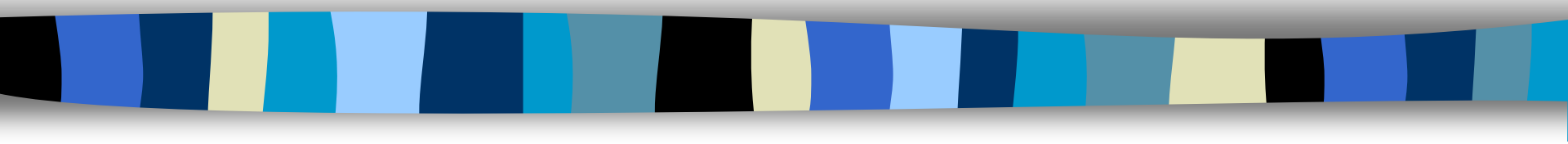
Chapitre 3



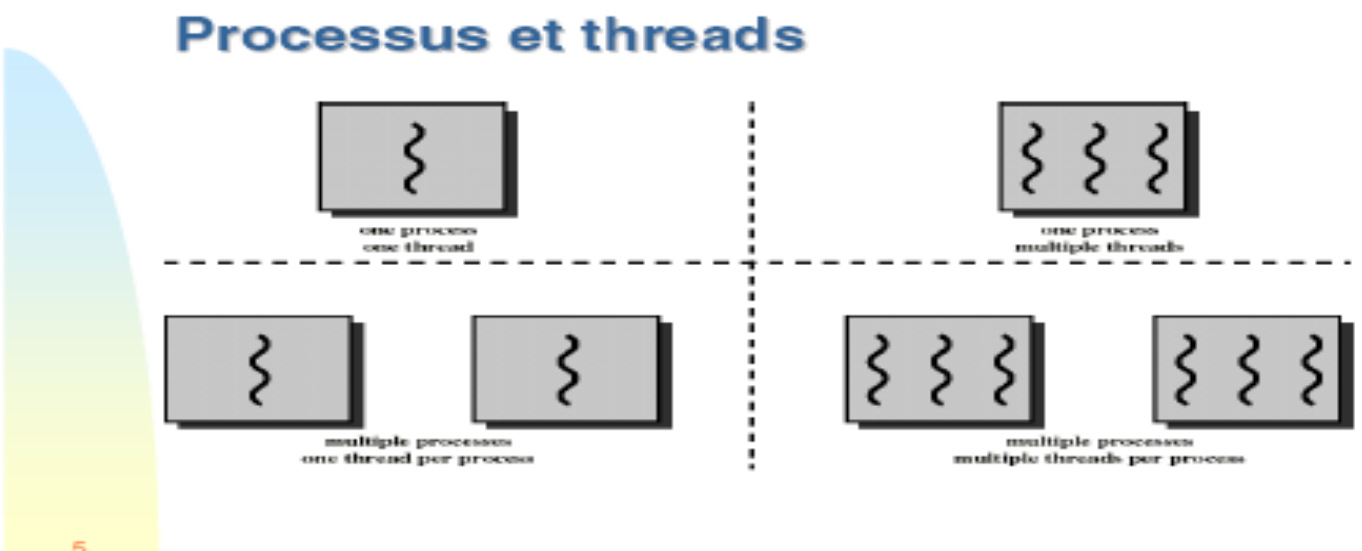
Threads

Plan du chapitre

- Qu'est ce qu'un thread?
- Usage des threads
- Implémentation des threads
  - Au niveau utilisateur (Java)
  - Au niveau noyau (Linux)
- Threads POSIX
  - Création d'un thread
  - Attente de la fin d'un thread
  - Terminaison d'un thread
  - Nettoyage à la terminaison

Qu'est ce qu'un thread ? Introduction (1)

- Décomposition d'une application en plusieurs tâches indépendantes.
  - Ces tâches peuvent s'exécuter en parallèle.
  - L'application peut s'exécuter sur plusieurs processeurs.
- Deux approches
  - Multi-processus
  - **Multi-*thread***
- **Multi-processus**
  - Les changement de contexte sont coûteux.
  - Chaque processus dispose de ses propres ressources.
  - La communication inter-processus est coûteuse (cf IPC).



