Processus et threads

ASR période 3

Sylvain Jubertie
sylvain.jubertie@univ-orleans.fr

Table des matières

Table des matières

Table des matières

Terminologie

Programme

Un **programme** est un fichier source écrit dans un langage donné, par exemple un programme `C`, `C++`, ou `Java`.

Programme exécutable, binaire

Un programme désigne également le fichier en langage binaire obtenu après compilation, on parle également de **programme exécutable** ou de **binaire**.

Processus

Un **processus** est un programme binaire en cours exécution par le système d'exploitation.

Qu'est-ce qu'un processus ?

Definition

Un processus est constitué d'un programme binaire : instructions + données statiques, associé à un **contexte d'exécution** :

- pile + tas
- données dans les registres du processeur
- instruction courante

Table des matières

Systèmes d'exploitation

Point de vue de l'OS

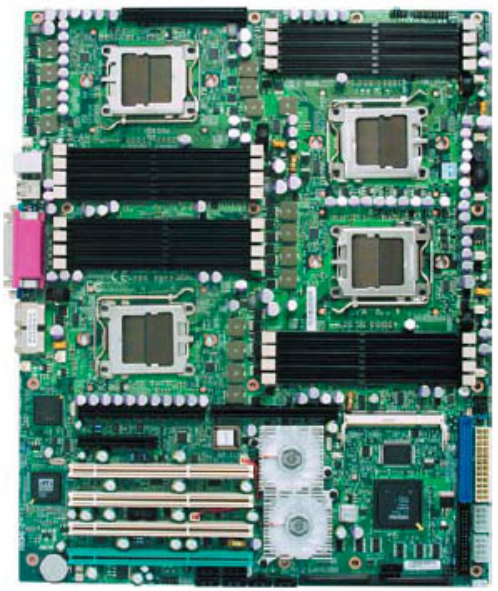
- L'OS est chargé du lancement des processus.
- L'OS attribue à chaque processus un identifiant nommé `pid` (process id).
- L'OS attribue un environnement mémoire au processus.
- L'OS contrôle l'exécution d'un processus, il peut l'interrompre, l'arrêter, le faire continuer, etc.
- Un OS multitâche est capable d'entrelacer l'exécution de ses processus en fonction de priorités (scheduling, préemption, commutation de contexte). On peut donc exécuter plusieurs processus sur un même processeur.

Table des matières

Architectures actuelles

Multi-processeurs

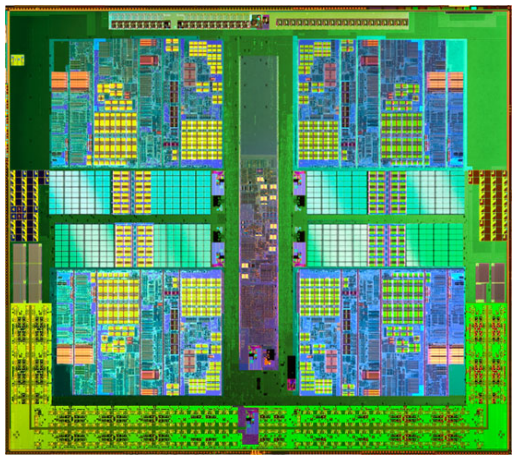
Plusieurs processeurs sont présents sur une même carte mère.



Architectures actuelles

Multi-coeurs

Chaque processeur contient plusieurs coeurs qui peuvent partager du cache.



Architectures actuelles

Hyper-threading (SMT)

Le processeur physique comporte plusieurs processeurs logiques chacun disposant de ses propres registres mais le pipeline, le cache et le bus sont partagés.

Table des matières

Enjeux

Multitâche

Effectuer plusieurs tâches sur une machine !

- Utilisation bureautique : écouter de la musique en surfant sur le web, ...
- Utilisation serveur : Gérer plusieurs serveurs sur une machine, plusieurs machines virtuelles, ...

Enjeux

Performance

Répartir les calculs sur plusieurs processeurs pour augmenter les performances.

Performance

Il est de plus en plus difficile d'augmenter la fréquence des processeurs (limite physique et coût). L'augmentation des performances passe donc aujourd'hui par l'utilisation de plusieurs processeurs.

Enjeux


JAGUAR

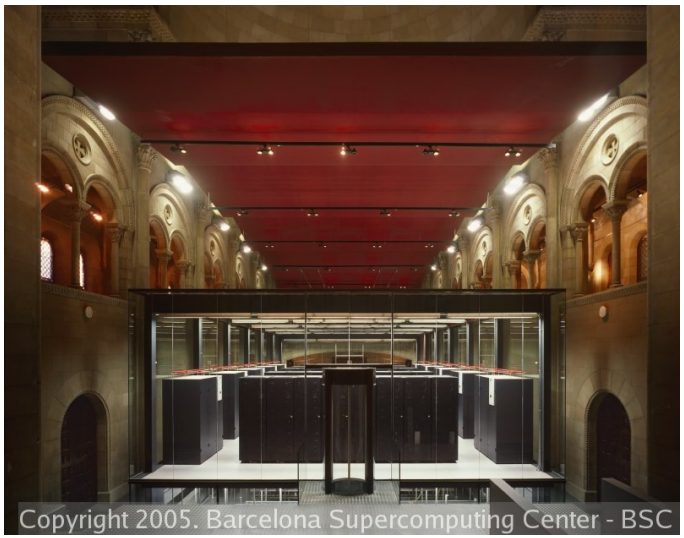

Copyright 2005. Barcelona Supercomputing Center - BSC

Table des matières

Table des matières

Programme exécutable

Construction d'un programme exécutable

cf. TD Architecture : Processus de compilation.

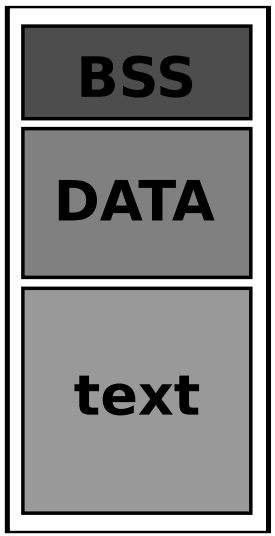
Programme exécutable

Organisation

Un processus est constitué de plusieurs **segments** :

- **text** : contient le code exécutable et les constantes
- **data** : contient les variables statiques et globales :
  - initialisées (**DATA**)
  - non initialisées (**BSS** : Block Started by Symbol)

Programme exécutable



programme

Programme exécutable

Exemple hello1.c

```
#include <stdio.h>
int main() {
   printf("Hello World\n");
   return 0;
}
```

size hello1

```
text       data     bss     dec     hex filename
 882        264       8    1154     482 hello1
```

Programme exécutable

Exemple hello2.c

```
#include <stdio.h>
int main() {
   printf("Hello World!\n");
   return 0;
}
```

size hello2

```
text       data     bss     dec     hex filename
 883        264       8    1155     483 hello2
```

Programme exécutable

Conclusion

Les chaînes constantes sont stockées dans le segment avec le code dans le segment **text** !

