Programmation Système sous Linux/Unix

Dr. Ing. Chiheb Ameur ABID

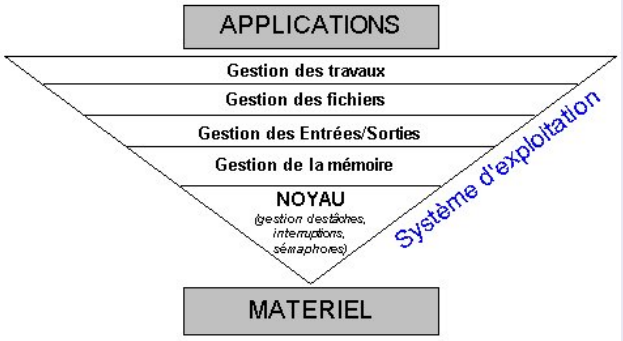
Contact: chiheb.abid@gmail.com

Mars 2023

Plan

Système d'exploitation ?

Système d'exploitation ?



La naissance d'Unix

La naissance d'Unix

- New Ken's System
  - 1969
  - En assembleur
  - Inspiré de Multics
- 1971 : réécriture en C
- 1975 : large distribution

La naissance d'Unix

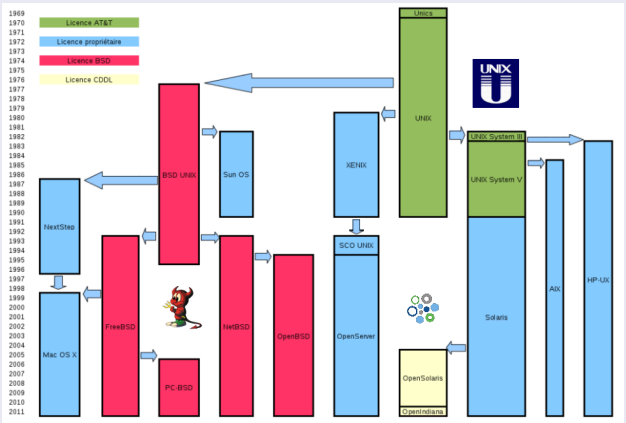
Dans les années 1970

- AT&T avait le monopole des télécommunications américaines
  - Des fonds disponibles pour divers domaines de recherche
- AT&T a développé Unix
  - Elle a offert UNIX gratuitement avec son code source
- D'autres organismes ont commencé à utiliser et modifier UNIX
  - BSD (Berkeley System Distribution)
- En 1982, AT&T a perdu son monopole
  - Bell labs a commencé par demander des licences pour l'utilisation de UNIX

AT&T Bell Laboratories

Famille Unix

Famille Unix



Les logiciels libres ?

Et un défaut dans une imprimante ! ! !

1. Première version
   - ✔ Pilote et code source disponible ->Problème corrigé
2. Seconde version
   - ✖ Pilote en binaire -> Impossible de résoudre le problème
   - ✖ Il faut attendre jusqu'à la mise à jour de la pilote du constructeur

Gnu is Not Unix

Gnu is Not Unix

- Lancé par Richard Stallman en 1983
  - Afin de développer un système d'exploitation entièrement libre
- Supporté par la FSF depuis 1985
  - Free Software Fondation
- Créer une suite complète de logiciels
  - Mais le noyau tarde à arriver
  - Projet Hurd, initié en 1990 mais toujours inabouti à ce jour

Linux

Linux, enfin

- Basé sur Minix, noyau UNIX pour les PC développé par Linus Trovalds
- Fonctionne avec des composants de GNU adaptés
- 26 août 1991 : annoncé sur le forum Usenet comp.os.minix
- 1992 : passage à la licence libre GNU GPL

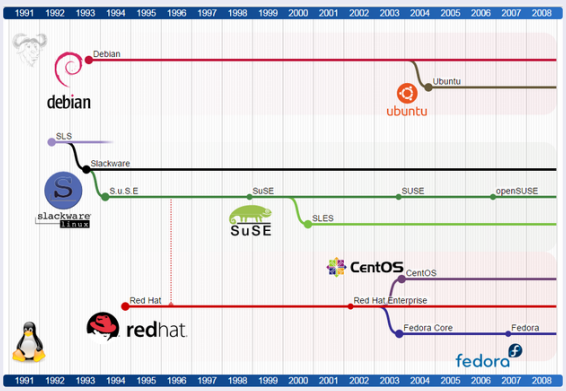
Linux : une philosophie

Linux : une philosophie

- Modifiable
- Partageable
- Adaptable
- Ludique
- Simple
- Répandu

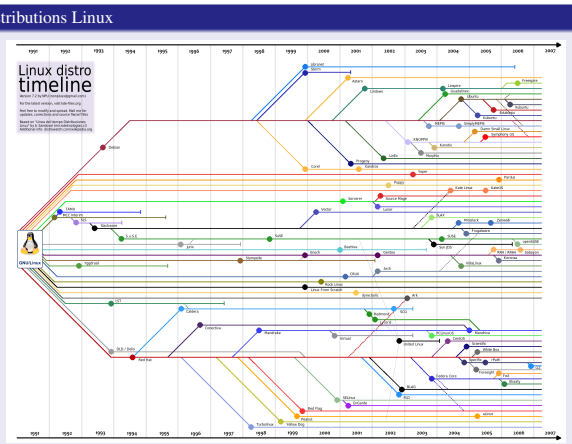
Les principales familles de distributions Linux

