Sixième chapitre

Programmer avec les threads

- Introduction aux threads
  - Comparaison entre threads et processus
  - Avantages et inconvénients des threads
  - Design d'applications multi-threadées
- Modèles de threads
  - Threads utilisateur ou noyau, modèles M:1, 1:1 et M:N
  - Études de cas
- Programmer avec les Pthreads
  - Gestion des threads
  - Mutexes, sémaphores POSIX et variables de condition
- Conclusion

Qu'est ce qu'un thread

Définition d'un thread

- Techniquement : un flot d'exécution pouvant être ordonnancé par l'OS
- En pratique : une procédure s'exécutant indépendemment du main
- Mot français : fil d'exécution ou processus léger (*lightweight process*)

Programme *multi-threadé*

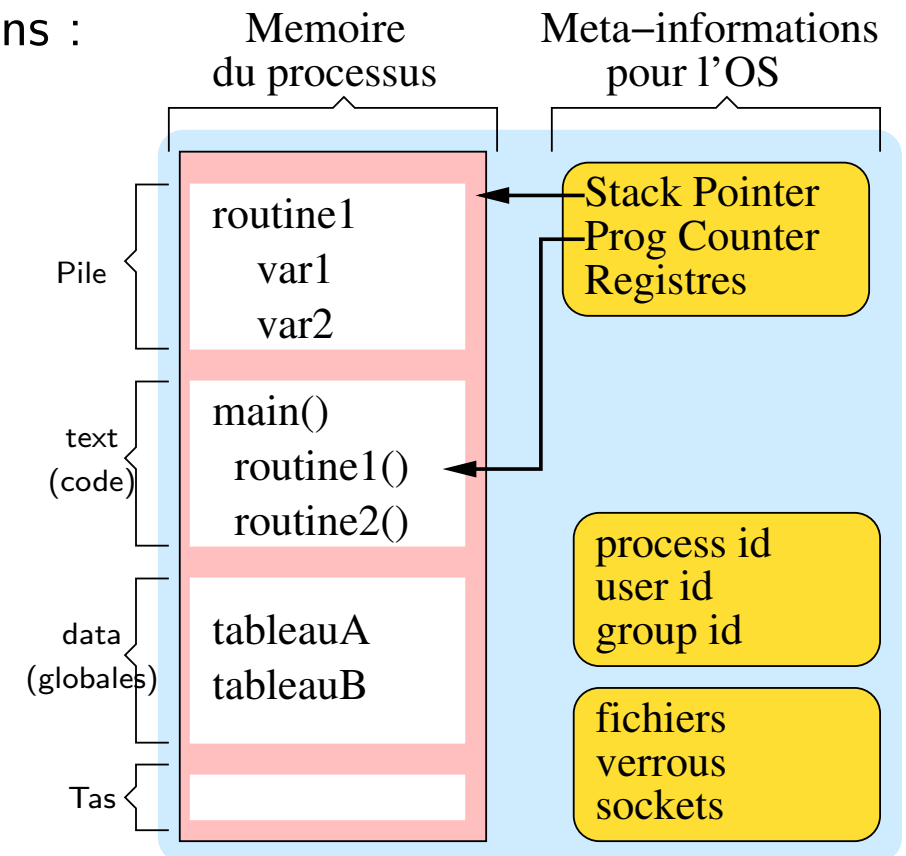
- Un programme contenant diverses fonctions et procédures
- Ces procédures peuvent être exécutées simultanément et/ou indépendamment

Comparaison de processus et threads (1/2)

Processus «classique» (ou «lourd»)

Processus UNIX contient diverses informations :

- PID, PGID, UID, GID
- L'environnement
- Répertoire courant
- Pointeur d'instruction (PC)
- Registres
- Pile
- Tas
- Descripteurs de fichier
- Gestionnaires de signaux
- Bibliothèques partagées
- Outils d'IPC (tubes, sémaphores, shmem, etc)