Autres variables

Où sont stockées les autres variables ?

Programme exécutable

Exemple hello3.c

```
#include <stdio.h>
int i;
int main() {
   printf("Hello World!\n");
   return 0;
}
```

size hello3

```
text       data     bss     dec     hex filename
 883        264      12    1159     487 hello3
```

Programme exécutable

Exemple hello4.c

```
#include <stdio.h>
int i=5;
int main() {
   printf("Hello World!\n");
   return 0;
}
```

size hello4

```
text       data     bss     dec     hex filename
 883        268       8    1159     487 hello4
```

Programme exécutable

Conclusion

Les variables globales sont stockées dans les segments :

- **DATA** si elles sont initialisées
- **BSS** si elles ne sont pas initialisées

Remarque : les variables initialisées explicitement à 0 sont considérées comme non initialisées.

Programme exécutable

Exemple hello5.c

```
#include <stdio.h>
const int i=5;
int main() {
   printf("Hello World!\n");
   return 0;
}
```

size hello5

```
text       data     bss     dec     hex filename
 887        264       8    1159     487 hello4
```

Programme exécutable

Conclusion

Les constantes même définies comme des variables sont stockées dans le segment **text** !

Programme exécutable

Exemple prog1.c

```
int var1 = 9, var2 = 0, var3;
const int var4 = 5;
int main() {
  int var5;
}
```

Programme exécutable

```
objdump -x prog1 | grep var
```

```
08048460 g O .rodata 00000004 var4
0804a010 g O .data   00000004 var1
0804a01c g O .bss    00000004 var2
0804a020 g O .bss    00000004 var3
```

Programme exécutable

var5 ? ? ?

Mais où est passée `var5` ?

Table des matières

Chargement du programme en mémoire

Objectif

Placer le programme exécutable, stocké sur le disque, dans la mémoire physique pour l'exécuter. Le chargeur de l'OS doit également ajouter des segments pour gérer entre autre la pile et le tas.

Mémoire réelle

La mémoire réelle, ou physique, correspond à la quantité de mémoire installée sur le système. Elle peut être vue comme un tableau d'octets.

Chargement du programme en mémoire

Attention !

Sous Linux le processus ne s'exécute pas en **mémoire réelle** mais en **mémoire virtuelle**.

Chargement du programme en mémoire

Mémoire virtuelle et pagination

- Sous Linux, tous les programmes s'exécutent en mémoire virtuelle et leurs segments sont placés au chargement dans des **pages** de la mémoire virtuelle.
- Les pages en mémoire virtuelle sont mises en correspondance avec les pages de mémoire réelle, appelées également **cases**, par une **table des pages** propre à chaque processus.
- Ce processus est implicite et est effectué par l'OS.
- Sous Linux la taille d'une page est de 4KO.

Chargement du programme en mémoire

Avantages sur la mémoire réelle

- Abstraction de l'implantation mémoire.
- Pas d'accès direct à la mémoire réelle : protection par l'intermédiaire de l'OS.
- Partage simplifié de la mémoire réelle par plusieurs programmes.
- Optimisation de l'utilisation mémoire : les pages du processus en cours d'exécution sont dans la mémoire centrale.
- Possibilité d'utiliser un espace **swap**.
- Espace d'adressage jusqu'à 3GO par processus pour les OS 32bits !

Chargement du programme en mémoire

Processus de chargement

1. Diviser le programme en pages
2. Passage d'un adressage linéaire à un adressage en mémoire virtuelle.
3. Vérification de la disponibilité de cases en mémoire réelle.
4. Chargement des pages utilisées dans les cases disponibles.

Chargement du programme en mémoire

Table des cases

La table des cases contient des informations sur l'état des cases en mémoire réelle :

- soit vide,
- soit le numéro du processus et de la page de ce dernier stockée dans la case.

Chargement du programme en mémoire

Table des pages d'un processus

La table des pages d'un processus contient les champs suivants :

- le numéro de la page
- la date de chargement de la page,
- la date de dernier accès ;
- un bit indiquant si la page est présente en mémoire réelle,
- un bit indiquant si la page a été modifiée en mémoire réelle.
- le numéro de case physique le cas échéant.