Qu'est ce qu'un thread ? Introduction (2)

- Multi-thread
  - Un thread est une unité d'exécution rattachée à un processus, chargée d'exécuter une partie du processus.
  - Un processus est vu comme étant un ensemble de ressources (espace d'adressage, fichiers, périphériques…) que ses threads (flots de contrôle) partagent.
  - Lorsqu'un processus est créé, un seul flot d'exécution (thread) est associé au processus. Ce thread peut en créer d'autres.

  ➔ Un *thread* est un processus léger (*light weight process)*
- Réactivité (le processus peut continuer à s'exécuter même si certaines de ses parties sont bloquées).
- Les ressources sont partagées
  - Création rapide (env. 10-100 fois plus rapide que pour un processus)
  - Changements de contexte simplifiés et rapides
  - La communication entre les *threads* est "directe".
- Très bonne performance
- Notion d'*hyperthreading*
  - *Gestion multi-threads au niveau du processeur*

Threads – Caractéristiques

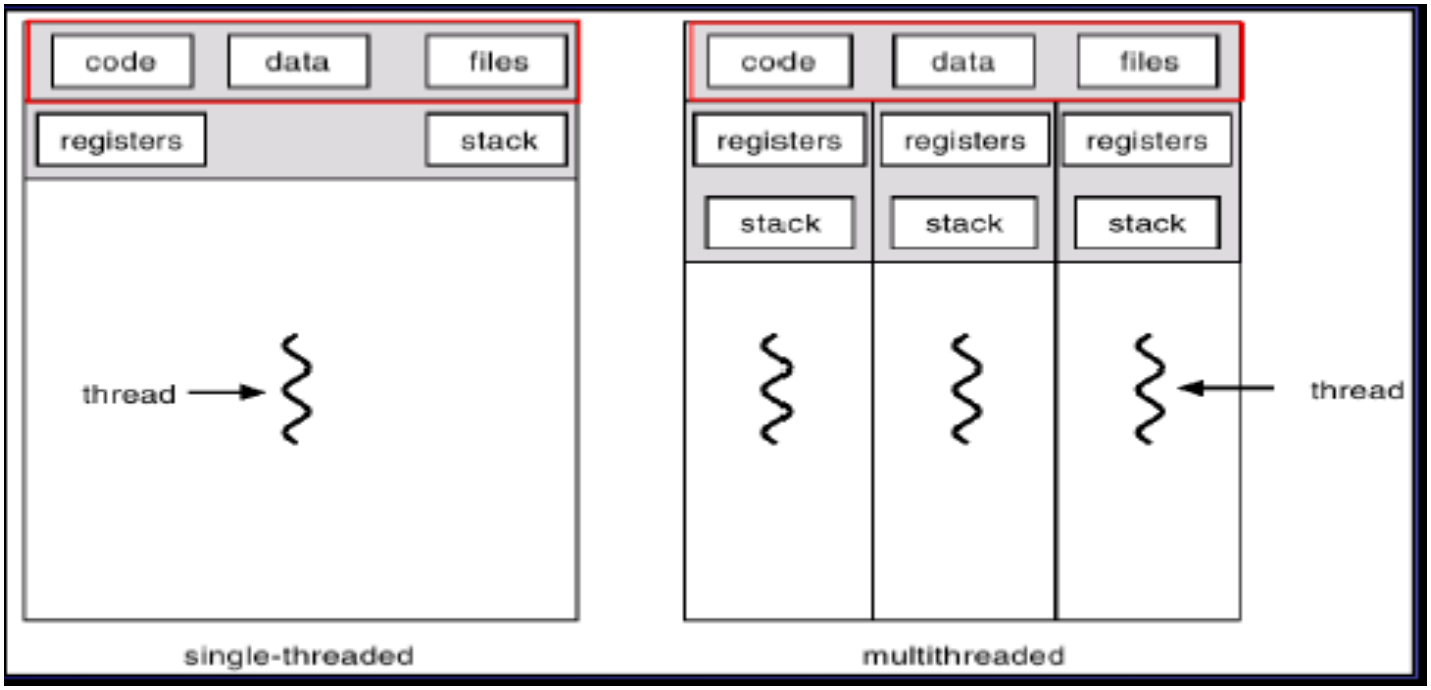
- Un ***thread*** a un contexte réduit
  - Un **compteur ordinal** (PC)
  - Des **registres**
  - Une **pile**
  - Un état
  - *Une priorité p*
- Par conséquent, un processus *multi-thread* a plusieurs piles !
- Le modèle de programmation *multi-thread* peut être soit **coopératif**, soit **préemptif**