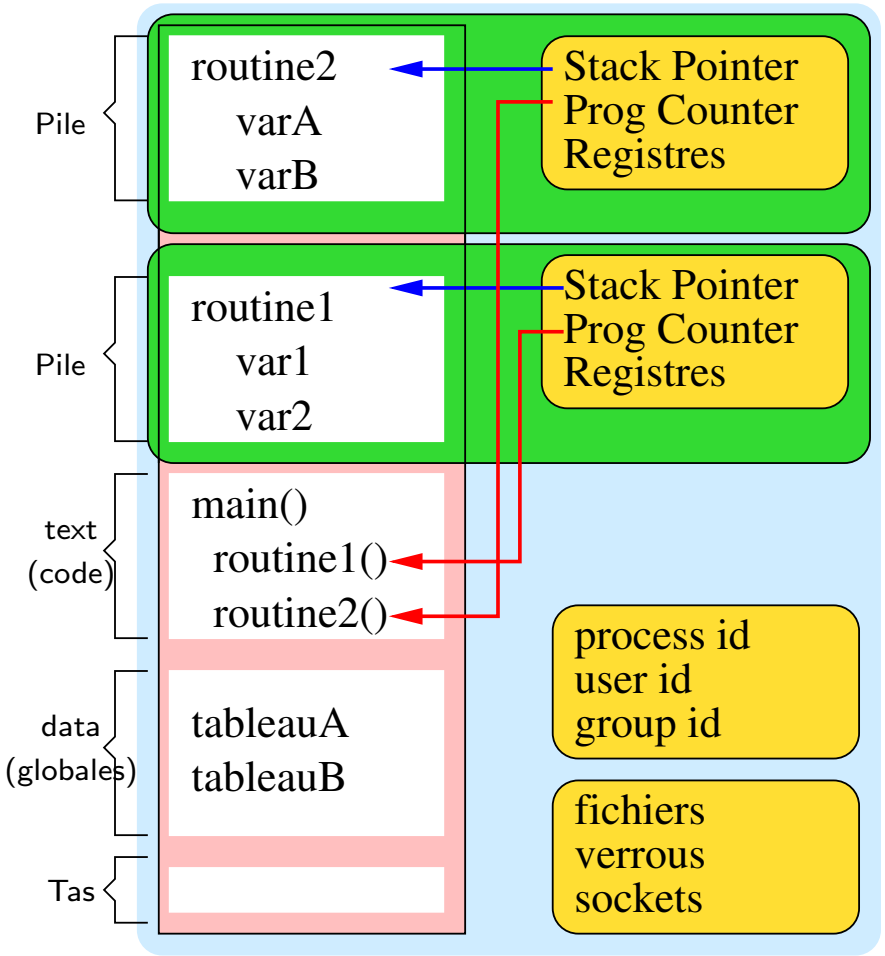


Processus classique

Comparaison de processus et threads (2/2)

Processus légers

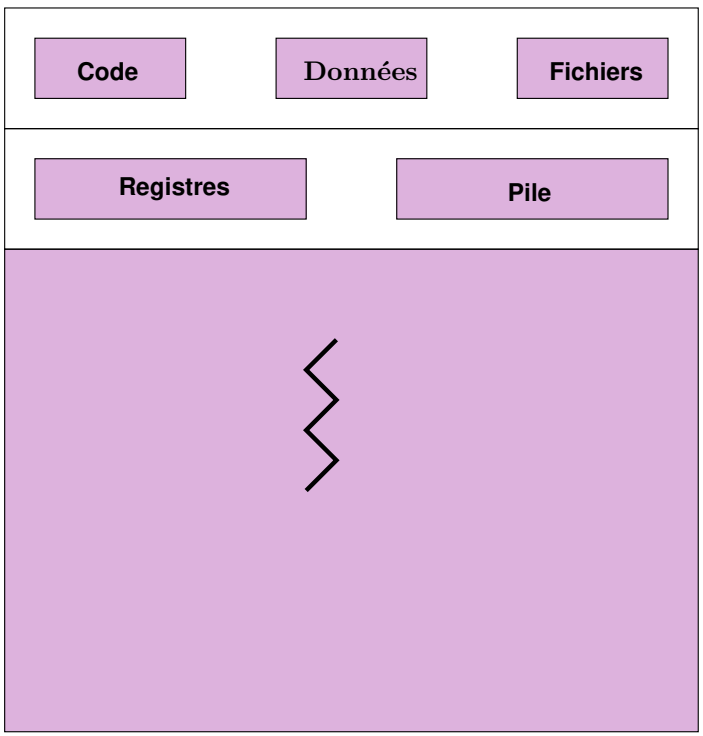
- Plusieurs threads coexistent dans le même processus ≪lourd≫
- Ils sont ordonnançables séparément
- Informations spécifiques
  - Pile
  - Registres
  - Priorité (d'ordonnancement)
  - Données spécifiques
  - Liste des signaux bloqués
- Le reste est partagé
  - Si un thread ferme un fichier, il est fermé pour tous
  - Si un thread fait `exit()`, tout s'arrête
  - Globales et pointeurs vers le tas : variables partagées (à synchroniser)



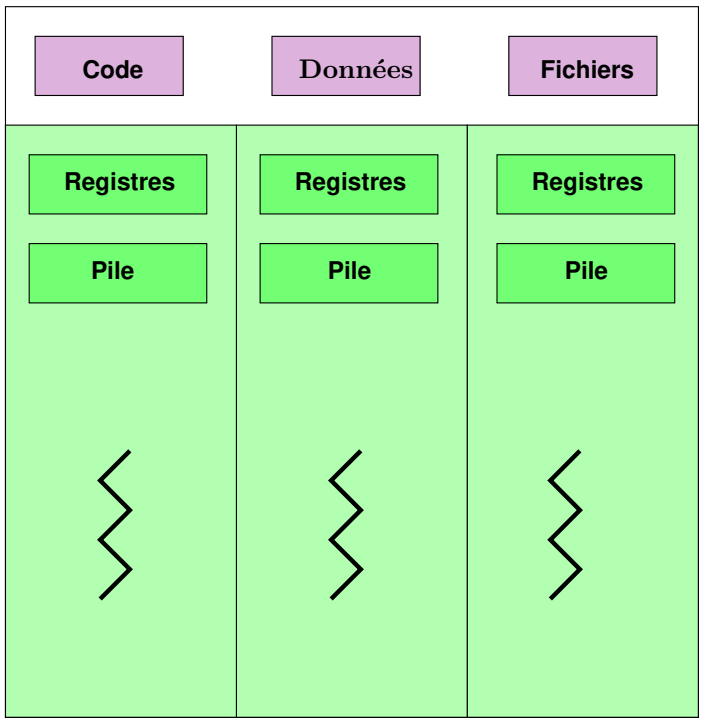
Processus legers

Processus vs. thread en résumé

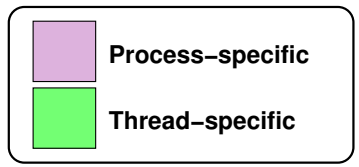
- Processus : environnement d'exécution pour les threads au sein de l'OS
  État vis-à-vis de l'OS (fichiers ouverts) et de la machine (mémoire)
- Processus = Thread + Espace d'adressage



Processus mono–thread



Processus multi–threads



L'espace d'adressage est *passif* ; le thread est *actif*

Pourquoi processus dits «légers» ?