Chargement du programme en mémoire

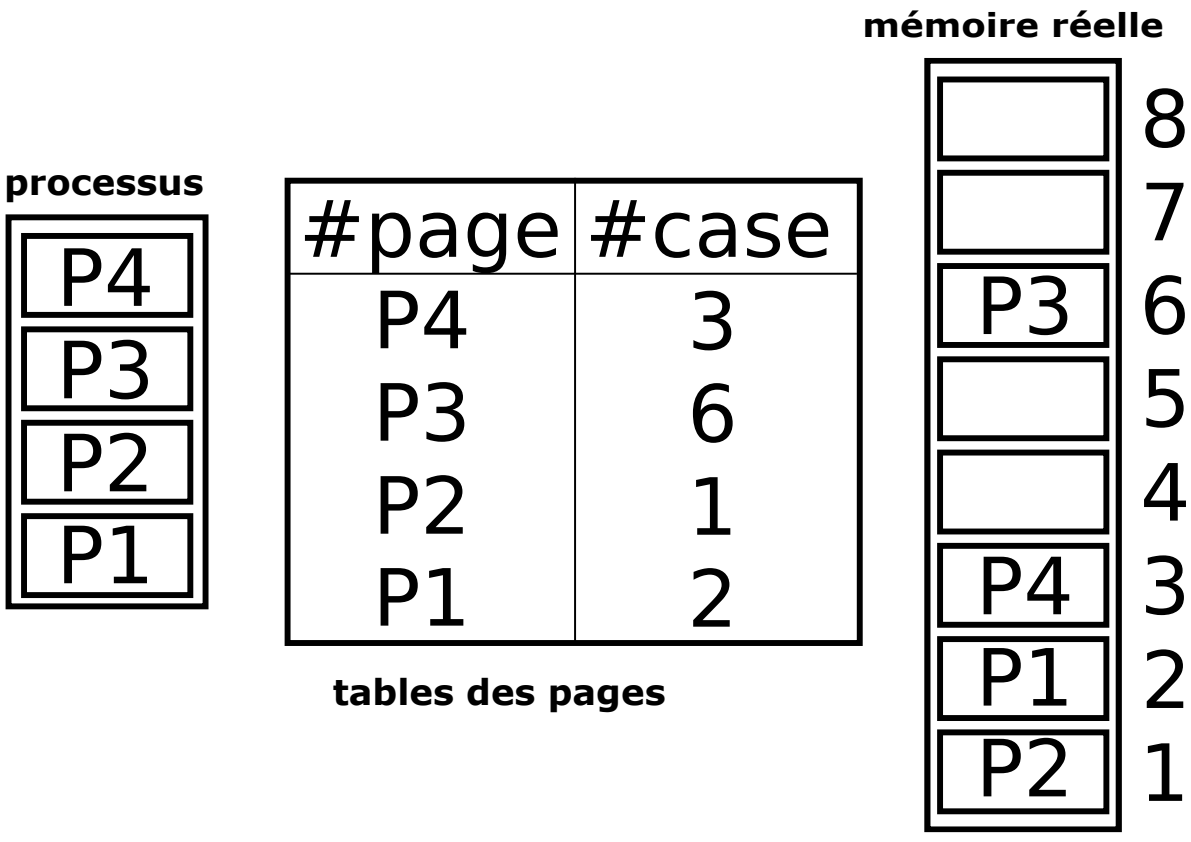


Table des matières

Processus en mémoire

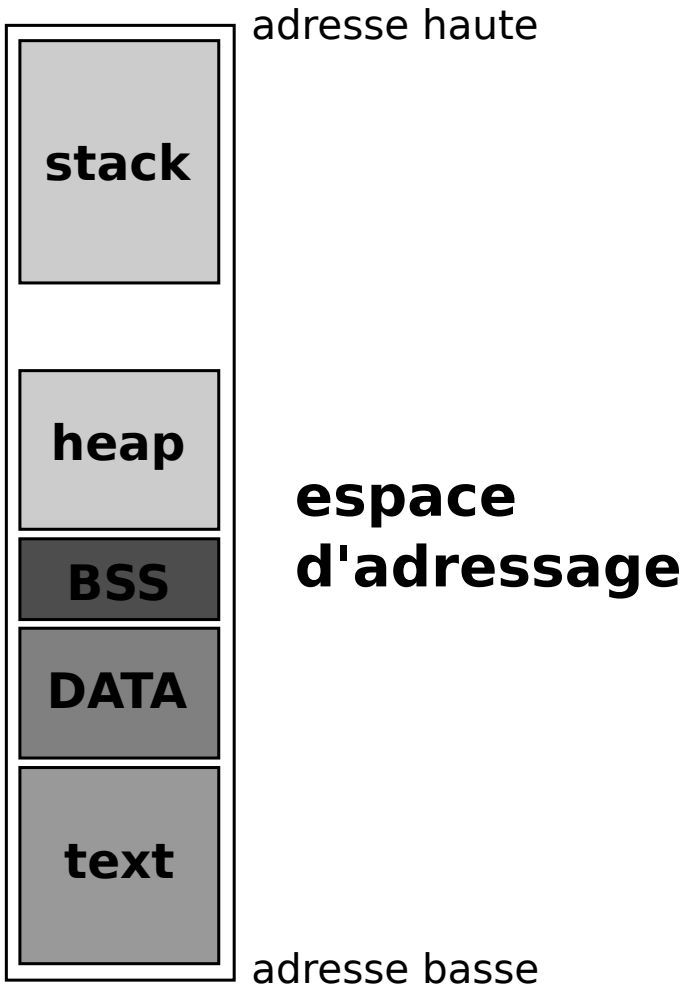
Segmentation

Un processus est constitué de plusieurs **segments** :

- **text** : contient le code exécutable et les constantes
- **data** : contient les variables statiques et globales :
  - initialisées (**DATA**)
  - non initialisées (**BSS** : Block Started by Symbol)

  ainsi que le **heap** (tas) pour les données alouées dynamiquement (`malloc` en `C`)
- **stack** (pile) : contient les variables locales

Processus en mémoire



Processus en mémoire

Evolution de la taille du processus en mémoire

- les segments **text**, **DATA** et **BSS** ont une taille constante en mémoire. Visualisation de la taille des segments **text**, **DATA** et **BSS** par la commande : `size` *progname*
- la taille du segment **heap** évolue en fonction des allocations dynamiques effectuées (`new` & `malloc`)
- la taille du segment **stack** évolue en fonction de l'appel aux fonctions et des variables locales contenues dans ces fonctions.

Processus en mémoire

Segments et pages

- Un segment est une zone contiguë de la mémoire virtuelle qui peut être partagée et protégée : **notion de droits**.
- Un segment est divisé en pages de 4KO.
- Visualisation des segments d'un processus en mémoire virtuelle par la commande : `cat /proc/`*pid*`/maps`

Processus en mémoire

démo

Table des matières

Introduction

Services fournis par l'OS

- Informations sur les processus
- Etats des processus
- Contrôle des processus
- Ordonnancement

Table des matières

Informations

`ps`

- liste des processus
- propriétaire de chaque processus
- `pid` et `parent pid` des processus
- ...

Utilisation

- `-e` : tous les processus
- `axjf` : arborescence des processus
- `-u` *username* : processus d'un utilisateur

Plus d'options avec `man ps` !

Table des matières

Etats des processus

top ou htop

- running
- sleeping
- stopped
- zombie !

Table des matières

Signaux

`kill`

- `KILL`
- `CONT`
- `STOP`
- `...`

`man kill` pour plus d'infos.

Exemple

`kill -9 -1` : supprime tous les processus possibles (en fonction des droits).

Table des matières

Scheduling

Ordonnanceur (Scheduler)

- Partie de l'OS chargée du contrôle de l'exécution des processus.
- Sur un système monoprocesseur, un seul processus peut être exécuté à un instant donné
- Sur un système à N processeurs, N processus peuvent être exécutés simultanément
- Notion de priorité : certains processus sont plus prioritaires que d'autres
- Notion d'affinité : le scheduler choisit le processeur sur lequel exécuter un processus

Scheduling

Principe

- Structure de données contenant les identifiants les processus et des informations sur ceux-ci (Tableau de listes de processus classé par priorité).
- Le scheduler commence par exécuter les processus de la liste de plus haute priorité, puis passe à la liste suivante.
- Une fois tous les processus traités, le scheduler recommence le traitement.
- Plus un processus est prioritaire, plus le scheduler lui attribue du temps d'exécution.

Scheduling

Priorité

La priorité d'un processus peut être déterminée de 2 manières :

- par le scheduler, qui "observe" chaque processus
- par l'utilisateur via la commande `nice`

Scheduling

`nice`

- -20, priorité la plus favorable à 19.
- Par défaut priorité à 10

Table des matières

Processus : Introduction

Au commencement...

- un seul processus `init`
- `init` lance d'autres processus et ainsi de suite
- les processus forment ainsi une arborescence
- chaque processus possède ainsi un processus père

Destruction du père

Un processus dont le père est détruit est "adopté" par le processus `init`

Processus : Introduction

Héritage

Chaque processus possède les informations suivantes :

- son identifiant `pid`
- l'identifiant de son processus père `ppid`
- un propriétaire
- un héritage de l'environnement du processus père

Table des matières

Processus : Création

`fork`

- La création d'un processus est effectuée par l'appel à la méthode : pid_t fork()
- Cet appel provoque la création d'une copie du processus père (mémoire virtuelle).
- Les processus père et fils ne diffèrent que par la valeur de retour de la fonction `fork`.
- Tous les processus, sauf `init`, sont créés par des appels à `fork`.
- Les processus père et fils continuent leur exécution juste après l'appel à `fork`.