Threads – Ressources

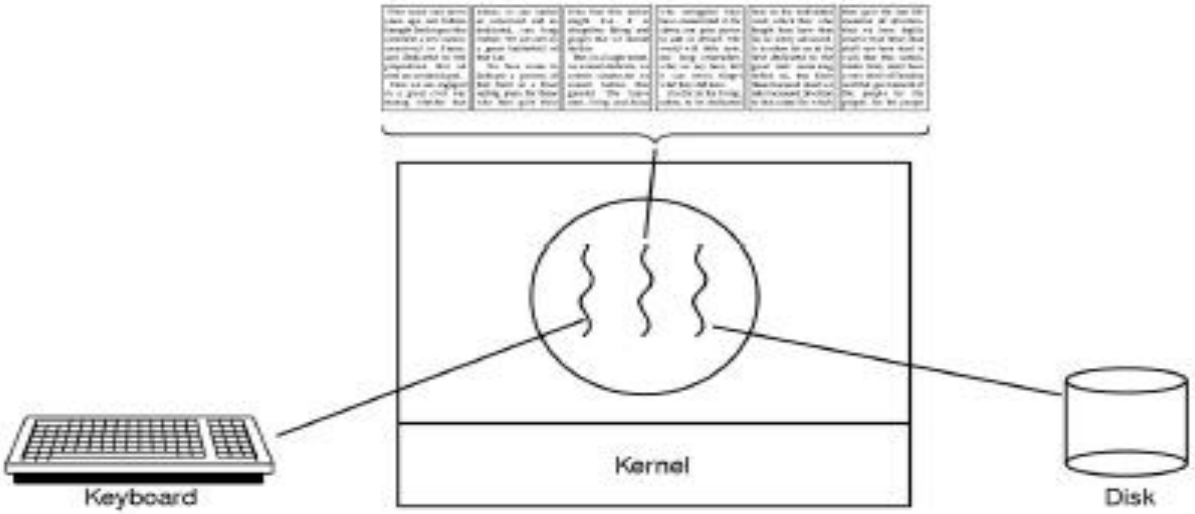
Le **thread** utilise **l'espace d'adressage du processus** ainsi que **toutes les ressources** gérées par le processus.

- Tous les threads se partagent les ressources du processus (code, variables globales, fichiers ouverts, …)
- La **protection** entre les *threads* **n'est pas garantie** !

- Les threads s'exécutent dans une même machine virtuelle.



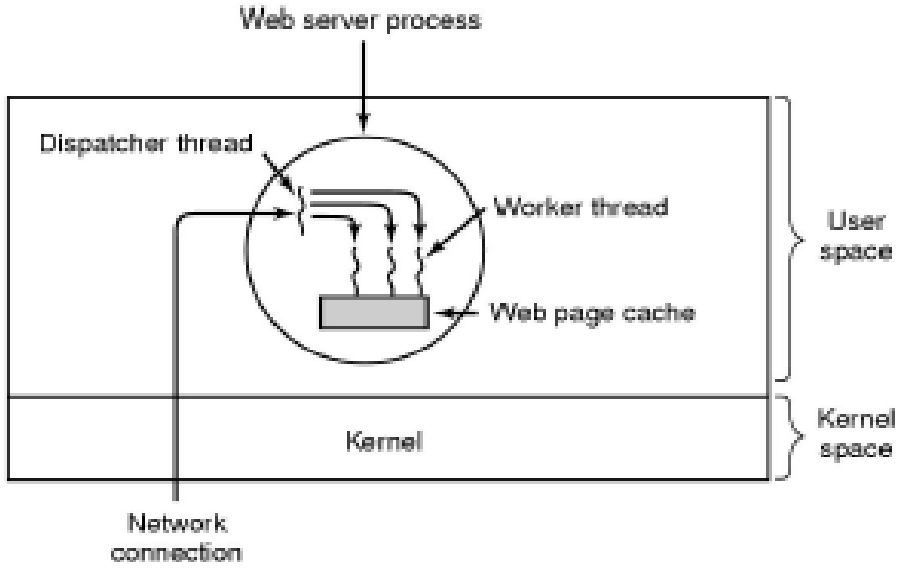
Usage des threads (1)
Word processor



- 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

- Le Dispatcher thread se charge de réceptionner les requêtes.
- Il choisit, pour chaque requête reçue, un thread libre (en attente d'une requête). Ce thread va se charger d'interpréter la requête.