Les threads contiennent moins de choses

- Partagent la plupart des ressources du processus lourd
- Ne dupliquent que l'indispensable
- $\Rightarrow$ 2 threads dans 1 processus consomment moins de mémoire que 2 processus

Les threads sont plus rapides à créer

Machine	fork()			threads		
	real	user	sys	real	user	sys
Celeron 2GHz	4.479s	0.364s	3.756s	1.606s	0.380s	0.388s
AMD64 2.5GHz (4CPU)	7.006s	0.936s	6.244s	0.903s	0.300s	0.640s

Les communications inter-threads sont rapides

- Bus PCI (donc, carte réseau) : 128 Mb/s
- Copie de mémoire (Tubes ; Passage message sur SMP) : 0,3 Gb/s
- Mémoire vers CPU (Pthreads) : 4 Gb/s

Usage des threads

Pourquoi utiliser les threads

- Objectif principal : gagner du temps (threads moins gourmands que processus)

Quand utiliser les threads

- Pour un recouvrement calcul/communication
- Pour avoir différentes tâches de priorité différentes
  ordonnancement temps réel ≪mou≫
- Pour gérer des événements asynchrones
  Tâches indépendantes activées par des événements de fréquence irrégulière
  Exemple : Serveur web peut répondre à plusieurs requêtes en parallèle
- Pour tirer profit des systèmes SMP ou CMP (multi-cœurs)

Quand (pourquoi) ne pas utiliser les threads

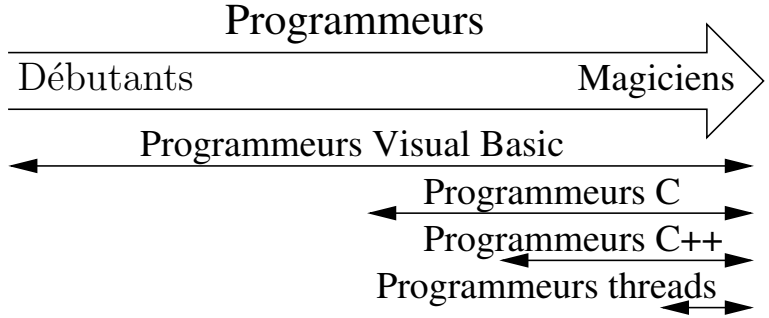
Problèmes du partage de la mémoire

- Risque de corruption mémoire (risque de compétition)
- Besoin de synchronisation (risque d'interblocage)

$\Rightarrow$ Communication inter-threads rapide mais dangereuse

- Segfault d'un thread $\rightarrow$ mort de tous
- Casse l'abstraction en modules indépendants
- Extrêmement difficile à debugger (dépendances temporelles ; pas d'outils)

Programmer avec les threads,
c'est enlever les gardes-fous de l'OS
pour gagner du temps



(Why Threads are a Bad Idea, USENIX96)

Obtenir de bonnes performances est très difficile

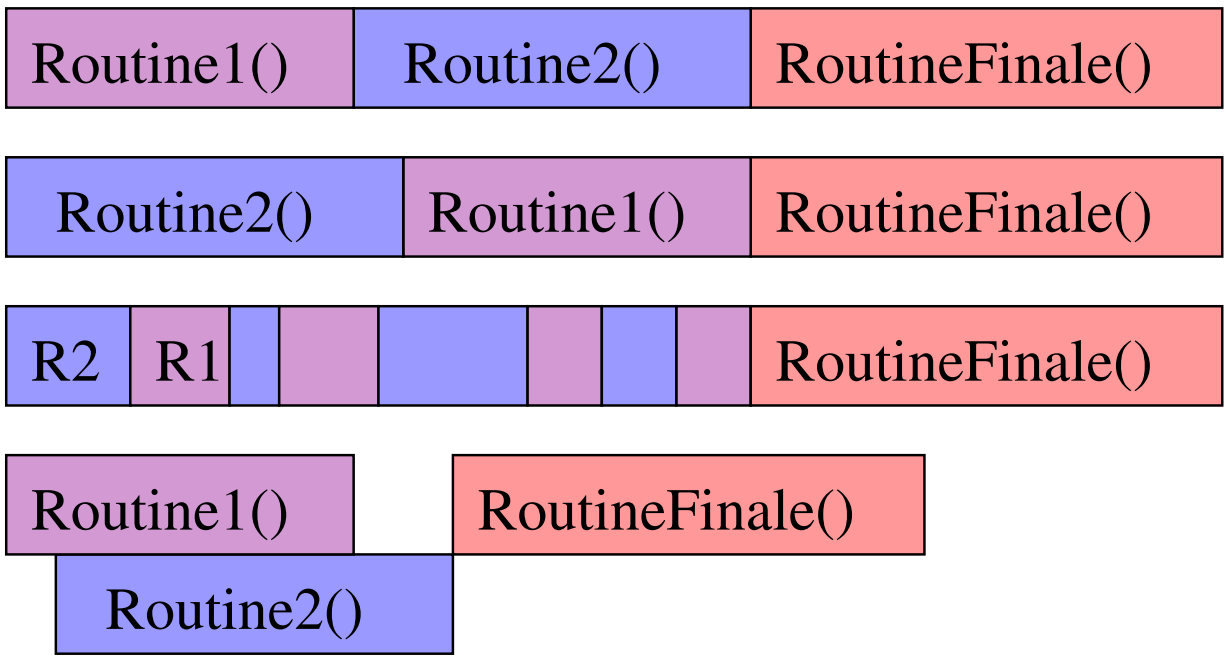
- Verrouillage simple (moniteurs) amène peu de concurrence
- Verrouillage fin augmente la complexité (concurrence pas facilement meilleure)