Les principales familles de distributions Linux



Les distributions Linux

Les distributions Linux



La norme POSIX ?

La norme POSIX ?

- Tous proviennent d'Unix, mais ils commençaient lentement à diverger.
- IEEE a lancé le POSIX spécifications en 1988, permettant interopérabilité entre les systèmes d'exploitation.
  - POSIX a évolué depuis (la spécification la plus récente en 2017).
- Tous les systèmes d'exploitation POSIX ont des caractéristiques spécifiques au-delà de POSIX, mais ils partagent la partie POSIX.
- POSIX et "Unix-like" sont parfois utilisé des manière interchangeable.

Plan

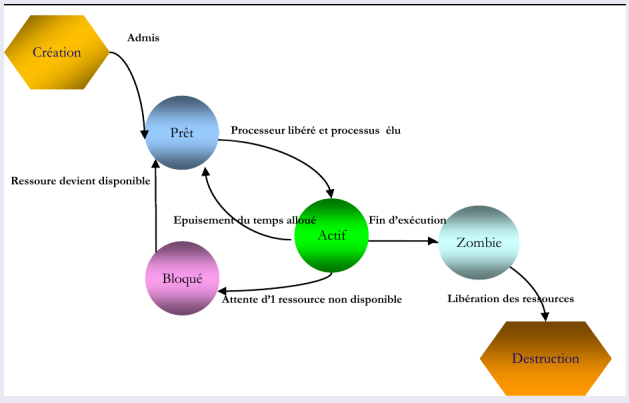
Notion de processus

Notion de processus

- Un processus est un programme encours d'exécution
- Permet de partager un même matériel entre plusieurs utilisateurs
- Isole les programmes
  - Communications simplifiées mais encadrées
  - Sécurité
  - Stabilité (exemple : crash de Firefox ne doit pas gêner LibreOffice)
- Notion des années 70, toujours d'actualité
  - Faire des traitements parallèles

Notion de processus

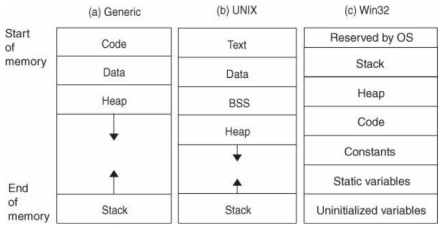
États d'un processus



Notion de processus

Notion de processus

- Dans le noyau du système d'exploitation, un processus est représenté par une structure de données qui comprend :
  - Un identificateur de processus
  - Un espace de mémoire virtuelle qui contient les instructions du programme et des bibliothèques dynamique, une pile (stack) et un tas (heap)
  - Une liste de descripteurs (handle ou descriptor) des différentes ressources systèmes (p. ex. les fichiers ouverts) accessibles par le processus.
  - Un jeton de sécurité qui permet au système d'exploitation de déterminer ce que le processus à le droit de faire.



Notion de processus

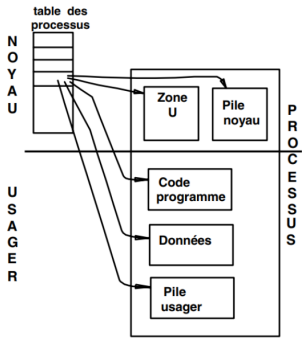
Ordonnancement des processus

- Dans un système multi-utilisateurs à temps partagé, plusieurs processus peuvent être présents en mémoire centrale en attente d'exécution
- Si plusieurs processus sont prêts, le système d'exploitation doit gérer l'allocation du processeur aux différents processus à exécuter. C'est l'ordonnanceur (scheduler) qui s'acquitte de cette tâche.
- Un ordonnanceur fait face à deux problèmes principaux :
  - Le choix du processus à exécuter ;
  - le temps d'allocation du processeur au processus choisi
- Les principaux objectifs d'un ordonnanceur :
  - s'assurer que chaque processus en attente d'exécution reçoive sa part de temps processeur ;
  - minimiser le temps de réponse ;
  - utiliser le processeur à 100
  - utiliser d'une manière équilibrée les ressources ;
  - prendre en compte les priorités ;
  - être prédictible.