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

(a)

```
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);
}
```

(b)

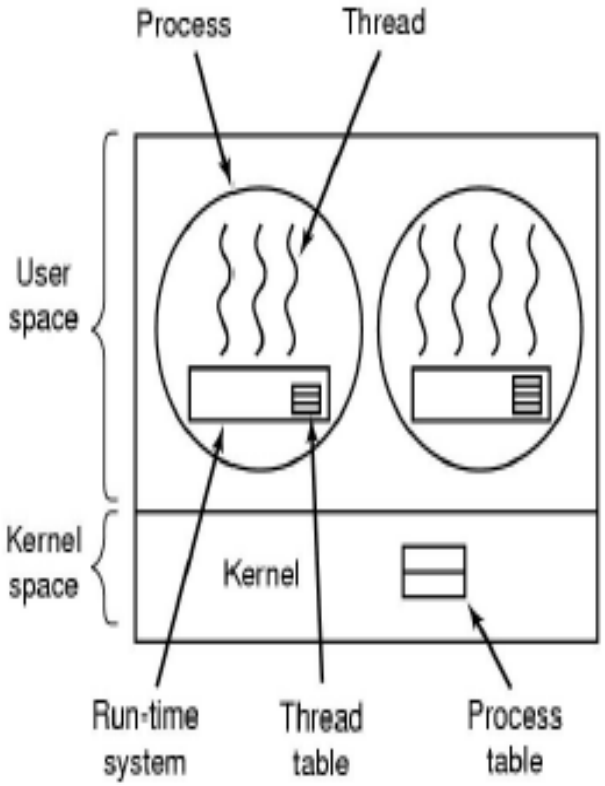
Implémentation des threads

- L'implémentation du multiflot (multithreading) varie considérablement d'une plateforme à une autre (threads Linux, threads de Win32, threads de Solaris et threads de POSIX).

- Le multiflot peut être implémenté :
  - au niveau utilisateur (threads utilisateur) -> modèle plusieurs-à- un,
  - au niveau noyau (threads noyau) -> modèle un-à-un, ou
  - aux deux niveaux (threads hybrides) -> modèle plusieurs-à-plusieurs.

Threads utilisateur (java threads, anciennes versions d'UNIX)

- Les threads utilisateur sont implantés dans une bibliothèque (niveau utilisateur) qui fournit un support pour les gérer.
- Ils ne sont pas gérés par le noyau.
- Le noyau gère les processus (table des processus) et ne se préoccupe pas de l'existence des threads (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).



Threads utilisateur (2)

- Les threads utilisateur sont généralement créés, et gérés rapidement.
- Ils facilitent la portabilité (comparativement aux autres implémentations)
- Inconvénients :
  - À tout instant, au plus un thread 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 utilisateur (3)
Threads Java

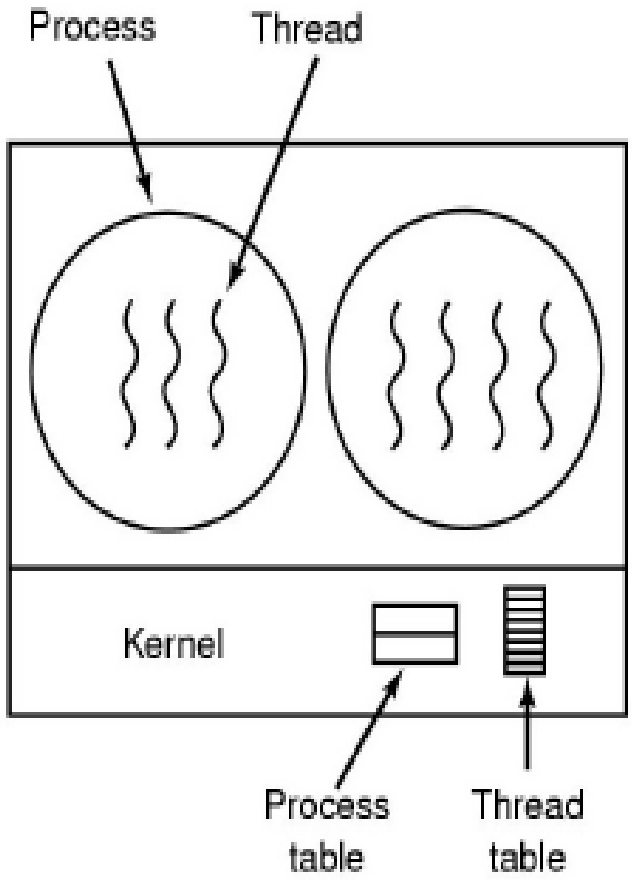
- Les threads Java peuvent être créés en :
  - en dérivant la class Thread ou
  - implémentant l'interface Runnable
- Plusieurs API pour contrôler les threads: **suspend()** , **sleep()** , **resume()** , **stop()** , etc.
- Les threads Java sont gérés par la machine virtuelle Java (JVM).