Design d'applications multi-threadées

Applications candidates

- Applis organisées en tâches indépendantes s'exécutant indépendamment
- Toutes routines pouvant s'exécuter en (pseudo-)parallèle sont candidates
- Interface interactive ($\rightsquigarrow$ latence indésirable) avec des calculs ou du réseau

Dans l'exemple, routine1() et routine2() sont candidates



Code réentrant et threads-safeness

Code réentrant

- Définition : Peut être appelé récursivement ou depuis plusieurs «endroits»
- Ne pas maintenir d'état entre les appels
  - Contre-exemple : `strtok`, `rand` (`strtok_r` est réentrante)
- Ne pas renvoyer un pointeur vers une statique
  - Contre-exemple : `ctime` (`ctime_r` est réentrante)

Code thread-safe

- Définition : Fonctionne même si utilisé de manière concurente
- Si le code n'est pas réentrant, il faut le protéger par verrous

Problème des dépendances

- Votre code est thread-safe, mais vos bibliothèques le sont-elles ?
- La libc est réentrante (sous linux modernes)
  - Exemple de errno : chaque thread a maintenant son propre errno
- Pour le reste, il faut vérifier (voire, supposer qu'il y a un problème)
- En cas de problème, il faut protéger les appels grâce à un verrou explicite

Patterns classiques avec les threads

Maitre/esclaves

- Un thread centralise le travail à faire et le distribue aux esclaves
- Pool d'esclaves statique ou dynamique
- Exemple : serveur web

Pipeline

- La tâche est découpée en diverses étapes successives
  Chaque thread réalise une étape et passe son résultat au suivant
- Exemple : traitement multimédia (streaming)

Peer to peer

- Comme un maitre/esclaves, mais le maitre participe au travail

Sixième chapitre

Programmer avec les threads

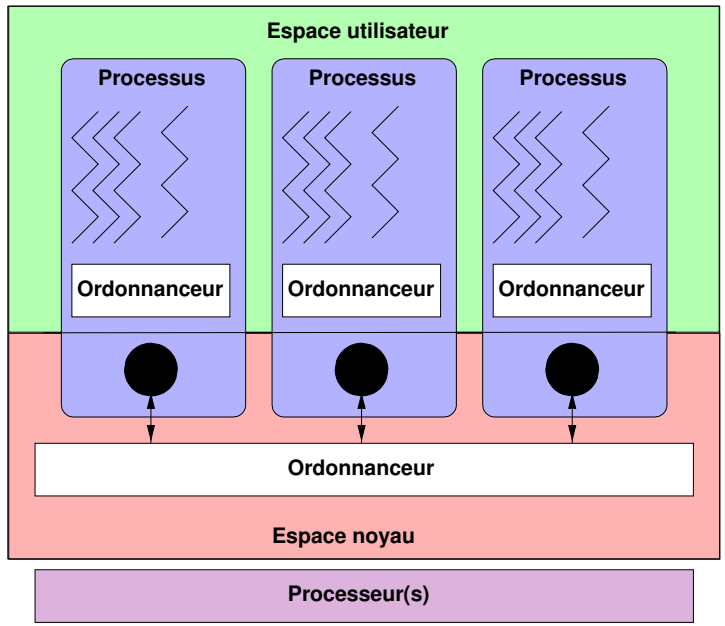
- Introduction aux threads
  - Comparaison entre threads et processus
  - Avantages et inconvénients des threads
  - Design d'applications multi-threadées
- Modèles de threads
  - Threads utilisateur ou noyau, modèles M:1, 1:1 et M:N
  - Études de cas
- Programmer avec les Pthreads
  - Gestion des threads
  - Mutexes, sémaphores POSIX et variables de condition
- Conclusion

Implantation des threads : Modèle M:1

Tous les threads d'un processus mappés sur un seul thread noyau

Service implémenté dans une bibliothèque spécifique

- Gestion des threads au niveau utilisateur
- Avantages :
  - Pas besoin de modifier le noyau
  - Rapides pour créer un thread, et pour changer de contexte
  - Portables entre OS
- Inconvénients :
  - Ordonnancement limité à celui d'un processus système
    $\Rightarrow$ ≪Erreurs≫ d'ordonnancement
  - Appel bloquant dans un thread
    $\Rightarrow$ blocage de tous
  - Pas de parallélisme
    (dommage pour les SMP)



(D'après F. Silber)