Commutation du contexte

Contexte d'un processus

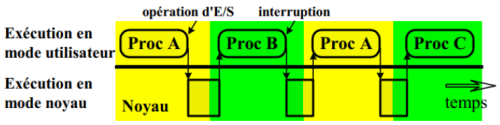
- Le contexte d'un processus est constitué de l'ensemble des données qui permettent de reprendre l'exécution d'un processus qui a été interrompu
  - Son état
  - Son mot d'état, en particulier : la valeur des registres actifs et le compteur ordinal
  - Les valeurs des variables globales statiques ou dynamiques
  - Son entrée dans la table des processus
  - Sa zone u : des informations relatives aux droits d'accès et aux régions mémoires
  - Les piles user et system
  - Les zones de code et de données



Commutation du contexte

Exécution d'un nouveau processus

- Pour exécuter un nouveau processus, il faut :
  1. Sauvegarder le contexte d'unité centrale du processus courant (mot d'état)
  2. Charger le nouveau mot d'état du processus à exécuter
- L'exécution d'un processus s'effectue en deux modes
  1. Mode utilisateur
  2. Mode noyau



Plan

Parallélisme

Programmation parallèle

- Il est parfois utile d'exécuter simultanément plusieurs sousprogrammes pour l'accomplissement d'une tâche.
- Il est possible de réaliser ces sous-programmes comme des programmes séparés et de les exécuter simultanément (p. ex. : scripts CGI).
- Cela présente toutefois certains désavantages :
  - La gestion des processus est relativement lourde pour le système.
  - Le temps nécessaire au démarrage d'un processus peut être long par rapport au temps d'exécution du programme.
  - Les processus sont isolés les uns des autres, l'échange de données ou la synchronisation entre les processus est mal aisé.

Gestion des processus

Création d'un processus

- Les primitives de gestion des processus sont définies dans `<unistd.h>`
- La primitive `fork()` permet de créer un processus identique au créateur par copie
  - Le créateur est appelé père (parent process)
  - Le créé est appelé fils (child process).

```
pid_t fork(void);
```

  - Le PID du processus fils >0 est renvoyé pour le processus parent si OK, sinon -1
  - 0 pour le processus fils

Exemple avec `fork()`

```
#include <stdio.h>
int main(void) {
   int pid;
   pid=fork();
   switch (pid) {
      case -1 : printf("Erreur!"); break;
      case 0 : printf("Je suis le processus fils"); break;
      default : printf("Je suis le processus père"); break;
   }
   return 0;
}
```


Gestion des processus

Terminer un processus

- Un processus se termine automatiquement lorsqu'il cesse d'exécuter la fonction main()
- Il est possible de terminer explicitement l'exécution du processus courant depuis n'importe quelle fonction :

```
void _exit (int status);
void exit (int status);
```

  - `status` est le code de retour compris entre 0 et 255 à envoyer au système d'exploitation. Ce code de retour peut être récupéré depuis le shell à travers la variable `$?`
- La fonction `exit()` effectue les opérations suivantes :
  - Invoquer les routines de terminaisons
  - Fermer les flux d'entrée/sortie
  - Invoquer l'appel `_exit()`

Avertissement

- Un processus qui se termine passe à l'état zombie en attendant que le processus père ait lu son code de retour
- Si le processus père ne lit pas le code de retour de son fils, celui-ci peut rester indéfiniment à l'état zombie

Gestion des processus

Enregistrer des fonctions de terminaison

- Il est possible d'enregistrer des routines de terminaison qui seront invoquées automatiquement lorsque le processus se terminera normalement
  - Il est possible d'enregistrer plusieurs routines
  - Les fonctions enregistrées seront invoquées dans l'ordre inverse de leur enregistrement

```
1 #include <stdlib.h>
2 int atexit(void (*routine)(void));
```

  - Renvoie 0 si OK, sinon une valeur non nulle
  - `routine` est un pointeur de fonction sur une routine de terminaison

Gestion des processus

Attendre la terminaison d'un processus

- Attendre la terminaison d'un processus fils

```
1 pid_t wait(int *status)
```

  - Si le processus courant n'a aucun fils, il retourne -1 avec `errno` valant `ECHILD`
  - Si le processus courant a au moins un fils zombie, un zombie est détruit et son pid est retourné, sinon `wait` bloque en attendant la mort d'un fils
  - `status` permet de stocker la raison de la terminaison du processus fils si sa valeur est non `NULL`

Gestion des processus

Attendre la terminaison d'un processus avec `waitpid()`

- Une version étendue de `wait()`
- Attendre la terminaison d'un processus identifié par son `PID`

```
1 pid_t waitpid(pid_t pid, int *status, int options);
```

  - `status` est le code de retour compris entre 0 et 255 à envoyer au système d'exploitation. Ce code de retour peut être récupéré depuis le shell à travers la variable `$?`