Processus : Création

Valeurs de retour pour `fork`

- -1 : si le processus ne peut être "forké"
- 0 : pour le processus fils
- `pid du fils` : pour le processus père

Différenciation

On utilise la valeur de retour de `fork` pour différencier les processus père et fils.

Processus : Création

pid et ppid

- pid_t getpid() : récupération du pid
- pid_t getppid() : récupération du ppid

Processus : Création

```c
#include <stdio.h>
#include <unistd.h>

int main() {

   char c;
   pid_t ret = fork();
   if(ret) {
      printf("Processus %d pere du processus %d\n",
             getpid(), ret);
   }
   else {
      printf("Processus %d fils du processus %d\n",
             getpid(), getppid());
   }
   scanf("%c", &c);
   return 0;

}
```

Processus : Création

Exécution d'un code différent

- L'appel à `fork` provoque la duplication du code du processus père pour la création du processus fils.
- On souhaite créer un processus fils disposant d'un code différent de celui du processus père.

Processus : Création

Famille de fonction exec()

- **int** execl(**const char** *path, **const char** *arg, ...);
- **int** execlp(**const char** * file , **const char** *arg, ...);
- **int** execle(**const char** *path , **const char** *arg , ..., **char** * **const** envp[]);
- **int** execv(**const char** *path, **char** ***const** argv []);
- **int** execvp(**const char** * file , **char** ***const** argv []);

Processus : Création

```c
#include <stdio.h>
#include <unistd.h>
int main() {
   char c;
   pid_t ret = fork();
   if(ret) {
      printf("Proc. %d pere du proc. %d\n", getpid(), ret);
      wait();
   }
   else {
      printf("Proc. %d fils du proc. %d\n", getpid(), getppid());
      execl("/bin/ls", "ls", "-l", NULL);
      printf("Erreur appel execl\n");
   }
   return 0;
}
```

Processus : Création

Remarques

- L'appel à `exec` provoque le remplacement du code pour le processus fils par celui du programme appelé.
- Le code du processus fils placé après l'appel à `exec` n'est donc jamais exécuté **sauf** si l'appel à `exec` échoue.

Processus : Création

Ordre d'exécution

- Les codes du processus père et des processus fils ne sont plus exécutés dans l'ordre d'écriture.
- Un même code peut être exécuté par plusieurs processus.
- L'exécution des processus est entrelacée par le scheduler, l'exécution d'un même programme peut donc provoquer des affichages/résultats différents.

Section suivante...

Besoin de synchronisation !

Table des matières

Processus : Synchronisation

Attente de processus fils

3 fonctions sont à disposition pour surveiller le changement d'état de processus fils (terminaison, pause, continuation) :

- `pid_t wait(int* status)`
- `pid_t waitpid(pid_t pid, int *status, int options)`
- `pid_t waitid(idtype_t idtype, id_t id, siginfo_t *infop, int options)`

Processus : Synchronisation

Sans appel à `wait`

2 cas possibles si aucune synchronisation entre le processus père et les processus fils :

1. le processus père se termine avant ses processus fils
2. le(s) processus fils se termine(nt) avant le processus père

Processus : Synchronisation

cas 1 : adoption

Si un processus père se termine avant ses processus fils, ceux-ci sont adopter par le processus `init`.

Observation

`ps -edf` : processus adoptés ont un `ppid` dont la valeur passe à 1 après la terminaison du père.

Processus : Synchronisation

```c
#include <stdio.h>
#include <unistd.h>

int main() {

   char c;
   pid_t ret = fork();
   if(ret) {
      printf("Processus %d pere du processus %d\n",
             getpid(), ret);
      sleep(10);
   }
   else {
      printf("Processus %d fils du processus %d\n",
             getpid(), getppid());
      sleep(20);
   }
   return 0;

}
```

Processus : Synchronisation

cas 2 : processus Zombie

Si un processus fils n'est pas attendu il reste en état zombie :

- jusqu'à la fin du processus père si aucun `wait` du père,
- ou jusqu'à un appel à `wait` du père récupérant la fin du processus fils.

Observation

`ps aux` : processus zombies identifiés par `Z+`

Processus : Synchronisation

```c
#include <stdio.h>
#include <unistd.h>

int main() {

   char c;
   pid_t ret = fork();
   if(ret) {
      printf("Processus %d pere du processus %d\n",
             getpid(), ret);
      sleep(10);
   }
   else {
      printf("Processus %d fils du processus %d\n",
             getpid(), getppid());
   }
   return 0;

}
```

Processus : Synchronisation

Remarques sur la synchronisation

- Ne jamais se baser sur des a priori concernant l'ordre d'exécution des processus.
- L'appel à `sleep` ne doit jamais être utilisé pour synchroniser des processus.
- La synchronisation doit se faire à l'aide de fonctions de synchronisation adéquates comme `wait`.

Processus : Synchronisation

Avec `wait`

- L'appel à `wait` est effectué par le processus père.
- `wait` provoque une attente par le processus père de la terminaison **d'un** processus fils.
- Pour attendre n processus fils il faut n appels à `wait`.

Processus : Synchronisation

```c
#include <stdio.h>
#include <unistd.h>

int main() {

   char c;
   pid_t ret = fork();
   if(ret) {
      printf("Processus %d pere du processus %d\n",
             getpid(), ret);
      wait(); // attente du processus fils.
   }
   else {
      printf("Processus %d fils du processus %d\n",
             getpid(), getppid());
      scanf("%s", &c);
   }
   return 0;

}
```

Processus : Synchronisation

```c
#include <stdio.h>
#include <unistd.h>
int main() {
  char c;
  pid_t ret = fork();
  if(ret) {
    printf("Proc. %d pere du proc. %d\n", getpid(), ret);
    pid_t ret2 = fork();
    if(ret2) {
      printf("Je suis le pere\n");
      wait(); wait(); // 2 appels a wait
    }
    else { printf("Je suis le 2eme fils"); }
  }
  else {
    printf("Proc. %d fils du proc. %d\n", getpid(), getppid());
    scanf("%s", &c);
  }
  return 0;
}
```

Processus : Synchronisation

Remarques

- Un appel à `wait` récupère la terminaison d'un des processus fils.
- Si plusieurs processus fils sont terminés, le système en choisi un arbitrairement.

Ordre de terminaison

- Dans de nombreux cas l'ordre de terminaison importe !
- Utilisation de `pid_t waitpid(pid_t pid, int *status, int options)`

Processus : Synchronisation

Appel à `waitpid`

Passage d'un paramètre `pid` permettant de spécifier le processus fils à attendre :

- < -1 Attente d'un processus fils dont le groupid est `-pid` (voir `setpgid()`).
- -1 Attente d'un processus fils (semblable à `wait`).
- 0 Attente d'un processus fils du même groupe que le processus père.
- > 0 Attente du processus fils `pid`.