$\Rightarrow$ Efficace, mais pas de concurrence

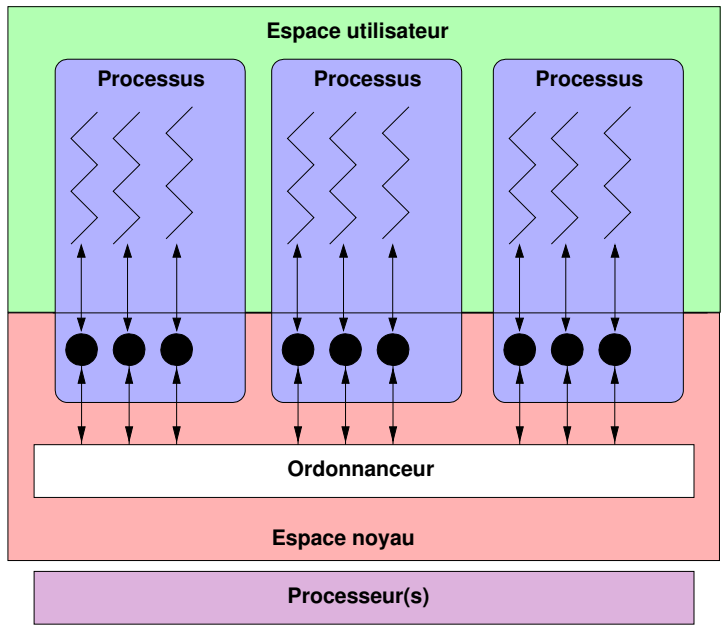
- Exemples : Fibres Windows, Java Green Threads, GNU pth (*portable thread*)

Modèle 1:1

Chaque thread utilisateur mappé sur un thread noyau

Modification du noyau pour un support aux threads

- Gestion des threads au niveau système (LightWeight Process – LWP)
- Avantages
  - Appel bloquant dans un thread
    $\Rightarrow$ exécution d'un autre
  - Parallélisme possible
  - Ordonnancement du noyau plus approprié
- Inconvénients
  - Besoin d'appels systèmes spécifiques
    (`clone()` pour la création sous linux)
  - Commutation réalisée par le noyau
    $\Rightarrow$ changements de contexte (coûteux)



(D'après F. Silber)

$\Rightarrow$ Concurrence importante, mais moins efficace
(nombre total de threads souvent borné)

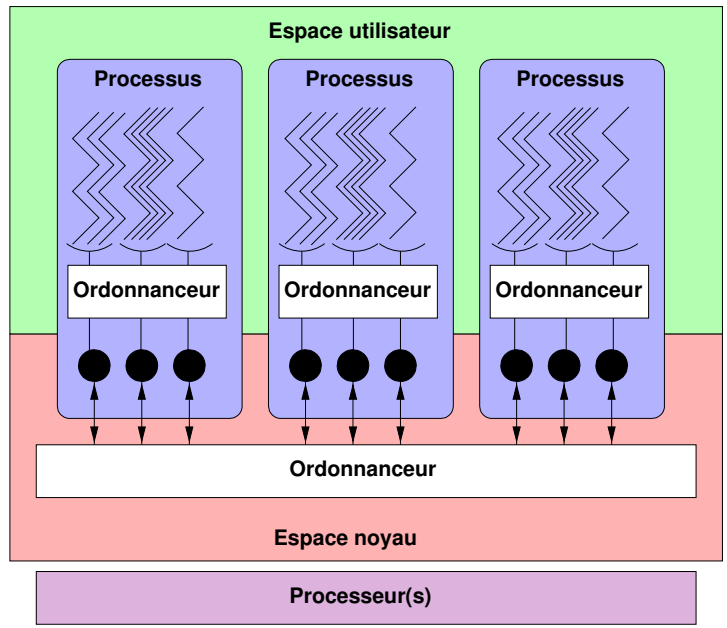
- Exemples : Windows 95/98/NT/2000, OS/2, Linux, Solaris 9+

Modèle M:N

M threads utilisateur mappés sur N threads noyau ($M \geq N \geq 1$)

Services utilisateur basés sur des services noyau

- Coopération entre l'ordonnanceur noyau et un ordonnanceur utilisateur
- Avantages : le meilleur des deux mondes
  - Threads noyaux plus basiques et efficaces
  - Threads utilisateurs plus flexibles
  - Moins de changement de contexte
  - Pas d'erreur d'ordonnancement
- Inconvénients :
  - Extrêmement difficile à implémenter
  - Un peu ≪usine à gaz≫ :
    - modèle théorique plus pur
    - implémentations complexes
    - efficacité parfois discutable
- Concurrence importante, bonne efficacité (?)
- Exemples : Solaris avant v9, IRIX, HP-UX



(D'après F. Silber)

Études de cas (1/3)

Pthreads

- POSIX threads : norme standardisé par IEEE POSIX 1003.1c (1995)
- Ensemble de types et fonctions C dont la sémantique est spécifiée
- Seulement une API : implémentation sous tous les UNIX (et même Windows)

Threads Solaris

- Modèle M:N
- Threads intermédiaires (LWP)

Threads Windows 2000

- Modèle 1:1 pour les WinThreads
- Modèle M:1 pour les fibres (l'OS ne voit pas les fibres)

Threads Java

- Extension de la classe `Thread` ; Implantation de l'interface `Runnable`
- Implémentation dépend de la JVM utilisée

Études de cas (2/3)

Threads sous linux

- Threads mappé sur des *tâches* (tasks)
- Appel système `clone()` pour duppliquer une tâche (fork() implémenté avec ça)