Gestion des processus

Lancement d'un programme

- Les appels de la famille `exec()` permettent de remplacer le programme en cours par un autre programme sans changer de numéro de processus (PID)

```
1 #include <unistd.h>
2 extern char **environ;
3 int execl(const char *path, const char *arg, ...);
4 int execlp(const char *file, const char *arg, ...);
5 int execle(const char *path, const char *arg,..., char * const envp[]);
6 int execv(const char *path, char *const argv[]);
7 int execvp(const char *file, char *const argv[]);
8 int execvpe(const char *file, char *const argv[],
9 char *const envp[]);
```

Plan

Identifier un processus

PID et PPID

- Connaitre son PID

```
1  pid_t getpid(void);
```

- Connaitre le PPID

```
1  pid_t getppid(void);
```

Exemple : identifier un processus

```
1  #include <stdio.h>
2  #include <unistd.h>
3  int main(void){
4     int pid;
5     pid = fork();
6     if (pid > 0)
7        printf("processus père : %d-%d-%d\n", pid, getpid(), getppid());
8     if (pid == 0) printf("prcessus fils : %d %d %d\n",pid,getpid(),getppid());
9     if (pid < 0)
10    printf("Probleme de creation par fork()\n");
11    system("ps -l");
12    return 0;
13 }
```


Identifier un processus

UID, EUID et SUID

- On distingue trois identifiants d'utilisateur par processus
  - UID désigne l'identifiant de l'utilisateur réel qui a lancé le processus
  - EUID (Effective UID) désigne l'identifiant de l'utilisateur effectif qui détermine les privilèges d'accès du processus
  - SUID (Saved UID) est la copie de l'ancien UID réel quand celui-ci a été modifié par un processus
- Rappel sur la permission `Set-UID`
  - Cette permission concerne uniquement les fichiers exécutables binaires
  - Elle permet à l'utilisateur ayant les droits d'exécution sur ce fichier d'exécuter le fichier avec les privilèges de son propriétaire

Identifier un processus

UID et EUID

- Connaitre l'UID et l'EUID
  - UID désigne l'identifiant de l'utilisateur réel qui a lancé le processus
  - EUID (Effective UID) désigne l'identifiant de l'utilisateur effectif qui détermine les privilèges d'accès du processus

```
1  #include <unistd.h>
2  uid_t getuid(void);
3  uid_t geteuid(void);
```

  - Ces fonctions renvoient toujours respectivement l'UID et l'EUID

Identifier un processus

Exemple d'affichage l'UID et l'EUID

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

int main() {
   printf("Real UID = %u, Effective UID = %u\n",getuid(),geteuid());
   return EXIT_SUCCESS;
}
```

- Exécution normale

```
Real UID = 1000 , Effective UID = 1000
```

- Exécution après l'ajout du bit `Set-UID` par l'utilisateur `root`

```
$ sudo chown root Exemple
$ sudo chmod u+s Exemple
Real UID = 1000 , Effective UID = 0
```


Identifier un processus

UID et EUID

- Un processus avec le bit `Set-UID` démarre avec les privilèges du propriétaire du programme
  - Généralement, le propriétaire du programme possède plus de privilèges
  - Il est possible de réduire les privilèges en demandant de remplacer l'UID effectif par l'UID réel

```c
#include <unistd.h>
/* Définir l'UID effectif du processus appelant. Si cet UID effectif est celui
 *  du superutilisateur, les UID réels et sauvés sont également définis */
int setuid(uid_t uid);

/* Définir l'UID effectif du processus appelant. Les processus non privilégiés
 *  peuvent uniquement définir leur UID effectif à la valeur de leur UID réel,
 *  de leur UID effectif ou de leur UID sauvé. */
int seteuid(uid_t euid);

/* Définir les ID d'utilisateur effectif et réel du processus appelant
 *  Les processus non privilégiés peuvent seulement définir leur UID effectif à
la valeur de l'UID réel, de l'UID effectif ou de l'UID sauvé.Les utilisateurs
non privilégiés peuvent seulement définir l'UID réel à la valeur de l'UID réel
ou de l'UID effectif. */

int setreuid(uid_t ruid, uid_t euid);
```

  - Renvoie 0 si OK, sinon -1 et met dans `errno` le code d'erreur

Identifier un processus

UID et EUID

- Lorsqu'une application s'exécute avec un UID effectif différent de son UID réel, c'est dans le but de disposer par moment de privilèges auxquels son utilisateur n'a pas droit directement
- Pour limiter les risques d'attaques, il est conseillé de modifier au plus vite l'UID effectif pour employer l'identité réelle de l'utilisateur