Threads utilisateur (4)
Threads Java

```
import java. awt.*;
class Travailleur extends Thread
{           public void run()
            {           System. out. println(" Je suis le thread Travailleur");
                        try { Thread. sleep( 1000); }
                        catch (InterruptedException e){}
                        System. out. println(" Je suis le thread Travailleur");
            }
}
public class Principal
{           public static void main( String args[])
            {           Travailleur t = new Travailleur();
                        t. start();
                        System. out. println(" Je suis le thread principal");
                        try { Thread. sleep( 1000); }
                        catch (InterruptedException e){}
                        System. out. println(" Je suis le thread principal");
            }
}
```

Threads noyau

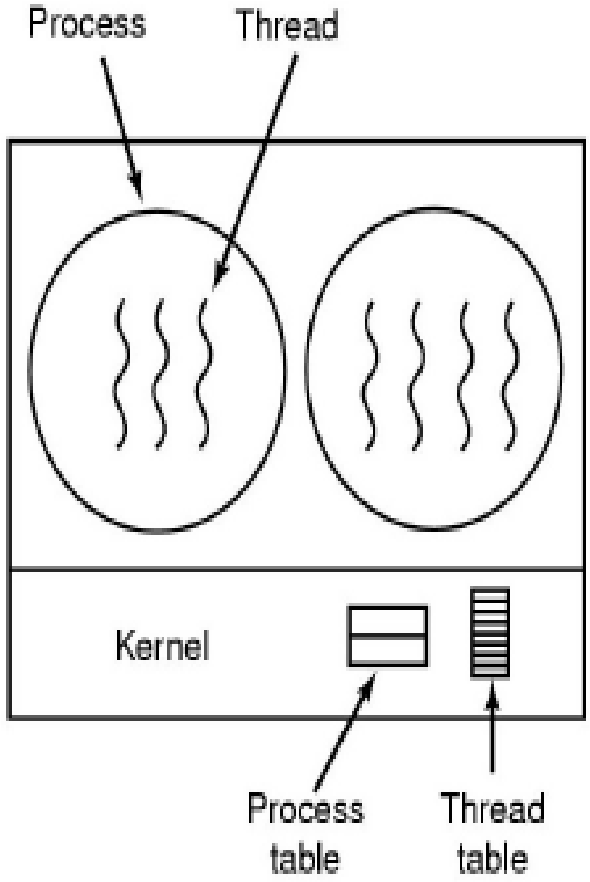
- Les threads noyau sont directement supportés par le système d'exploitation.
- Le système d'exploitation se charge de leur gestion. Un temps CPU est alloué à chaque thread. (modèle un-à-un)
- Si un thread d'un processus est bloqué, un autre thread du processus peut être élu par le noyau
- Cette implémentation est plus intéressante pour les systèmes multiprocesseurs.
- Un processus peux 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 threads noyau sont moins portables que ceux qui utilisent des threads utilisateur.



Threads noyau (3)
Linux

- Linux ne fait pas de distinction entre les processus et les threads 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 clone().
- Clone() permet de spécifier les ressources à partager (espace d'adressage, fichiers, signaux..) entre les tâches créatrice et créée.