Plusieurs bibliothèques implémentant le standard Pthreads

- Depuis 1996 : LinuxThreads
  - Modèle 1:1, par Xavier Leroy (INRIA, créateur d'Ocaml)
  - Pas complètement POSIX (gestion des signaux, synchronisation)
  - $\approx$ 1000 threads au maximum
- Next Generation POSIX Threads
  - Modèle M:N, par IBM ; abandonné

  $\rightsquigarrow$ NPTL avançait plus vite

  $\rightsquigarrow$ M:N induisait une complexité trop importante
- Maintenant Native POSIX Thread Library
  - Modèle 1:1, par Ulrich Drepper (acteur principal libc, chez Red Hat)
  - Totalement conforme à POSIX
  - Bénéficie des fonctionnalités du noyau Linux 2.6 (ordonnancement $O(1)$)
  - Création de 100 000 threads en 2 secondes (contre 15 minutes sans)

Études de cas (3/3)

Possibilités de quelques systèmes d'exploitations

	1 seul espace d'adressage	plusieurs espaces d'adressage
1 seul thread par espace d'adressage	MS/DOS Macintosh (OS9)	Unix traditionnels
plusieurs threads par espace d'adressage	Systèmes embarqués	Win/NT, Linux, Solaris HP-UP, OS/2, Mach, VMS

Sixième chapitre

Programmer avec les threads

- Introduction aux threads
  - Comparaison entre threads et processus
  - Avantages et inconvénients des threads
  - Design d'applications multi-threadées
- Modèles de threads
  - Threads utilisateur ou noyau, modèles M:1, 1:1 et M:N
  - Études de cas
- Programmer avec les Pthreads
  - Gestion des threads
  - Mutexes, sémaphores POSIX et variables de condition
- Conclusion

Généralités sur l'interface Pthreads

Trois grandes catégories de fonctions / types de données

Gestion des threads

- Créer des threads, les arrêter, contrôler leurs attributs, etc.

Synchronisation par mutex (**mut**ual **ex**clusion)

- Créer, détruire verrouiller, déverrouiller des mutex ; Contrôler leurs attributs

Variables de condition :

- Communications entre threads partageant un mutex
- Les créer, les détruire, attendre dessus, les signaler ; Contrôler leurs attributs

Préfixe d'identificateur	Groupe
`pthread_`	Threads eux-mêmes et diverses fonctions
`pthread_attr_`	Attributs des threads
`pthread_mutex_`	Mutexes
`pthread_cond_`	`Variables de condition`
...	...

L'interface Pthreads en pratique

- Types de données sont structures opaques, fonctions d'accès spécifiques
- L'interface compte 60 fonctions, nous ne verrons pas tout
- Il faut charger le fichier d'entête `pthread.h` dans les sources
- Il faut spécifier l'option `-pthread` à gcc

thread-ex1.c

```
#include <pthread.h>
void *hello( void *arg ) {
  int *id = (int*)arg;
  printf("%d: hello world \n", *id);
  pthread_exit(NULL);
}
int main (int argc, char *argv[ ]) {
  pthread_t thread[3];
  int id[3]={1,2,3};
  int i;

  for (i=0;i<3;i++) {
    printf("Crée thread %d\n",i);
    pthread_create(&thread[i], NULL,
                   hello, (void *)&id[i]);
  }
  pthread_exit(NULL);
}
```

```
$ gcc -pthread -o thread-ex1 thread-ex1.c
$ ./thread-exemple1
Crée thread 1
Crée thread 2
Crée thread 3
1 : hello world
2 : hello world
3 : hello world
$
```

Identification et création des threads

Identifier les threads

- **pthread_t** : équivalent du pid_t (c'est une structure opaque)
- pthread_t **pthread_self** () : identité du thread courant
- int **pthread_equal** (pthread_t, pthread_t) : test d'égalité

Créer de nouveaux threads

- Le programme est démarré avec un seul thread, les autres sont créés à la main
- int **pthread_create**(identité, attributs, fonction, argument) ;
  - identité : [pthread_t*] identifieur unique du nouveau thread (*rempli par l'appel*)
  - attributs : Pour modifier les attributs du thread (NULL → attributs par défaut)
  - fonction : [void *(*)(void *)] Fonction C à exécuter (le ≪main du thread≫)
  - argument : argument à passer à la fonction. Doit être transtypé en void*
  - retour : 0 si succès, errno en cas de problème

Exercice : Comment savoir dans quel ordre vont démarrer les threads créés ?

On peut pas, ca dépend des fois (gare aux races conditions)

Exercice : Comment passer deux arguments à la fonction ?

en définissant une structure

Exemple FAUX de passage d'arguments

thread-faux1.c

```
#include <pthread.h>
void *hello( void *id ) {
  printf("%d: hello world \n", (char *) id);
  pthread_exit(NULL);
}

int main (int argc, char *argv[ ]) {
  pthread_t th[3];
  int i;

  for (i=0;i<3;i++) {
    printf("Crée thread %d\n",i);
    pthread_create(&th[i], NULL, hello, (void *)&i);
  }

  pthread_exit(NULL);
}
```

Exercice : Quel est le problème ?

On transforme i en globale inter-thread, et sa valeur peut donc être changée dans le thread lanceur avant le démarrage du thread utilisateur. C'est une belle condition de compétition.

