout);
            pthread_exit(NULL);
}
```

Threads POSIX : Exemple 4

Annexe

```
void *threadC(void *inutilise)
{
        afficher(150,'C');
        printf("\n Fin du thread C\n");
        fflush(stdout);
        pthread_exit(NULL);
}

void *threadB(void *inutilise)
{
        pthread_t thC;
        pthread_create(&thC, NULL, threadC, NULL);
        afficher(100,'B');
        printf("\n Le thread B attend la fin du thread C\n");
        pthread_join(thC,NULL);
        printf("\n Fin du thread B\n");
        fflush(stdout);
        pthread_exit(NULL);
}
```

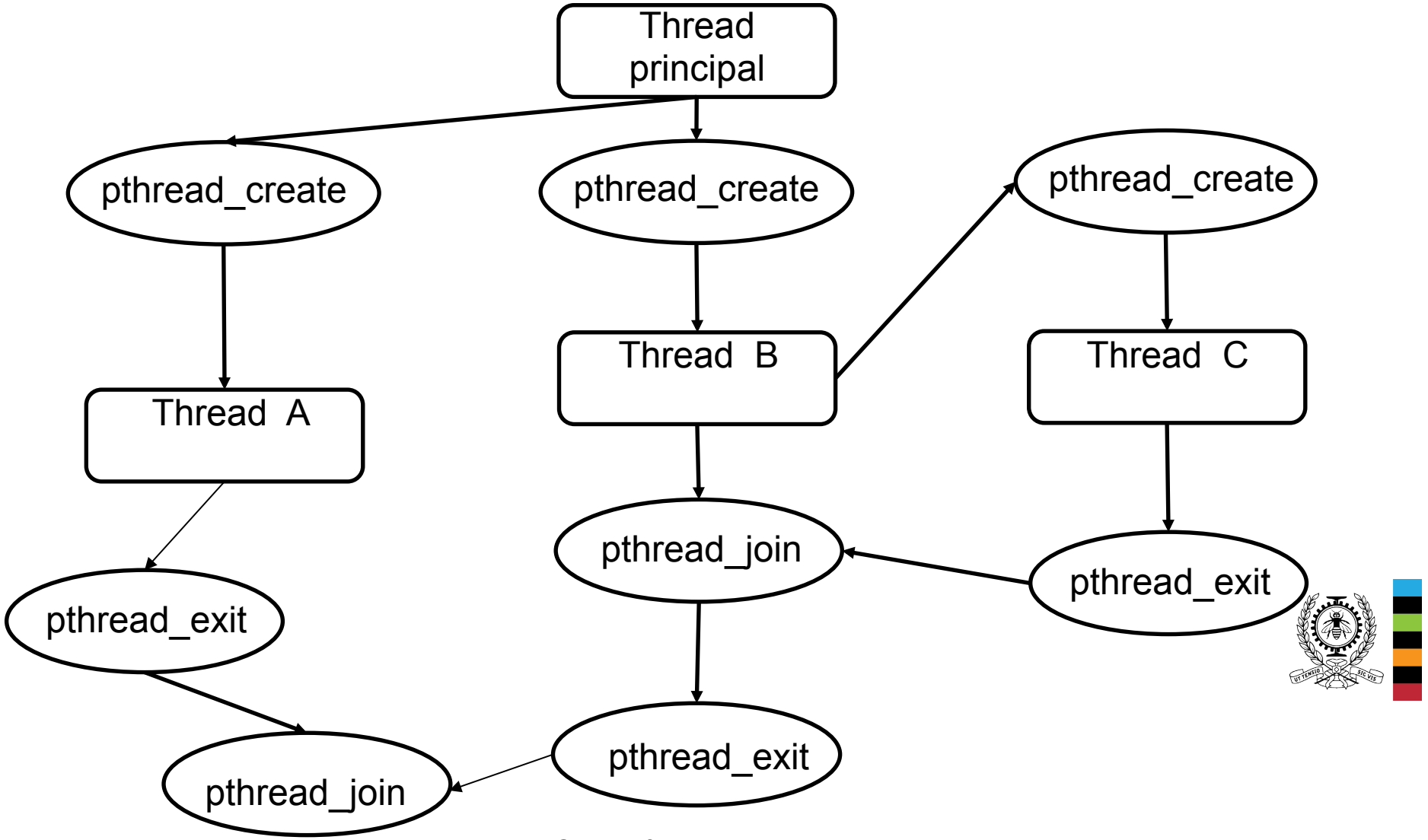
Threads POSIX : Exemple 4

Annexe

```
int main()
{
        int i;

        pthread_t thA, thB;

        printf("Creation du thread A");

        pthread_create(&thA, NULL, threadA, NULL);
        pthread_create(&thB, NULL, threadB, NULL);

        sleep(1); // simule un traitement
        printf("Le thread principal attend que les autres se terminent\n");

        //attendre la fin des threads
        pthread_join(thA,NULL);
        pthread_join(thB,NULL);

        exit(0);
}
```

Threads POSIX : Exemple 4

Annexe



Threads POSIX : Exemple 4

Annexe