Processus : Synchronisation

```c
#include <stdio.h>
#include <unistd.h>
int main() {
  char c;
  pid_t ret = fork();
  if(ret) {
    printf("Proc. %d pere du proc. %d\n", getpid(), ret);
    pid_t ret2 = fork();
    if(ret2) {
      printf("Je suis le pere\n");
      waitpid(ret); waitpid(ret2); // 2 appels a wait
    }
    else { printf("Je suis le 2eme fils"); }
  }
  else {
    printf("Proc. %d fils du proc. %d\n", getpid(), getppid());
    scanf("%s", &c);
  }
  return 0;
}
```

Processus : Synchronisation

Retour du statut des processus fils

Les fonctions `wait` et `waitpid` permettent au processus père de récupérer le statut d'un processus fils.

Côté processus fils

void exit(**int** status)

Côté processus père

```
int status;
wait(&status);
```

Processus : Synchronisation

Traitement de la valeur retour `status`

La valeur `status` peut être traitée par les fonctions suivantes :

WIFEXITED(status) : vrai si le processus fils s'est terminé normalement

WEXITSTATUS(status) : code retour passé à `exit`

D'autres fonctions sont disponibles pour traiter les cas ou l'état du processus fils est modifié par des signaux : `man wait`.

Processus : Synchronisation

```c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/wait.h>

int main() {
  int res=0;
  pid_t ret = fork();
  if(ret) {
    printf("Processus pere\n");
    wait(&res);
    printf("Valeur renv par fils = %d\n", WEXITSTATUS(res));
  }
  else {
    printf("Processus fils\n");
    exit(12);
  }
  return 0;
}
```

Processus : Synchronisation

Les signaux

Les signaux sont des interruptions logicielles à destination d'un processus, par exemple pour signaler une erreur. Le processus recevant un signal possède une fonction pour traiter celui-ci.
Ce système de signaux peut être détourné pour effectuer des synchronisations entre plusieurs processus en écrivant ses propres fonctions.

Processus : Synchronisation

Envoi d'un signal

L'envoi de signaux :

- au processus courant s'effectue par la fonction `raise`
- à un autre processus est effectué par la fonction `kill`

int kill (pid_t pid , **int** sig)

- `pid` : identifiant du processus destinataire
- `sig` : type de signal à envoyer

Processus : Synchronisation

Types de signaux

Les signaux sont numérotés de 0 à 31, quelques exemples :

- SIGHUP 1 : hangup
- SIGINT 2 : interruption
- SIGQUIT 3 : quit
- SIGILL 4 : instruction illégale
- SIGKILL 9 : hard kill
- SIGALRM 14 : alarme
- SIGCONT 19 : continuation d'un processus
- SIGCHLD 20 : vers le processus parent lorsqu'un processus fils s'arrête

Processus : Synchronisation

```c
#include <stdio.h>
#include <unistd.h>
#include <signal.h>

int main() {
  char c;
  pid_t ret = fork();
  if(ret) { // Section du processus parent
    scanf("%c", &c);
    kill(ret, SIGKILL); // Envoi du signal SIGKILL
    sleep(10);
  }
  else { // Section du processus fils
    while(1); // Attente active
  }
  return 0;
}
```

Processus : Synchronisation

Attente active (polling)

Le processus en attente du signal reste actif mais ne fait rien dans l'attente d'un signal. Cette technique n'est pas efficace car elle consomme des ressources processeur inutilement.

Attente passive

Il est possible de mettre un processus en sommeil dans l'attente d'un signal à l'aide de la fonction `pause`.

int pause()

Processus : Synchronisation

```c
#include <stdio.h>
#include <unistd.h>
#include <signal.h>

int main() {
  char c;
  pid_t ret = fork();
  if(ret) {
    scanf("%c", &c);
    kill(ret, SIGKILL);
    sleep(10);
  }
  else {
    pause();
  }
  return 0;
}
```

Processus : Synchronisation

Autre exemple : Contrôle du processus fils

Mise en pause et relance du processus fils à l'aide, respectivement, des signaux SIGSTOP et SIGCONT.

Processus : Synchronisation

```c
#include <stdio.h>
#include <unistd.h>
#include <signal.h>

int main() {
  char c;
  pid_t ret = fork();
  if(ret) {
    sleep(10);
    kill(ret, SIGSTOP);
    sleep(10);
    kill(ret, SIGCONT);
    sleep(10);
    kill(ret, SIGKILL);
  }
  else {
    while(1);
  }
  return 0;
}
```

Processus : Synchronisation

Réception d'un signal et traitement

Il est possible de spécifier sa propre fonction à exécuter lors de la réception d'un signal à l'aide de la fonction `signal`.

signal (sig , **void** (∗func)())

- `sig` : signal à traiter
- `void (*func)()` : pointeur vers la fonction de traitement

Processus : Synchronisation

Pointeur sur fonction : déclaration

typeretour (∗nompointeur)(arguments)

Pointeur sur fonction : appel

(∗nomdupointeur)(arguments)

Processus : Synchronisation

```c
#include <stdio.h>
#include <unistd.h>
#include <signal.h>

void onint() {printf("INTERRUPTION RECUE\n");}

int main() {
  char c;
  pid_t ret = fork();
  if(ret) {
    scanf("%c", &c);
    kill(ret, SIGINT);
    sleep(10);
  }
  else {
    signal(SIGINT, onint);
    pause();
  }
  return 0;
}
```

Processus : Synchronisation

```c
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
void onint() {signal(SIGINT, onint);printf("INT\n");}
void onquit() {printf("QUIT\n"); exit(0);}

int main() {
  char c;
  pid_t ret = fork();
  if(ret) {
    scanf("%c", &c);
    switch(c) {
    case 'i' : kill(ret, SIGINT); break;
    case 'q' : kill(ret, SIGQUIT); break;}
    sleep(5);}
  else {
    signal(SIGINT, onint);
    signal(SIGQUIT, onquit);
    pause();}
  return 0;
}
```

Table des matières

Processus : Communication

Statut

Utilisation de la valeur de retour pour communiquer entre le processus père et les processus fils.

Limitations

On ne peut récupérer qu'un octet !

Processus : Communication

Moyens à dispositions

- Fichiers
- Pipes
- Messages
- Mémoire partagée
- Sockets
- ...

Processus : Communication : Fichiers

Communication par les fichiers

Utilisation de fichiers pour faire communiquer des processus.

Processus : Communication : Fichiers

Rédacteur/Lecteur

- Le processus fils écrit dans un fichier
- Le processus père lit dans le même fichier

Processus : Communication : Fichiers

```c
#include <stdio.h>
#include <stdlib.h>

int main() {
  pid_t pid;
  int status;
  char buffer[10];
  FILE* file = fopen("test", "w+");
  if(fork()) {
    wait();
    fseek(file, 0, SEEK_SET);
    fread(buffer, 1, 10, file);
    printf("%s", buffer);
    fclose(file);
  }
  else
    fwrite("Hello!\n", 1, 7, file);
  return 0;
}
```

Processus : Communication : Fichiers

Type

`FILE*` : pointeur sur un fichier

Fonctions

- `FILE* fopen("cheminverslefichier", "r|r+|w|w+|a|a+")`
- `int fseek(FILE*, offset, SEEK_{SET|CUR|END})`
- `int fclose(FILE*)`

Processus : Communication : Fichiers

Inconvénients

- Pas très efficace : accès disque
- Pas très sûr : accès concurrents

Processus : Communication : Pipes

Pipes

Idée : Utiliser le système de pipe pour faire communiquer les processus.