Exemple de passage d'arguments complexes

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define NUM_THREADS 8

typedef struct data {
   int thread_id,sum;
   char *msg;
} data_t;

data_t data_array[NUM_THREADS];

void *PrintHello(void *arg) {
  data_t *mine = (data_t *)arg;

  sleep(1);
  printf("Thread %d: %s  Sum=%d\n",
         mine->thread_id,
         mine->msg,
         mine->sum);
  pthread_exit(NULL);
}
```

```c
int main(int argc, char *argv[ ]) {
  pthread_t threads[NUM_THREADS];
  int rc, t, sum=0;

  char *messages[NUM_THREADS] = {
    "EN: Hello World!", "FR: Bonjour, le monde!",
    "SP: Hola al mundo", "RU: Zdravstvytye, mir!",
    "DE: Guten Tag, Welt!", "Klingon: Nuq neH!",
    "JP: Sekai e konnichiwa!",
    "Latin: Orbis, te saluto!" };

  for(t=0;t<NUM_THREADS;t++) {
    sum = sum + t;
    thread_data_array[t].thread_id = t;
    thread_data_array[t].sum = sum;
    thread_data_array[t].message = messages[t];
    printf("Creating thread %d\n", t);
    rc = pthread_create(&threads[t], NULL,
                        PrintHello,
                        (void*)&thread_data_array[t]);
    if (rc) {
      printf("ERROR; _create() returned %d\n", rc);
      exit(-1);
    }
  }
  pthread_exit(NULL);
}
```

Terminaison des threads

Causes de terminaison des threads

- Le thread termine sa fonction initiale
- Le thread appelle la routine `pthread_exit()`
- Le thread est tué par un autre thread appelant `pthread_cancel()`
- Tout le processus se termine à cause d'un `exit`, `exec` ou `return` du main()

Terminer le thread courant

- void **pthread_exit**(void *retval) ;
  - retval : valeur de retour du thread (optionnel)
- Pour récupérer code de retour, un autre thread doit utiliser `pthread_join()`

Attendre la fin d'un thread

Joindre un thread

- int **pthread_join** (pthread_t, void **)
- Le thread A joint le thread B : A bloque jusqu'à la fin de B
- Utile pour la synchronisation (rendez-vous)
- Second argument reçois un pointeur vers la valeur retour du `pthread_exit()`
- Similaire au `wait()` après `fork()`
  (mais pas besoin d'être le créateur pour joindre un thread)

Détacher un thread

- int **pthread_detach** (pthread_t)
- Détacher un thread revient à dire que personne ne le joindra à sa fin
- Le système libère ses ressources au plus vite
- Évite les threads zombies quand on ne veut ni synchro ni code retour
- On ne peut pas ré-attacher un thread détaché

Attributs des threads

C'est le second argument du create

Les fonctions associées

- Création et destruction d'un objet à passer en argument à _create :
  - int **pthread_attr_init** (pthread_attr_t *)
  - int **pthread_attr_destroy** (pthread_attr_t *)
- Lecture : int **pthread_attr_getX** (const pthread_attr_t *, **T** *)
- Mise à jour : int **pthread_attr_setX** (pthread_attr_t *, **T**)

Attributs existants

- detachstate (int) : PTHREAD_CREATE_[JOINABLE|DETACHED]
- schedpolicy (int)
  - SCHED_OTHER : Ordonnancement classique
  - SCHED_RR : Round-Robin (chacun son tour)
  - SCHED_FIFO : Temps réel
- schedparam (int) : priorité d'ordonnancement
- stackaddr (void*) et stacksize (size_t) pour régler la pile
- inheritsched, scope

Exemple de gestion des threads

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define NUM_THREADS 3

void *travail(void *null) {
 int i;
 double result=0.0;
 for (i=0; i<1000000; i++)
   result = result + (double)random();
 printf("Resultat = %e\n",result);
 pthread_exit(NULL);
}

int main(int argc, char *argv[ ]) {
 pthread_t thread[NUM_THREADS];
 pthread_attr_t attr;
 int rc, t;
 /* Initialise et modifie l'attribut  */
 pthread_attr_init(&attr);
 pthread_attr_setdetachstate(&attr,
                PTHREAD_CREATE_JOINABLE);
```

```
 for(t=0;t<NUM_THREADS;t++) {
  printf("Creating thread %d\n", t);
  if ((rc = pthread_create(&thread[t], &attr,
                           travail, NULL)))  {
   printf("ERREUR de _create() : %d\n", rc);
   exit(-1);
 }}
 /* Libère l'attribut */
 pthread_attr_destroy(&attr);
 /* Attend les autres */
 for(t=0;t<NUM_THREADS;t++) {
  if ((rc = pthread_join(thread[t],
     /*pas de retour attendu*/NULL))) {
   printf("ERREUR de _join() : %d\n", rc);
   exit(-1);
  }
  printf("Rejoint thread %d. (ret=%d)\n",
         t, status);
 }
 pthread_exit(NULL);
}
```

Sixième chapitre

Programmer avec les threads

- Introduction aux threads
  - Comparaison entre threads et processus
  - Avantages et inconvénients des threads
  - Design d'applications multi-threadées
- Modèles de threads
  - Threads utilisateur ou noyau, modèles M:1, 1:1 et M:N
  - Études de cas
- Programmer avec les Pthreads
  - Gestion des threads
  - Mutexes, sémaphores POSIX et variables de condition
- Conclusion