```
-bash-3.2$ g++ exemple4.cpp -lpthread –o exemple4
-bash-3.2$ ./exemple4
Creation du thread
AACBACBACBACBACBACBACBACBACBABCABCABCABCABCABCABCA
BCABCABCABCABCABCABCABCACBACBACBACBLe thread principal
attend que les autres se terminent
ACBACBACBACBACBACBACBACBACBACBACBACBACBACBACBACBAC
BACBACBACBACBACBACBACBACBACBACBACBACBACBACBACBACBA
CBACBACBACBACBACBACBACBACBACBACBACBACBACBACBACBACB
ACBACBACBACBACBACBACBACBACBACBACBACBACBACBACBACBAC
BACBACBACBA
 Fin du thread A
CB
 Le thread B attend la fin du thread C
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
 Fin du thread C

 Fin du thread B
```

Threads POSIX : Exemple 4

Annexe

```
-bash-3.2$ g++ exemple4.cpp -lpthread –o exemple4
-bash-3.2$ ./exemple4
Creation du thread
AACBACBACBACBACBACBACBACBACBACBACBACBACBACBACBACBACBACBA
CBACBACBACBACBACBACBACBACBACBACBACBLe thread principal
attend que les autres se terminent
ACBACBACBACBACBACBACBACBACBACBACBACBACBACBACBACBACBACBAC
BACBACBACBACBACBACBACBACBCABCABCABCABCABCABCABCABCABC
ABCABCABCABCABCABCABCABCABCABCABCABCABCABCABCABCABCABCAB
CABCABCABCABCABCABCABCABCABCABCABCABCABCABCABCABCABCABCA
BCABCABCABCA
 Fin du thread A
B
 Le thread B attend la fin du thread C
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
 Fin du thread C

 Fin du thread B
-bash-3.2$
```

Threads POSIX : Exemple 5

Annexe

```
// Programme exemple5.cpp - passage de paramètres à un thread
#define _REENTRANT
#include <iostream>
#include <cstdio>
#include <cstdlib>
#include <pthread.h>
#include <sys/types.h>
#include <sys/time.h>
#include <unistd.h>
using namespace std;

int MAX=5;
inline int gen_alea( int, int );
void *afficher( void * );
```

```
// Afficher, un nombre (aléatoire) de fois,  un mot
 void * afficher(void *mot)
{
    int  *nombre = new int;
    *nombre = gen_alea(2,6);
    cout << (char *)mot << "\t sera affiche " <<
                *nombre << "fois." << endl;
     for (int i=0; i < *nombre; ++i){
        sleep(1);
        cout << (char *) mot << "  ";  }
   return nombre;
}

int gen_alea( int LOW, int HIGH )
{       time_t seconds;
       time(&seconds);
       srand((unsigned int) seconds);
       return rand() % (HIGH - LOW + 1) + LOW;
}
```

Threads POSIX : Exemple 5

Annexe

```
int main(int argc, char *argv[])
{
    pthread_t thread_id[MAX];
    int  *retour;
    setvbuf(stdout, (char *) NULL, _IONBF, 0);
    if ( argc > MAX+1 ){ // verifier la liste d'arguments
        cerr << *argv << " arg1, arg2, ... arg" << MAX << endl;
        return 1;
    }
    cout << "Affichage" << endl;
    for (int i = 0; i < argc-1; ++i)
    { // creation des threads
      if( pthread_create(&thread_id[i],NULL,afficher, (void *)argv[i+1]) > 0)
        {
                cerr << "Echec a la creation des threads" << endl;
                return 2;
        }

    }
```

Threads POSIX : Exemple 5

Annexe

```
  for (int i=0; i < argc-1; ++i) { // attendre les threads
        if ( pthread_join(thread_id[i], &retour) > 0){
              cerr << "Echec de l'attente des threads" << endl;
              return 3;
        }
        cout<<endl<<"Thread"<<i<<" retourne "<<* retour<< " ";
  }
  cout << endl << "Termine" << endl;
  return 0;
  // fin de main
}
```

```
-bash-3.2$ g++ exemple3.cpp  -lpthread –o
exemple3
-bash-3.2$ ./exemple3  A B C
Affichage
A       sera affiche C  sera affiche 4 fois.4 fois.
B       sera affiche 4 fois.
A C B A C B A C B A C
Thread B 0 retourne 4
Thread 1 retourne 4
Thread 2 retourne 4
Termine
```

Threads POSIX : Exemple 5

Annexe

```
-bash-3.2$ ./exemple3 mot0 mot1 mot2 mot3 mot4
Affichage
mot3mot2mot0mot4     sera affiche mot1        sera affiche     sera
affiche   sera affiche 5   sera affiche 444fois.4fois.fois.fois.fois.

mot1mot3mot2mot4mot0        mot0mot1mot2mot4  mot3
mot3mot0mot1mot2  mot4      mot3mot4mot1mot2mot0
Thread 0 retourne 4
Thread 1 retourne 4
Thread 2 retourne 4 mot3
Thread 3 retourne 5
Thread 4 retourne 4
Termine
```