Rappels

```
ls | sort
```

Processus : Communication : Pipes

Fonctions

- `popen`
- `pipe`

Processus : Communication : Pipes

popen

`FILE* popen("command", "r|w")`

1. lance la commande dans un processus
2. crée un pipe et retourne un pointeur vers celui-ci
3. lit ou écrit à partir du pointeur

pclose

`int pclose(FILE*)`

1. attente de la fin du processus associé
2. retourne le statut du processus

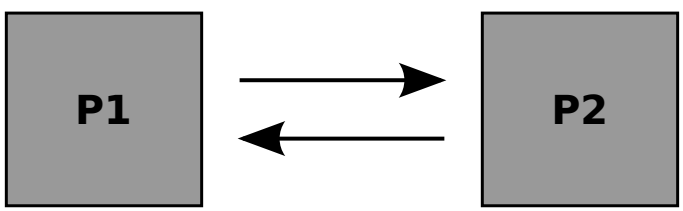
Processus : Communication : Pipes

```c
#include <stdio.h>


int main() {

   FILE* file;
   char buffer[10];

   file = popen("date", "r");

   while(fgets(buffer, 10, file)) {
      printf("%s", buffer);
   }

   pclose(file);

   return 0;

}
```

Processus : Communication : Pipes

`pipe`

Idée : créer 2 canaux de communication unidirectionnels entre 2 processus.



Processus : Communication : Pipes

Fonctions

1. `int pipe(int[2])`
2. - `ssize_t write(int, const void*, size_t)`
   - `ssize_t read(int, void*, size_t)`
3. `int close(int)`

Processus : Communication : Pipes

```c
#include <stdio.h>

int main() {
  int pipes[2];
  pipe(pipes);
  char buffer[10];
  if(fork()) {
    close(pipes[1]);
    while(read(pipes[0], buffer, 10))
      printf("%s", buffer);
    close(pipes[0]);
  }
  else {
    close(pipes[0]);
    write(pipes[1], "Hello!\n", 7);
    close(pipes[1]);
  }
  return 0;
}
```

Processus : Communication : Pipes

```c
#include <stdio.h>

int main() {
  int pipes[2];
  pipe(pipes);
  int i = 0;
  if(fork()) {
    close(pipes[1]);
    read(pipes[0], &i, 4);
    printf("%i\n", i);
    close(pipes[0]);
  }
  else {
    i = 5;
    close(pipes[0]);
    write(pipes[1], &i, 4);
    close(pipes[1]);
  }
  return 0;
}
```

Processus : Communication : Messages

Files de messages

Passage de messages entre processus par un système de files de messages.

Processus : Communication : Messages

Fonctions

- `mqd_t mq_open(const char* name, int oflag, {mode_t mode, struct mq_attr *attr})`
- `mqd_t mq_send(mqd_t mqdes, const char* msg_ptr, size_t msg_len, unsigned msg_prio)`
- `ssize_t mq_receive(mqd_t mqdes, char* msg_ptr, size_t msg_len, unsigned* msg_prio)`
- `mqd_t mq_close(mqd_t mqdes)`

Processus : Communication : Messages

```c
#include <stdio.h>
#include <mqueue.h>
#include <fcntl.h>
#include <sys/stat.h>
int main() {
   mqd_t mq;
   mq = mq_open("/mq5", O_CREAT | O_RDWR, 0600, NULL);
   mq_send(mq, "abcd", 4, 0);
   return 0;
}
```

Processus : Communication : Messages

```c
#include <stdio.h>
#include <mqueue.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <stdlib.h>
int main() {
   mqd_t mq;
   void* buffer;
   struct mq_attr attr;
   mq = mq_open("/mq5", O_RDONLY);
   mq_getattr(mq, &attr);
   buffer = malloc(attr.mq_msgsize);
   mq_receive(mq, buffer, attr.mq_msgsize, NULL);
   printf("%s", (char*)buffer);
   return 0;
}
```

Processus : Communication : Messages

```c
#include <stdio.h>
#include <mqueue.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <stdlib.h>
int main() {
   mqd_t mq;
   void* buffer;
   struct mq_attr attr;
   mq = mq_open("/mq5", O_CREAT | O_RDWR, 0600, NULL);
   if(fork())
        mq_send(mq, "abcd", 4, 0);
   else {
      mq_getattr(mq, &attr);
      buffer = malloc(attr.mq_msgsize);
      mq_receive(mq, buffer, attr.mq_msgsize, NULL);
      printf("%s\n", (char*)buffer);
   }
   return 0;
}
```

Processus : Communication : Mémoire partagée

Mémoire partagée

Créer un bloc de mémoire du processus père et le marquer comme accessible par ses processus fils. A l'appel de `fork()`, ce bloc ne sera pas dupliqué mais commun aux processus.

Processus : Communication : Mémoire partagée

mmap

```
void* mmap(void* addr, size_t length, int prot, int flags, int fd, off_t offset)
```

1. addr : adresse de début du segment de mémoire partagée
2. length : longueur du segment souhaité
3. prot : protection de l'accès (lecture, écriture, ...)
4. flags : type de segment (voir man)
5. fd : descripteur de fichier
6. offset : offset dans le fichier pour l'initialisation des données

- retourne un pointeur vers le début du segment partagé

Processus : Communication : Mémoire partagée

munmap

`int munmap(void* addr, size_t length)`

Supprime le partage du segment passé en paramètre.

Processus : Communication : Mémoire partagée

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/mman.h>
int main() {
   pid_t res;
   int* mem = 0;
   mem = (int*)mmap(NULL, 4, PROT_READ | PROT_WRITE,
                    MAP_SHARED | MAP_ANONYMOUS, -1, 0);
   if(mem) {
      res = fork();
      if(res) {
         wait();
         printf("%i\n", *mem);
         munmap(NULL, 4);
      }
      else
         *mem = 6;
   }
   return 0;
}
```

Processus : Communication : Sockets

Sockets

Etablir des communications point-à-point entre plusieurs processus.
Utilisation des sockets en période 4...

Table des matières

Introduction

Rappels sur les processus

- La création d'un nouveau processus implique la duplication de l'espace d'adressage.
- Les processus sont indépendants : données + pile d'exécution.
- Un seul fil d'exécution.

Introduction

Limitations

1. Pas de partage des informations simples (tubes, fichiers, mémoire partagée, réseau, ...).
2. Découpage d'un programme en plusieurs processus "gros grain".
3. Commutation entre les processus coûteuse.

Introduction

Threads

- Processus "légers" partageant l'espace d'adressage du processus père.
- Un fil d'exécution par thread.

Introduction

Un thread

1. 1 compteur d'instruction propre.
2. 1 pile d'exécution propre : variables locales privées.

Introduction

Avantages

- Commutation plus simple : mise à jour du compteur d'instruction + pointeurs sur pile d'exécution.
- Communication plus simple par accès à l'espace d'adressage partagée.
- Découpage des tâches à "grain fin".

Inconvénients

- Accès concurrents aux données partagées.
- Problèmes de terminaison.
- Problèmes d'interblocages.

Introduction

Programmation sous Linux

- Utilisation de la bibliothèque POSIX `pthread`.
- POSIX : Portable Operating System Interface for LinuX, ensemble de standards des systèmes UNIX.

Introduction

Header

#include <pthread.h>

Compilation