```c
uid_t e_uid_initial;
uid_t r_uid;
int main (int argc, char * argv []) {
   /* Sauvegarde des différents UIDs */
   e_uid_initial = geteuid ();
   r_uid = getuid ();
   seteuid (r_uid); /* limitation des droits à ceux du lanceur */
   ...
   fonction_privilegiee ();
   ...
}

void fonction_privilegiee (void) {
   seteuid (e_uid_initial); /* Restitution des privilèges initiaux */
   ...
   /* Portion nécessitant les privilèges */
   ...
   seteuid (r_uid); /* Retour aux droits du lanceur */
}
```


Plan

Plan

Parallélisme vs concurrence

Notion de thread

- Un thread est un fil d'exécution qui représente l'exécution d'une séquence d'instruction.
- À la création d'un processus, le système d'exploitation crée automatiquement un thread qui démarre au point d'entrée du programme (fonction main).
- L'ordonnanceur (scheduler) du système d'exploitation partage les ressources CPU entre les différents threads.
- Les systèmes d'exploitation modernes permettent de créer des threads supplémentaires dans un même processus pour exécuter plusieurs sous-programmes simultanément.
  - Le programmeur est responsable de la création de threads supplémentaires.

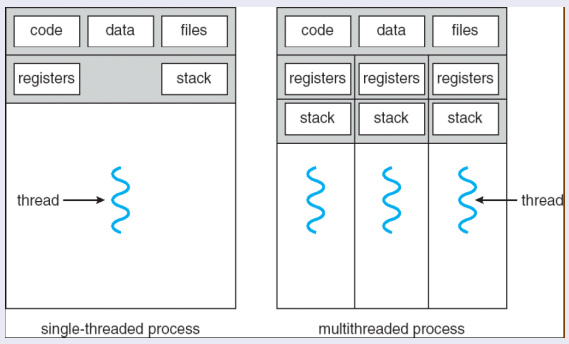
Parallélisme vs concurrence

Processus vs Thread

- Processus (lourd)
  - Contient tout ce qui est nécessaire à l'exécution (registre du processeur, ressources, mémoire, etc.)
  - Communication uniquement via des canaux dédiés (IPC) : Signaux, Tubes, Memory Map, Sockets, etc.
  - Commutation de contexte lente
- Thread (processus léger)
  - Un processus contient un ou plusieurs threads
  - Chaque thread a son propre compteur programme, sa propre pile
  - Ces threads partagent les mêmes ressources, la même mémoire
  - Commutation de contexte plus rapide
  - Communications = partage de variables ⇒ Accès concurrents

Parallélisme vs concurrence

Processus vs Thread



Programmation concurrentielle

Les threads

- Programmation concurrente
  - Travailler sur plusieurs tâches à la fois
- Un thread (tâche) représente (généralement) une fonction (une méthode) en cours d'exécution
- Un processus peut contenir un ou plusieurs threads. Ces threads partagent alors la mémoire ainsi que diverses autres ressources
  - Deux threads qui veulent collaborer peuvent le faire par l'intermédiaire de variables partagées.
  - Le contexte d'un thread est léger par rapport à un processus

Plan

Modèles de séparation d'un programme en threads

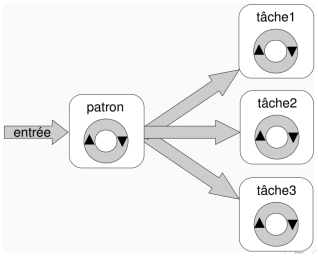
Comment décomposer un programme en threads ?

- Comment décomposer un programme en plusieurs threads ?
- Il existe plusieurs modèles
  - Le modèle délégation (boss-worker model ou delegation model)
  - Le modèle pair (peer model)
  - Le modèle pipeline (pipeline model)

Modèles de séparation d'un programme en threads

Le modèle délégation

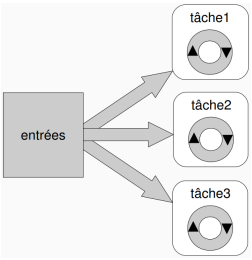
- Un seul thread, le patron, accepte les entrées pour l'ensemble du programme
  - Sur la base de cette entrée, le patron transmet des tâches spécifiques à un ou plus aux threads de travail
- Le patron crée chaque thread de travail, lui attribue des tâches et, si nécessaire, attend qu'il se termine
- Il est important que vous minimisez la fréquence à laquelle le patron et ouvriers communiquent.



Modèles de séparation d'un programme en threads

Le modèle pair

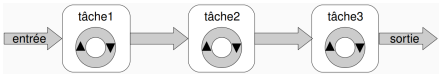
- Tous les threads travaillent simultanément sur leurs tâches sans leader.
- Un seul thread doit créer tous les autres threads homologues au démarrage du programme.
  - Ce thread agit ensuite comme un autre thread, ou se suspend en attendant les autres pairs finir.
- Le modèle de pair convient aux applications qui ont un ensemble fixe et bien défini d'entrées



Modèles de séparation d'un programme en threads

Le modèle pipeline

- Le modèle de pipeline suppose
  - Un long flux d'entrée
  - Une série de sous-opérations (appelées étapes ou filtres) à travers lesquelles chaque unité d'entrée doit être traitée.
  - Chaque étape de traitement peut gérer une unité d'entrée différente à la fois.



Programmation concurrentielle

POSIX ?

- POSIX : Portable Operating System Interface for UNIX
- Avant le POSIX
  - Chaque OS possède sa propre API pour gérer les threads
  - Difficile d'écrire des programmes multi-threadé :
- Après
  - POSIX (IEEE 1003.1c-1995) offre un standard pour la gestion des threads

Programmation concurrentielle

Généralités sur l'interface Pthreads

- Trois grandes catégories de fonctions / types de données
1. Gestion des threads
   - Créer des threads, les arrêter, contrôler leurs attributs, etc.
2. Synchronisation par mutex (mutual exclusion)
   - Créer, détruire verrouiller, déverrouiller des mutex ; Contrôler leurs attributs
3. Variables de condition
   - Communications entre threads partageant un mutex
   - Les créer, les détruire, attendre dessus, les signaler ; Contrôler leurs attributs

Programmation concurrentielle

Généralités sur l'interface Pthreads

- 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`

Plan

Gestion des threads

Créer un thread

- Créer un nouveau thread