Threads POSIX

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

#include <pthread.h>

Thread call	Description
pthread_create	Create a new thread in the caller's address space
pthread_exit	Terminate the calling thread
pthread_join	Wait for a thread to terminate
pthread_mutex_init	Create a new mutex
pthread_mutex_destroy	Destroy a mutex
pthread_mutex_lock	Lock a mutex
pthread_mutex_unlock	Unlock a mutex
pthread_cond_init	Create a condition variable
pthread_cond_destroy	Destroy a condition variable
pthread_cond_wait	Wait on a condition variable
pthread_cond_signal	Release one thread waiting on a condition variable

Threads POSIX (2)
Fonction pthread_create()

- **int pthread_create(**

pthread_t *tid,
// sert à récupérer le TID du thread créé

const pthread_attr_t *attr,
// sert à préciser les attributs du thread `(taille de la pile, priorité….)
//attr = NULL pour les attributs par défaut

void * (*func) (void*),
// est la fonction à exécuter par le thread

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

- L'appel renvoie 0 s'il réussit, sinon il renvoie une valeur non nulle identifiant l'erreur qui s'est produite

Threads POSIX (3)
Fonctions 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.

Threads POSIX (4)
Fonction 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 (plus de valeur de retour attendue).
- Un thread peut annuler ou terminer l'exécution d'un autre thread (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

```
// exemple_threads.c
#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);
}
```

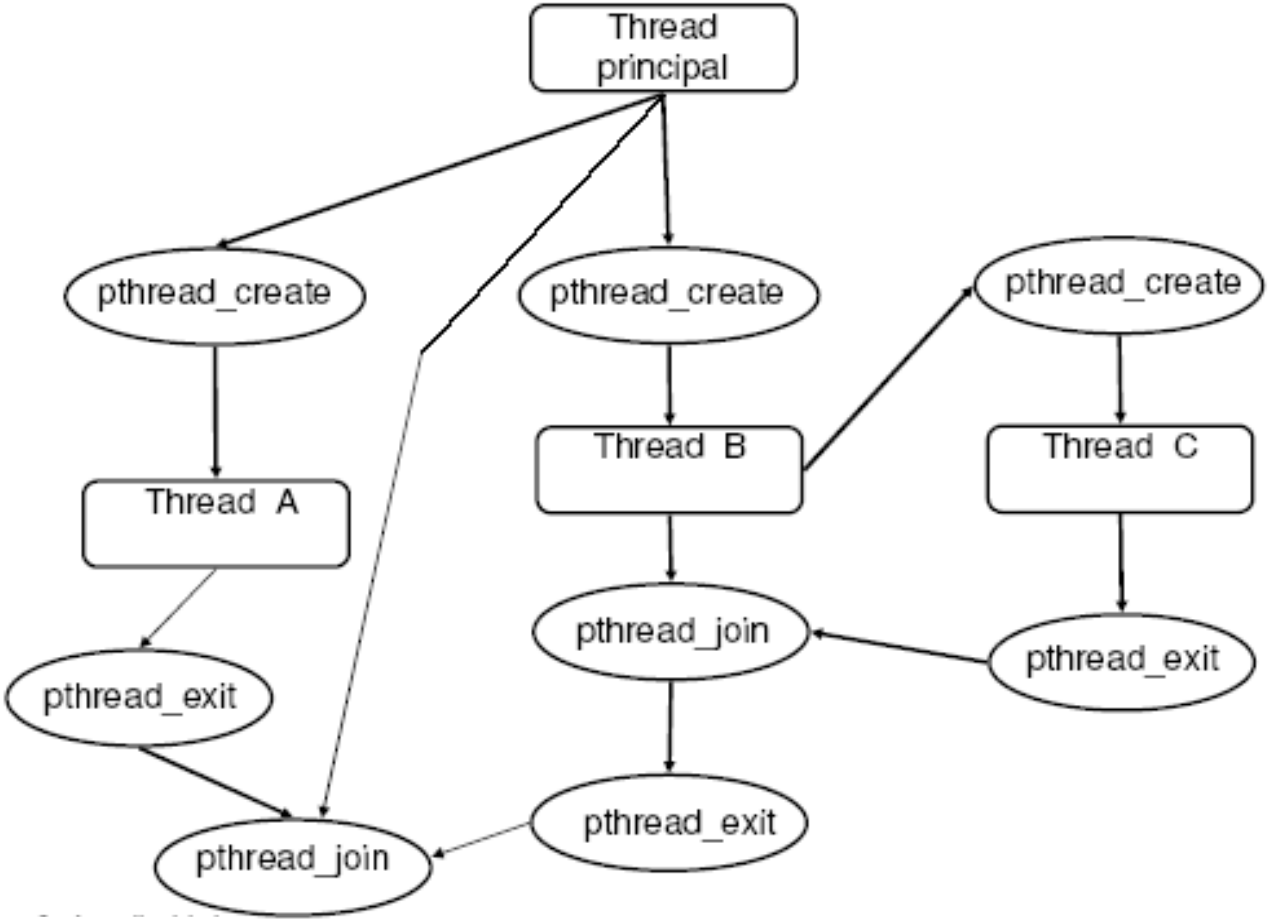
Threads POSIX (6)
Exemple 1 (suite)

```
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 (7)
Exemple 1 (suite)

