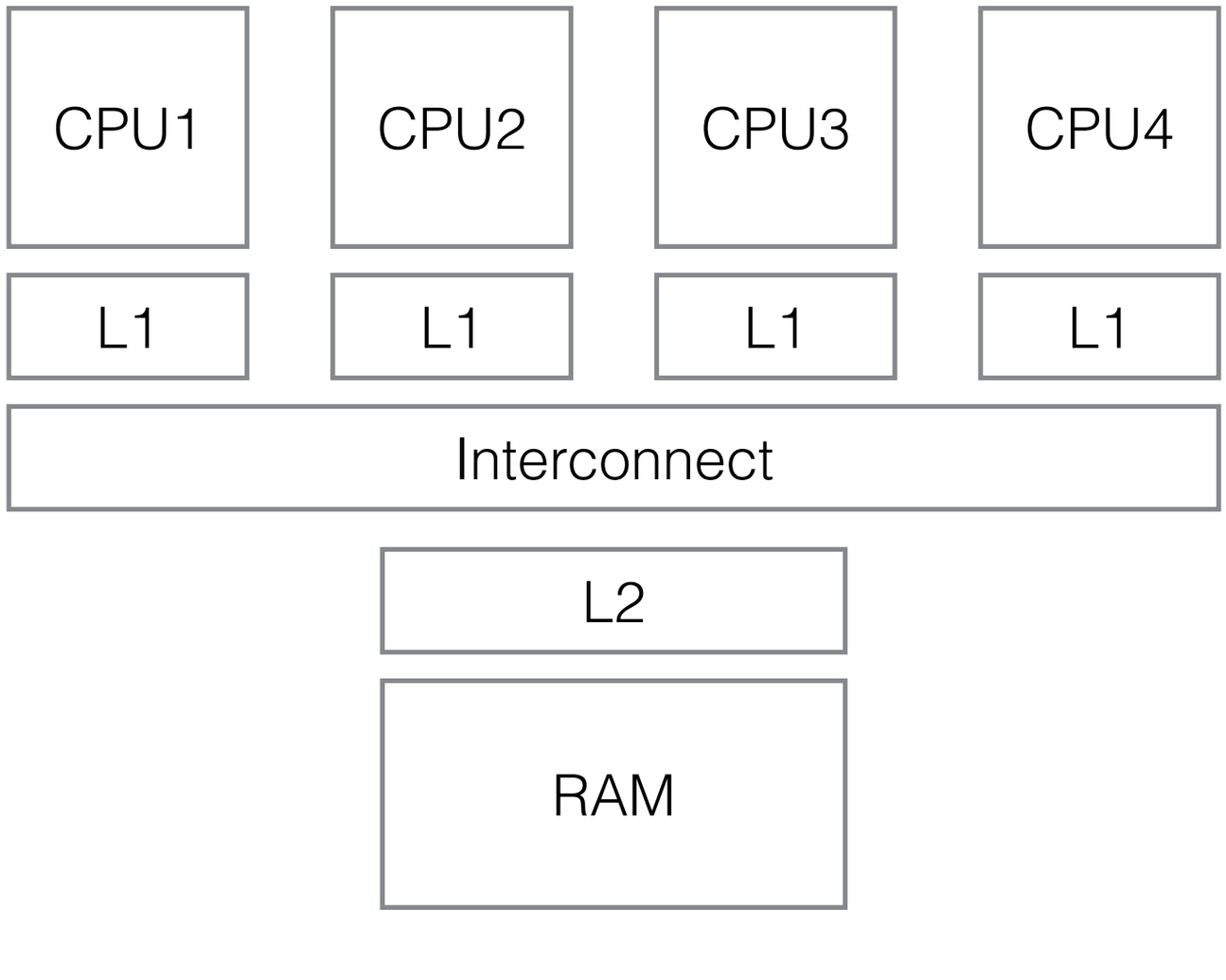
# INF4033 - Partie 5 : Introduction aux Pthread

Alexandre BRIÈRE
alexandre.briere@esiea.fr

# Plan

- Introduction aux threads
- Le standard Pthread
- Gestion de thread
- Exclusion mutuelle
- Variable de condition