```
1 int pthread_create(pthread_t *restrict thread,const pthread_attrt *restrict attr,
2       void *(*start routine)(void*), void *restrict arg);
```

  - ☞ Si `attr` est `NULL`, les attributs par défaut seront utilisés
  - ☞ En cas de réussite, `pthread_create()` stocke l'ID du thread créé dans l'emplacement référencé par `thread`

Gestion des threads

Terminer un thread

- Terminer le thread appelant

```
1 void pthread_exit(void *value ptr)
```

  - ☞ Si `attr` est `NULL`, les attributs par défaut seront utilisés
  - ☞ En cas de réussite, `pthread_create()` stocke l'ID du thread créé dans l'emplacement référencé par `thread`

Gestion des threads

Attendre la terminaison d'un thread

- Attendre la fin d'un thread

```
int pthread_join(pthread_t thread, void **value ptr)
```

  - 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()`

Gestion des threads

Détacher un thread

- Terminer le thread appelant

```
int pthread_detach(pthread_t thread)
```

  - Détacher un thread revient à dire que personne ne le joindra à sa fin
  - Le système libère ses ressources au plus vite
  - Evite les threads zombies quand on ne veut ni synchroniation, ni code de retour ´
  - On ne peut pas ré-attacher un thread détaché

Plan

Attributs d'un threads

Attributs d'un threads

- Les attributs d'un threads sont représentés par la structure `pthread_attr_t`
  - Il s'agit d'une structure opaque : il ne faut faire ni affectation, ni affectation, ni comparaison directe
- La structure doit être initialisée avant son utilisation