```
gcc -o ... -pthread
```

Table des matières

Création

Structure

pthread_t thread

Création

```
int   pthread_create(pthread_t* thread,
                     pthread_attr_t* attr,
                     void* (*start_routine)(void*),
                     void* arg);
```

Création

Arguments

1. pthread_t* thread : pointeur sur le thread
2. pthread_attr_t * attr : pointeur sur attributs
3. **void*** (* start_routine)(**void***) : pointeur sur la fonction à exécuter par le thread
4. **void*** arg : pointeur sur arguments de la fonction à exécuter

Valeur retournée

- 0 en cas de succès, id du nouveau thread dans `thread`.
- code erreur EAGAIN, pas assez de ressources.

Création

```c
#include <stdio.h>

void fct() {
  printf("fct\n");
}

int main() {
  void (*pf)() = &fct;
  (*pf)();
}
```

Création

```c
#include <stdio.h>

int fct(int a) {
  return a*2;
}

int main() {
  int (*pf)(int) = &fct;
  printf("%d\n", (*pf)(5));
}
```

Création

1er programme

1. déclaration d'une structure `thread`
2. appel à `pthread_create`

Création

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

void* fonction() {
   printf("Thread\n");
}

int main() {
   pthread_t thread;
   pthread_create(&thread, NULL, fonction, NULL);
   return 0;
}
```

Création

Terminaison

La terminaison du processus père entraÃ®ne la terminaison de ses threads !

Remède

Attente de la terminaison des threads par le processus père (à l'instar de `wait` pour les processus) :

int pthread_join (pthread_t thread , **void**** thread_return)

Création

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

void* fonction() {
   printf("Thread\n");
}

int main() {
   pthread_t thread;
   pthread_create(&thread, NULL, fonction, NULL);
   pthread_join(thread, NULL);
   return 0;
}
```

Création

Partage de l'espace d'adressage

- Les variables globales du processus père sont partagées par ses threads.
- Chaque thread possède ses propres variables locales (comme les fonctions).

Création

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

int A = 5; // variable globale
void* fonction() {
  int a = 7; // variable locale
  printf("locale : %d, globale : %d\n", a, A);
}
int main() {
  pthread_t thread;
  pthread_create(&thread, NULL, fonction, NULL);
  pthread_join(thread, NULL);
  printf("globale : %d\n", A);
  return 0;
}
```

Création

Accès concurrents aux données

Que se passe-t-il quand le processus et un thread accèdent à une même donnée ?

Création

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

int a = 5; // variable globale
void* fonction() {
   printf("Thread\n");
   a+=5;
}
int main() {
   pthread_t thread;
   pthread_create(&thread, NULL, fonction, NULL);
   printf("%d\n", a); // Quelle valeur ?
   pthread_join(thread, NULL);
   printf("%d\n", a); // et ici ?
   return 0;
}
```

Création

Indéterminisme

Différentes exécutions peuvent générer différents résultats !

Création

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

int a = 5; // variable globale
void* fonction() {
   printf("Thread\n");
   a+=5;
}
int main() {
   pthread_t thread;
   pthread_create(&thread, NULL, fonction, NULL);
   a*=2;
   pthread_join(thread, NULL);
   printf("%d\n", a); // et ici ?
   return 0;
}
```

Création

Création de multiples threads

Plusieurs threads peuvent exécuter :

- la même fonction
- des fonctions différentes

Création

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

void* fonction() {
  printf("Fonction\n");
}
int main() {
  pthread_t threads[2];
  pthread_create(&threads[0], NULL, fonction, NULL);
  pthread_create(&threads[1], NULL, fonction, NULL);
  pthread_join(threads[0], NULL);
  pthread_join(threads[1], NULL);
  return 0;
}
```

Création

```c
#include <pthread.h>
#include <stdio.h>
void* fonction1() {
   printf("Fonction1\n");
}
void* fonction2() {
   printf("Fonction2\n");
}
int main() {
   pthread_t threads[2];
   pthread_create(&threads[0], NULL, fonction1, NULL);
   pthread_create(&threads[1], NULL, fonction2, NULL);
   pthread_join(threads[0], NULL);
   pthread_join(threads[1], NULL);
   return 0;
}
```

Table des matières

Communication

Communication entres les threads

Uniquement par :

- des données globales (dans le segment `data`),
- des données dynamiques allouées dans le tas.

Variables locales

Chaque thread possède sa propre pile et donc ses variables locales, inaccessibles par les autres threads.

Communication

Passage d'arguments aux threads fils

La fonction `pthread_create` peut prendre en argument un pointeur vers un argument à passer au thread créé.

Cast

Attention à bien utiliser des pointeurs ! La taille d'un pointeur est dépendante de l'architecture.

Communication

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

void* fonction(void* arg) {
  printf("%d\n", *(unsigned int*)arg);
}

int main() {
  pthread_t thread;
  unsigned int value = 5;
  pthread_create(&thread, NULL, fonction, (void*)&value);
  pthread_join(thread, NULL);
  return 0;
}
```

Communication

```c
#include <pthread.h>
#include <stdio.h>
struct Arg { int value ; char* str ; };
void* fonction(void* arg) {
   printf("%d\n", ((struct Arg*)arg)->value);
   printf("%s\n", ((struct Arg*)arg)->str);
}
int main() {
   pthread_t thread;
   struct Arg arg;
   arg.value = 4;
   arg.str = "Hello!";
   pthread_create(&thread, NULL, fonction, (void*)&arg);
   pthread_join(thread, NULL);
   return 0;
}
```

Communication

```c
#include <pthread.h>
#include <stdio.h>
void* fonction(void* arg) {
  printf("%d\n", *(unsigned int*)arg);
}
int main() {
  pthread_t threads[10];
  unsigned int id;
  for(id = 0 ; id < 10 ; ++id)
    pthread_create(&threads[id], NULL,
                   fonction, (void*)&id);
  for(id = 0 ; id < 10 ; ++id)
    pthread_join(threads[id], NULL);
  return 0;
}
```

Communication

Accès concurrent au données partagées

Le thread parent peut modifier le contenu d'une variable passée à un thread fils avant que le processus fils ne le récupère !