Les mutex Pthreads

Interface de gestion

- Type de donnée : **pthread_mutex_t**
- Création :
  - statique : pthread_mutex_t mutex = **PTHREAD_MUTEX_INITIALIZER**;
  - dynamique : **pthread_mutex_init**(pthread_mutex_t *, pthread_mutexattr_t *);
    Premier argument : adresse de (pointeur vers) la variable à initialiser
- Destruction : **pthread_mutex_destroy**(mutex) (doit être déverrouillé)
- (pas d'attributs POSIX, mais il y en a dans certaines implémentations)

Interface d'usage

- **pthread_mutex_lock**(mutex) Bloque jusqu'à obtention du verrou
- **pthread_mutex_trylock**(mutex) Obtient le verrou ou renvoie EBUSY
- **pthread_mutex_unlock**(mutex) Libère le verrou

Usage des mutex

Les mutex sont un «gentlemen's agreement»

Thread 1

```
Lock
A = 2
Unlock
```

Thread 2

```
Lock
A = A+1
Unlock
```

Thread 3

```
A = A*B
```

- Thread 3 crée une condition de compétition même si les autres sont disciplinés (cf. les verrous consultatifs sur les fichiers)
- Pas d'équivalent des verrous impératifs sur la mémoire
- Pas de moniteurs dans PThreads (durs à implémenter en C)

Sémaphores POSIX

- On parle ici de l'interface POSIX, pas de celle IPC Système V
- `#include <semaphore.h>`
- Type : **sem_t**

Interface de gestion

- int **sem_init**(sem_t *sem, int pshared, unsigned int valeur)
  pshared !=0 ⇒ sémaphore partagée entre plusieurs processus (pas sous linux)
- int **sem_destroy**(sem_t * sem) (personne ne doit être bloqué dessus)

Interface d'usage

- int **sem_wait**(sem_t * sem) réalise un P()
- int **sem_post**(sem_t * sem) réalise un V()
- int **sem_trywait**(sem_t * sem) P() ou renvoie EAGAIN pour éviter blocage
- int **sem_getvalue**(sem_t * sem, int * sval)

Variables de condition

Interface de gestion

- Type de donnée : **pthread_cond_t**
- Création :
  - statique : pthread_cond_t cond = **PTHREAD_COND_INITIALIZER**;
  - dynamique : int **pthread_cond_init**(pthread_cond_t *, pthread_condattr_t *)
- Destruction : int **pthread_cond_destroy**(pthread_cond_t *)
- Il n'y a pas d'attributs POSIX (ni linux) pour les conditions

Rappel du principe des conditions

- Cf. conditions Java (`wait()` et `notifyAll()`)
- Autre mécanisme de synchronisation.
  - mutex : solution pratique pour l'exclusion mutuelle
  - sémaphore : solution pratique pour les cohortes
  - condition : solution pratique pour les rendez-vous
  - compare-and-swap : solution pour la synchronisation non-bloquante

Exemple d'usage des conditions

Thread principal

(1) Déclare et initialise une globale requérant une synchronisation (ex : `compteur`)

(1) Déclare et initialise une variable de condition et un mutex associé

(1) Crée deux threads (A et B) pour faire le travail

Thread A

(2) Travaille jusqu'au rendez-vous (point où B doit remplir une condition, *compteur* > 0 par ex)

(2) Verrouille mutex et consulte variable

(2) Attend la condition de B
ça libère le mutex (pour B) et gèle A

(4) Au signal, réveil
mutex automagiquement repris

(4) Déverrouille explicitement

Thread B

(2) Fait des choses

(3) Verrouille le mutex

(3) Modifie la globale attendue par A

(3) Si la condition est atteinte, signale A

(3) Déverrouille le mutex

Thread principal : (5) Rejoint les deux threads puis continue

Les conditions POSIX

Interface d'usage

- `pthread_cond_signal(pthread_cond_t *)` Débloque un éventuel thread bloqué
- `pthread_cond_wait(pthread_cond_t *, pthread_mutex_t *)` Bloque tant que la condition n'est pas réalisée
- `pthread_cond_timedwait(pthread_cond_t*, pthread_mutex_t*,struct timespec*)` Bloque au plus tard jusqu'à la date spécifiée
- `pthread_cond_broadcast(pthread_cond_t *)` Réveille tous les bloqués (sont encore bloqués par mutex)

Pièges des conditions

- Risque d'interblocage ou de pas de blocage si protocole du mutex pas suivi
- Différence sémantique avec sémaphore :
  - Si sem==0, alors V(sem) $\Rightarrow$ *sem* := 1
  - Signaler une condition que personne n'attend : noop (info perdue)

Résumé du sixième chapitre

- Définition de threads (vs. processus) : proc=espace d'adressage+meta-données OS ; thread=fil d'exécution
- Avantages et inconvénients : Ca va plus vite, mais c'est plus dangereux
- Schémas d'implémentations :
  - Modèle M:1 : M threads par processus (portable et rapide, pas parallèle)
  - Modèle 1:1 : Threads gérés par l'OS directement (plus dur à implémenter, plus efficace en pratique)
  - Modèle M:N : Gestion conjointe (modèle théorique meilleur, difficile à faire efficacement)
- Les fonctions principales de l'interface POSIX :
  - `pthread_create` : crée un nouveau thread
  - `pthread_exit` : termine le thread courant
  - `pthread_join` : attend la fin du thread et récupère son errcode
  - `pthread_detach` : indique que personne ne rejoindra ce thread
  - `mutex_{lock,unlock,*}`: usage des mutex
  - `sem_{wait,post,*}`: usage des sémaphore
  - `cond_{wait,signal,*}`: usage des conditions