```
int pthread_attr_init(pthread_attr_t * attributs)
```

  - Il s'agit d'une structure opaque : il ne faut faire ni affectation, ni affectation, ni comparaison directe
- La libération de l'objet des attributs

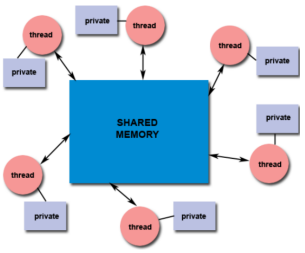
```
int pthread_attr_destroy(pthread_attr_t * attributs)
```

Plan

Partage de données

Modèle de mémoire partagée

- Tous les threads ont accès à la même mémoire partagée globalement
- Les données peuvent être partagées ou privées
- Les données partagées sont accessibles par tous les threads
- Les données privées ne peuvent être accédées que par les threads qui les possèdent
- Synchronisation explicite



Partage de données entre les threads

Partage des données

- Toute variable créée avant la création du thread et accessible depuis un ou plusieurs threads est une variable partagée
  - Les variables globales sont des variables partagées
  - Les variables locales à la fonction associée à un thread ne sont pas partagées
- Le spécificateur de stockage `thread_local` permet à plusieurs threads d'utiliser le stockage local via un point d'accès global.
  - Avec `thread_local` une variable globale devient une variable locale au thread

Partage de données entre les threads

Section critique

- Partie d'un thread dont l'exécution ne doit pas entrelacer avec d'autres threads
  - Indivisibilité de la section critique
- Une fois un thread entre dans la section critique
  - Le thread qui entre dans la SC doit terminer son travail en empêchant les autres threads de jouer sur les mêmes données
  - La section critique doit être verrouillée afin de devenir indivisible

Partage de données entre les threads

Problème de section critique

- Lorsqu'un thread manipule une donnée (ressource) partagée avec d'autres, nous disons qu'il se trouve dans une section critique
- Le problème de la SC est de trouver un algorithme d'exclusion mutuelle de threads
  - Les résultats de leurs actions ne dépend pas de l'ordre de leur entrelacement
- L'exécution de la section critique doit être mutuellement exclusive et indivisible
  - Un seul thread exécute la SC

Plan

Les mutex

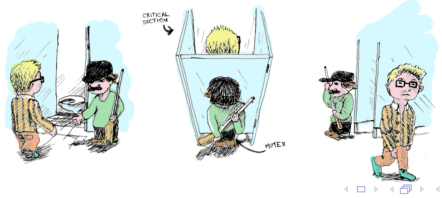
Coordination par les `mutex`

- Un mutex (mutual exclusion) est une structure de données qui permet de contrôler l'accès à une ressource
- Un mutex qui contrôle une ressource peut se trouver dans deux états
  1. Libre (unlocked) : indique que la ressource est libre et peut être utilisée sans risquer de provoquer une violation d'exclusion mutuelle
  2. Réservée (locked) : indique que la ressource associée est actuellement utilisée et qu'elle ne peut pas être utilisée par un autre thread

Les mutex

Utilisation d'un mutex

- L'utilisation typique d'un mutex pour assurer l'accès en exclusion mutuelle à une SC
  - Créer et initialiser une variable du mutex
  - Plusieurs threads tentent de vérouiller le mutex avant d'entrer dans la section critique
  - Seulement un thread réussit et prend le mutex, les autres sont bloqués ⇒ le mutex est vérouillé
  - Le thread propriétaire du mutex effectue un ensemble d'actions dans la section critique
  - Le thread propriétaire dévrouille le mutex en quittant la section critique
  - Un seul parmi les threads bloqués est réveillé, et il prend le contrôle mutex en répétant le processus



Les mutex

Création et destruction d'un mutex

- Type de donnée d'un mutex : `pthread_mutex_t`
- Création
  - Statique

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER ;
```

  - Dynamique

```
pthread_mutex init(pthread_mutex_t *, pthread_mutexattr_t *) ;
```

- Destruction

```
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

  - Le mutex doit être déverrouillé avant sa destruction
  - Retourne 0 en cas de succès, sinon un code d'erreur

Les mutex

Verouillage/Déverouillage d'un mutex

- Bloquer le thread courant si le mutex est verouillé par un autre thread, sinon verouiller le mutex

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
```

  - Renvoie 0 si OK, sinon un code d'erreur
  - `mutex` est le mutex à verrouiller
- Tenter de verrouiller un mutex sans bloquer le thread courant

```
int pthread_mutex_trylock(pthread_mutex_t *mutex);
```

  - Renvoie 0 si le mutex a été bien verrouillé, sinon l'erreur `EBUSY`
- Libérer un mutex

```
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```


Les mutex

Exemple d'utilisation des mutex

```
#include <pthread.h>

pthread_mutex_t count_mutex;
long long count;

void increment_count() {
   pthread_mutex_lock(&count_mutex);
   count = count + 1;
   pthread_mutex_unlock(&count_mutex);
}

long long get_count()
{
   long long c;
   pthread_mutex_lock(&count_mutex);
   c = count;
   pthread_mutex_unlock(&count_mutex);
   return (c);
}
```


Plan

Sémaphore

Présentation

- Sémaphores généralisent les verrous
- Inventés par Dijkstra à la fin des années 1960
- Principale primitive de synchronisation à l'origine dans UNIX
- Définition : un sémaphore est un entier non négatif qui a deux opérations
  - `P()` : opération atomique qui attend que le sémaphore soit positif et le décrémente de 1. Proberen (tester en néerlandais) = puis-je ?
  - `V()` : opération atomique qui incrémente le sémaphore de 1, et réveille les threads/processus bloquant sur `P()`. Verhogen (incrémente en néerlandais) = vas-y

Sémaphore

Création d'un sémaphore nommé

- Un sémaphore peut être partagé entre plusieurs processus
- Créer un sémaphore nommé

```
1          sem_t *sem_open(char *name, int flags);
2          sem_t *sem_open(char *name, int flags, mode_t mode,unsigned int value);
```

  - Renvoie 0 si OK, sinon un code d'erreur
  - `name` : le nom du sémaphore
  - `value` : valeur initiale du sémaphore. Par défaut 1.
  - `mode` : spécifie les droits d'accès et l'accès écriture doit être autorisé
  - `flags` : Seulement quelques valeurs possibles de flags parmi celles du `open` : `O_CREAT` : création du sémaphore s'il n'existe pas
    `O_EXCL` : création du sémaphore s'il n'existe pas sinon échec
- Libérer un sémaphore