```
int main()
{
        int i;

        pthread_t thA, thB;

        printf("Creation du thread ");

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

        sleep(1);
        //attendre que les threads aient termine
        printf("Le thread principal attend que les autres se terminent\n");

        pthread_join(thA,NULL);
        pthread_join(thB,NULL);

        exit(0);
}
```

Threads POSIX (8)
Exemple 1 (suite)



Threads POSIX (9)
Exemple 1 (suite)

```
bash-2.05b$ gcc -o ex_th -lpthread exemple_threads.c
bash-2.05b$ ./ex_th
Creation du thread
AAAAAAAAAABBBBBBBBBCCCCCCCCCAAAAAAAAABBBBBBBBBCCCC
CC Le thread principal attend que les autres se terminent
CCAAAAAAAAABBBBBBBBBCCCCCCCCCAAAAAAAAABBBBBBBBBCC
CCCCCCCAAAAAAAABBBBBBBBBCCCCCCCCCAAAAAAAABBBBBBB
BBCCCCCCCCAAAAAAAABBBBBBBBBCCCCCCCCCAAAAAAAABBB
BBBBBBCCCCCCCCCAAAAAAAABBBBBBBBCCCCCCCCAAAAAAAAB
BBBBBBBBCCCCCCCCCAAAAAAAA
Fin du thread A
BBBBBBBBBCCCCCCCCCB
Le thread B attend la fin du thread C
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
Fin du thread C
Fin du thread B
```

Threads POSIX (10)
Exemple 2 : Partage de variables

```
// programme threads.c
#include <unistd.h> //pour sleep
#include <pthread.h>
#include <stdio.h>
int glob=0;

void* decrement(void * x)
{
        int dec=1;
        sleep(1);
        glob = glob - dec ;
        printf("ici decrement[%d], glob = %d\n",pthread_self(),glob);
        pthread_exit(NULL);
}

void* increment (void * x)
{
        int inc=2;
        sleep(10);
        glob = glob + inc;
        printf("ici increment[%d], glob = %d\n",pthread_self(), glob);
        pthread_exit(NULL);
}
```

Threads POSIX (11)
Exemple 2 : Partage de variables (suite)

```
int main( )
{
        pthread_t tid1, tid2;
        printf("ici main[%d], glob = %d\n", getpid(),glob);
        //creation d'un thread pour increment
        if ( pthread_create(&tid1, NULL, increment, NULL) != 0)
                return -1;
        printf("ici main: creation du thread[%d] avec succes\n",tid1);
        // creation d'un thread pour decrement
        if ( pthread_create(&tid2, NULL, decrement, NULL) != 0)
                return -1;
        printf("ici main: creation du thread [%d] avec succes\n",tid2);
        // attendre 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 (12)
Exemple 3

```
/* Trois_Th.c */

#include <pthread.h>
     int compteur[3];
     /* fonction executee par chaque
     thread */
     void *fonc_thread(void *k) {
     printf("Thread numero %d :
     mon tid est %d\n",(int) k,
     pthread_self());
     for(;;) compteur[(int) k]++;
}
main() {
     int i, num; pthread_t pth_id[3];
     /* creation des threads */
     for(num=0;num<3;num++)
     {pthread_create(pth_id+num, 0,
     fonc_thread,(void *) num);
     printf("Main: thread numero
     %d creee: id = %d\n",num,
     pth_id[num]);
}
     usleep(10000); /* attente de
     10 ms */
     printf("Affichage des
     compteurs\n");
     for(i=0;i<20; i++) {
     printf("%d \t%d \t%d\n",
     compteur[0], compteur[1],
     compteur[2]);
     usleep(1000);
     /* attente de 1 ms entre 2
     affichages */
}
     exit(0); }
```