Communication

```c
#include <pthread.h>
#include <stdio.h>
void* fonction(void* arg) {
  printf("%d\n", *(unsigned int*)arg);
}
int main() {
  pthread_t threads[10];
  unsigned int id;
  unsigned int ids[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
  for(id = 0 ; id < 10 ; ++id)
    pthread_create(&threads[id], NULL,
                   fonction, (void*)&ids[id]);
  for(id = 0 ; id < 4 ; ++id)
    pthread_join(threads[id], NULL);
  return 0;
}
```

Table des matières

Synchronisation

Entrelacements & Accès concurrents

Besoin de synchroniser les threads pour contrôler :

1. l'entrelacement des exécutions
2. les accès concurrents aux données partagées

Synchronisation

Cas d'utilisation

- structures de données : plusieurs threads ajoutent/suppriment des éléments dans une même liste.
- accès concurrents à des périphériques.
- base de données : plusieurs clients/threads accèdent à la même base.
- ...

Synchronisation

Primitives de synchronisation

- Mutex
- Condition variables
- Semaphores

Synchronisation

Mutex

- Abbréviation de "Mutual Exclusion".
- Permet de garantir que l'exécution d'un thread ne sera pas entrelacée avec d'autres threads partageant le "mutex".
- Mise en place d'une section critique.
- Le mutex peut être vu comme un jeton que seule un thread peut acquérir à la fois.

Synchronisation

Mutex : scénario

1. Déclaration du mutex
2. Initialisation du mutex
3. les threads tentent d'acquérir le mutex
4. 1 seul peut l'obtenir, les autres sont mis en attente
5. le thread possédant le mutex continue sont exécution
6. le thread possédant le mutex libère le mutex
7. 1 thread parmi ceux en attente est réveillé et récupère le mutex
8. ...

Synchronisation

Mutex : mise en oeuvre

1. Déclaration d'une variable de type mutex : `pthread_mutex_t`
2. Initialisation : `pthread_mutex_init(...)`
3. Capture du mutex : `pthread_mutex_lock(...)`
4. Libération du mutex : `pthread_mutex_unlock(...)`
5. Destruction : `pthread_mutex_destroy(...)`

Synchronisation

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

pthread_mutex_t mutex;

int main() {
  unsigned int ids[4] = {0, 1, 2, 3};
  pthread_t threads[4];
  pthread_mutex_init(&mutex, NULL);

  pthread_create(&threads[0],NULL, fct, (void*)&ids[0]);
  pthread_create(&threads[1],NULL, fct, (void*)&ids[1]);
  pthread_create(&threads[2],NULL, fct, (void*)&ids[2]);
  pthread_create(&threads[3],NULL, fct, (void*)&ids[3]);
  pthread_exit(NULL);
}
```

Synchronisation

```
void* fct(void* arg) {
  unsigned int id = *(unsigned int*)arg;
  unsigned int i;
  for(i = 0 ; i < 10 ; ++i) {
    pthread_mutex_lock(&mutex);
    printf("Starting %d\n", id);
    sleep(1);
    printf("Stopping %d\n", id);
    pthread_mutex_unlock(&mutex);
  }
}
```

Synchronisation

Condition variables

- Mise en attente d'un thread tant qu'une condition n'est pas remplie.
- Réveil du thread en attente par un autre thread.
- Utilisation combinée avec les mutex.

Synchronisation

Condition variables : scénario

1. Déclaration d'1 mutex + d'1 condition variable
2. Initialisation
3. Les threads tentent d'acquérir le mutex.
   - 1 thread va se mettre en attente sur la variable et débloquer le mutex.
   - 1 autre thread signalera que la condition est vérifiée au thread bloqué.
4. Destruction du mutex et de la condition variable.

Synchronisation

Condition variables : mise en oeuvre

1. Déclaration d'une condition variable : `pthread_cond_t`
2. Initialisation : `pthread_cond_init(...)`
3. Mise en attente : `phread_cond_wait(...)`
4. Envoi du signal : `phread_cond_signal(...)`
5. Destruction : `phread_cond_destroy(...)`

Synchronisation

```c
#include <pthread.h>
#include <stdio.h>
pthread_mutex_t mutex;
pthread_cond_t cond;
unsigned int count = 0;
int main() {
   pthread_t threads[4];
   pthread_mutex_init(&mutex, NULL);
   pthread_cond_init(&cond, NULL);
   pthread_create(&threads[0], NULL, compute, NULL);
   pthread_create(&threads[1], NULL, observer, NULL);
   pthread_join(threads[0], NULL);
   pthread_join(threads[1], NULL);
   pthread_mutex_destroy(&mutex);
   pthread_cond_destroy(&cond);
}
```

Synchronisation

```
void* observer() {

  pthread_mutex_lock(&mutex);

  printf("Observer has mutex!\n");

  if(count < 10)
    pthread_cond_wait(&cond, &mutex);

  printf("10 !\n");

  pthread_mutex_unlock(&mutex);

}
```

Synchronisation

```
void* compute(void* arg) {
  unsigned int i;
  for(i = 0 ; i < 20 ; ++i) {
    pthread_mutex_lock(&mutex);
    printf("Compute : inc %d\n", i);

    if(count == 10) {
      printf("count = 10\n");
      pthread_cond_signal(&cond);
    }
    ++count;
    pthread_mutex_unlock(&mutex);
    sleep(1);
  }
}
```

Synchronisation

Semaphores

- Mutex avec compteur.
- Le compteur est décrémenté chaque fois qu'un thread récupère le sémaphore.
- Lorsque le compteur est à zéro, les threads sont mis en attente.

Synchronisation

Semaphores : scénario

1. Déclaration d'un sémaphore.
2. Initialisation du sémaphore (compteur).
3. Les threads récupèrent le sémaphore tant que le compteur $>$ 0.
4. Si le compteur $==$ 0 les threads sont mis en attente.
5. Chaque fois qu'un thread libère le sémaphore, le compteur est incrémenté et un thread en attente est réveillé.

Synchronisation

Semaphores : mise en oeuvre

1. **#include** <semaphore.h>
2. Déclaration d'un sémaphore : `sem_t`
3. Initialisation : `sem_init(...)`
4. Capture : `sem_wait(...)`
5. Libération : `sem_post(...)`
6. Destruction : `sem_destroy(...)`

Synchronisation

```c
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
sem_t sem;
int main() {
  pthread_t threads[4];
  sem_init(&sem, 0, 2);
  unsigned int i;
  unsigned int ids[] = {0, 1, 2, 3};
  for(i = 0 ; i < 4 ; ++i)
    pthread_create(&threads[i], NULL,
                   fct, (void*)&ids[i]);
  for(i = 0 ; i < 4 ; ++i)
    pthread_join(threads[i], NULL);
  sem_destroy(&sem);
}
```

Synchronisation

```
void* fct(void* arg) {

  sem_wait(&sem);

  printf("Thread %d\n", *(unsigned int*)arg);
  sleep(4);

  sem_post(&sem);

}
```