```
1  int sem_close(sem_t *s);
```


Sémaphore

Création d'un sémaphore anonyme

- Un sémaphore anonyme ne peut être utilisé que par les threads d'un même processus
- Créer un sémaphore anonyme

```
1  int sem_init(sem_t *sem, int pshared, unsigned int value);
```

  - Renvoie 0 si OK, sinon un code d'erreur
  - `name` : le nom du sémaphore
  - `pshared` : 0 partagé entre les threads, 1 partagé entre les processus
  - `flags` : Seulement quelques valeurs possibles de flags parmi celles du open : `O_CREAT` et `O_EXCL` pour flags
- Libérer un sémaphore

```
1  int sem_destroy(sem_t *sem);
```


Sémaphore

Contrôler un sémaphore

- Incrémenter un sémaphore : l'opération `V()`

```
1  int sem_post(sem_t *s);
```

  - Renvoie 0 si OK, sinon un code d'erreur
- Décrémenter un sémaphore : l'opération `P()` et bloque le thread si nécessaire

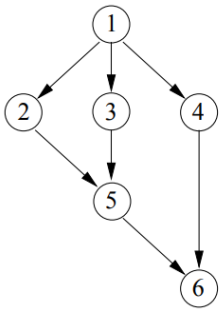
```
1  int sem_wait(sem_t *s);
```

  - Bloque si la valeur est <= 0. Le processus est débloqué si la valeur > 0. Retourne -1 si erreur.

Sémaphore

Exercice d'application

- Donner le pseudo-code pour synchroniser les processus de l'arborescence suivante s´exécutant en parallèle



Plan

Variable de condition

Présentation

- Attendre (sans CPU) qu'un autre thread ait réalisé un traitement
- Notifier les autres threads quand un travail est terminé
  - Coopération entre les threads !
- Nécessite la mise en place de variables partagées
  - Pour indiquer la fin d'un traitement
  - Pas d'attente active
  - Donc, basée sur l'utilisation de mutex

Variable de condition

Présentation

- En C, les variables de conditions sont représentées par le type `pthread_cond_t`
- Principe de fonctionnement
  - Un thread reste bloqué jusqu'à que la condition spécifiée par la variable de condition soit réalisée
  - Un autre thread sert à signaler que la condition a été bien remplie
- Avant son utilisation, une variable de condition doit être initialisée
  - Initialisation statique

```
pthread_cond_t cond=PTHREAD_COND_INITIALIZER;
```

  - Initialisation dynamique

```
int pthread_cond_init(pthread_cond_t *cond,const pthread_condattr_t *cattr);
```

    L'argument `cattr` n'a pas d'utilité et il est fixé à `NULL`
- Une variable de condition inutilisée est libérée avec `pthread_cond_destroy()`

```
int pthread_cond_destroy(pthread_cond_t *cond);
```

Variable de condition

Principe d'utilisation d'une variable de condition

- Une variable de condition est toujours associée à un mutex
  - Éviter les problèmes d'accès concurrents à la variable
- Le thread qui doit attendre une condition
  - Initialise la variable et le mutex qui lui est associé
  - Bloque le mutex , puis se mettre en attente de la réalisation de la condition
  - Libère le mutex
- Le thread réalisant la condition
  - Travaille jusqu'à réaliser la condition
  - Bloque le mutex associé à la condition
  - Envoie un signal pour montrer que la condition est remplie
  - Libère le mutex

Variable de condition

Mise en attente sur une variable de condition

- Mise en attende d'un thread sur une variable de condition

```
1 int pthread_cond_wait(pthread_cond_t *cond,pthread_mutex_t *mutex);
```

  - Renvoie 0 si OK, sinon le code de l'erreur
  - Bloque le mutex associé à la condition
  - `cond` est la variable de condition
  - `mutex` est le mutex associé à la variable de condition

Exemple

```
1 void foo () {
2    pthread_mutex_lock(&verrou);
3    while (done == 0)
4       pthread_cond_wait (&varcond,&verrou);
5    pthread_mutex_unlock(&verrou);
6 }
```


Variable de condition

Signaler la réalisation d'une condition

- Signaler la réalisation d'une condition à un seul thread bloqué : un seul thread sera réveillé

```
1 int pthread_cond_signal(pthread_cond_t *cond);
```

- Signaler la réalisation d'une condition à tous les threads bloqués : tous les threads seront réveillés

```
1 int pthread_cond_broadcast(pthread_cond_t *cond);
```

  - Renvoie 0 si OK, sinon le code de l'erreur
  - `cond` est la variable de condition

Exemple

```
1 void bar() {
2    pthread_mutex_lock(&verrou);
3    done = 1;
4    pthread_cond_signal (&varcond);
5    pthread_mutex_unlock(&verrou);
6 }
```


MERCI POUR VOTRE ATTENTION

Questions ?