Thread POSIX (13)
Exemple 4 : Annulation d'un thread

```
//cancel2.c
#define _REENTRANT
#include <pthread.h>
#include <unistd.h>
#include <stdio.h>
// à appeler pour libérer l'espace alloué
void free (void *arg){
        int * a = (int*)arg;
        printf("ici free %d \n", *a);
        delete a;
}

//à appeler si pas de libération
void notfree (void * arg)
{
  if (arg ==NULL) printf("ici notfree \n");
}
```

```
// la fonction des threads
void *mon_thread(void *allocate)
{
        int* block;
        if (*((int*)allocate))
        {       block = new int;  // allocation dynamique
                *block = 123;
          pthread_cleanup_push(free, (void *) block);
           printf( "push free\n");
           pthread_cleanup_pop(1);
        } else
        {     block = NULL;   // pas d'allocation
          pthread_cleanup_push(notfree, (void *) block);
              printf("push notfree\n");
              pthread_cleanup_pop(1);
        }
       sleep(3);
       printf("fin du thread\n");
       pthread_exit(0);
}
```

Thread POSIX (14)
Exemple 4 : Annulation d'un thread (suite)

```
int main()
{       int allocate = 1;
        pthread_t th;
      // création d'un thread qui effectuera le bloc du if
        if (pthread_create(&th,NULL,mon_thread,(void*)&allocate))
                perror("Erreur dans la creation du thread");
        sleep(1);
        pthread_cancel(th);
        allocate = 0;
      //création d'un thread qui exécutera le else du if
        if (pthread_create(&th,NULL,mon_thread,(void*)&allocate) != 0)
                perror("Erreur dans la creation du thread");
        sleep(1);
        pthread_cancel(th);
        printf("fin du main\n");
}

//Dans un bloc, il doit y avoir autant pthread_cleanup_pop qu'il y a de pthread_cleanup_push.
//En cas d'annulation, les push/pop des blocs en cours sont considérées.
// Le fonctionnement est similaire aux destructeurs des classes de C++.
```

```
d5333-09> cancel2
push free
ici free 123
fin du thread
push notfree
ici notfree
fin du thread
fin du main
```

Thread POSIX (15)
Exemple 5

```
#include <pthread.h>
#include <stdio.h>
#include <errno.h>
typedef struct {
int x, y;
} data;
                /* fonction executee par chaque thread */
                void *mul(data *ptrdata)
                {
                pthread_exit((void *)((ptrdata->x) *
                (ptrdata->y)));
                }
                void main(int argc, char *argv[]){
                int i;
                int a, b, c, d;
                pthread_t pth_id[2];
                data donnees1;
                data donnees2;
                int res1, res2;
                if(argc < 5) { perror("\007Nbre
                d'arguments incorrect"); exit(1);}
```

Thread POSIX (16)
Exemple 5 (suite)

```
a=atoi(argv[1]); b=atoi(argv[2]);
c=atoi(argv[3]); d=atoi(argv[4]);
donnees1.x=a; donnees1.y=b;
donnees2.x=c; donnees2.y=d;

/* creation des threads */

pthread_create(pth_id, 0, (void *(*)())mul, &donnees1);
pthread_create(pth_id+1, 0, (void *(*)())mul, &donnees2);

/* Attente de la fin de la thread 1 */
pthread_join(pth_id[0], (void **)&res1);

/* Attente de la fin de la thread 2 */
pthread_join(pth_id[1], (void **)&res2);

/* Affichage du resultat a*b + c*d */

printf("Resultat =%d\n", res1+res2);

/* Suppression des ressources des threads */

pthread_detach(&pth_id[0]);
pthread_detach(&pth_id[1]);
exit(0);
}
```

Thread POSIX (15)
Exemple 5

- compilation avec :
- gcc thread_exo1.c -o exo1 -lpthread