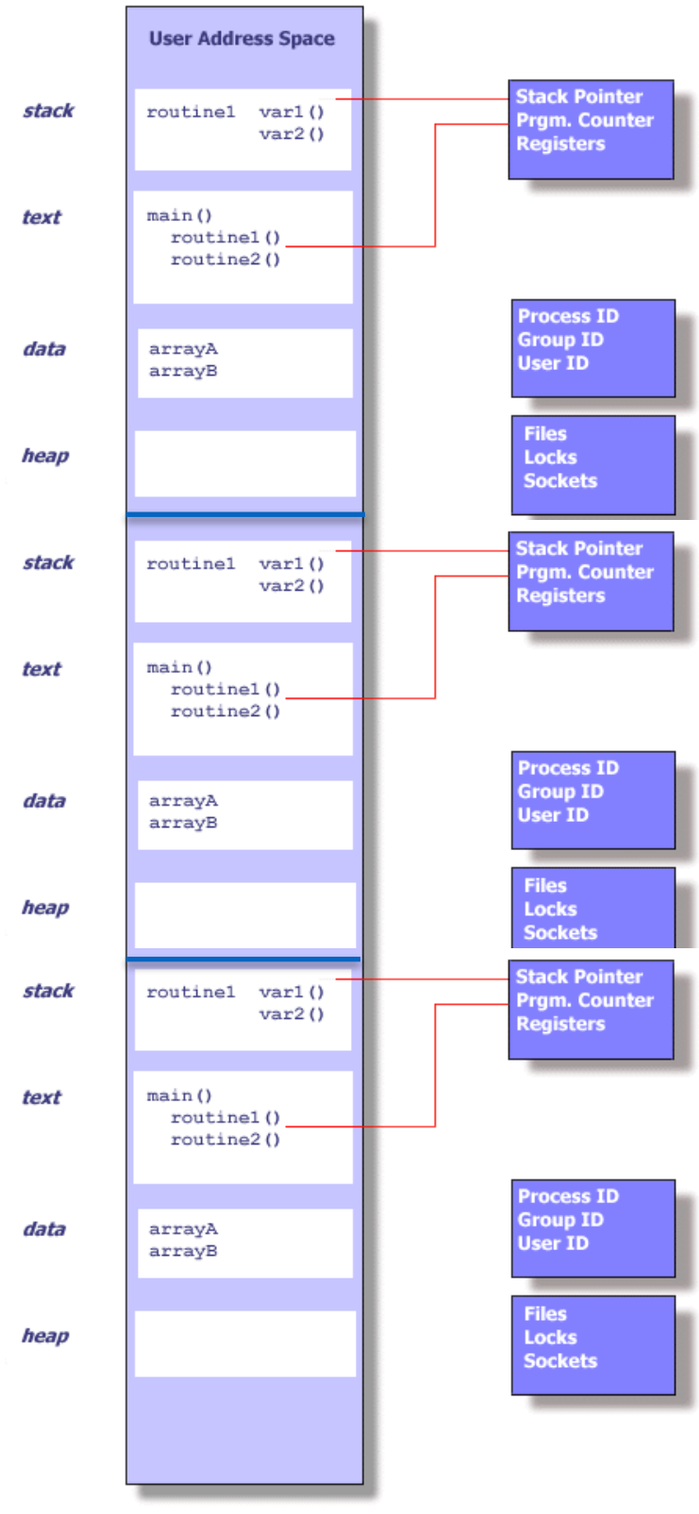
# Architectures actuelles




- SMP : Symetric Multi Processor
- RAM : partagée

# Processus

- Service du système (OS)
  - Mémoire isolée
  - Communication encadrée
- Sauvegarde d'un processus :
  - Contexte: PID, GID, UID
  - Environnement, répertoire de travail
  - Code
  - Registres
  - Pile
  - Tas
  - Descripteurs de fichiers
  - Signaux
  - Librairies
  - IPC



# Thread

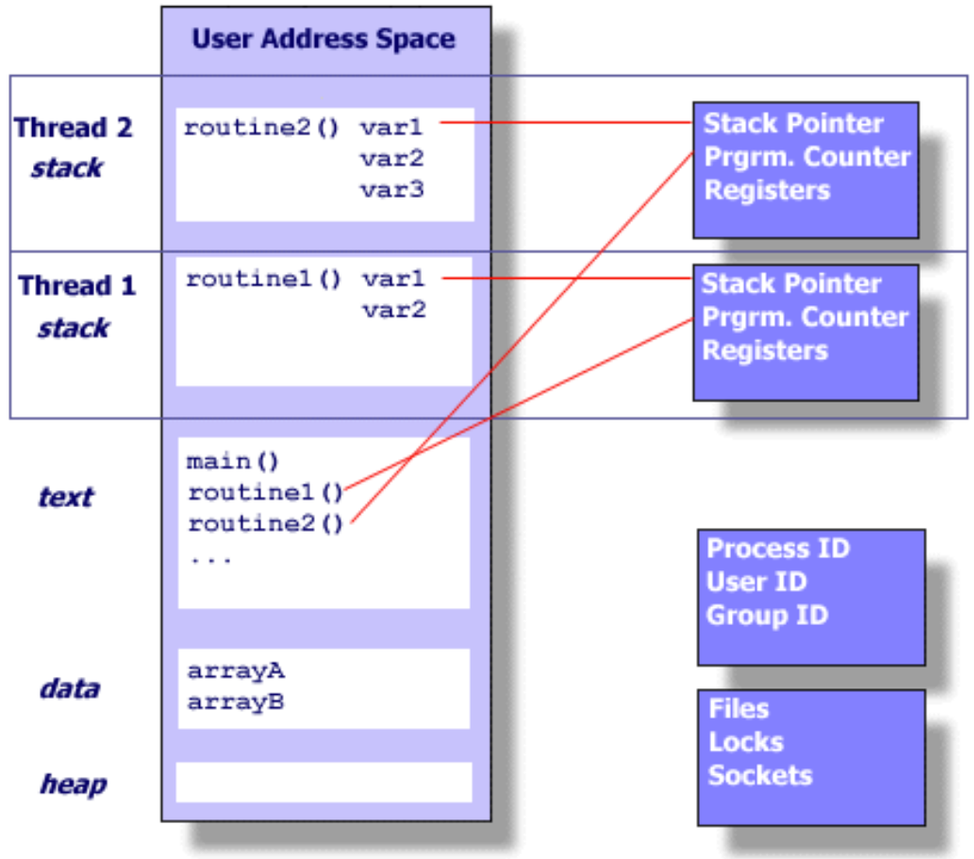
Aussi appelé « processus léger »

Service du système ou librairie :

- Mémoire partagée

Sauvegarde d'un thread :

- Code
- Registres
- Pile
- Données spécifiques au thread




Coût de changement de contexte est plus faible que pour un processus :

- Les modifications d'un thread dans une zones globale sont vues par tous les autres.
- Les lectures et écritures dans une case mémoire partagée doit être protégées.

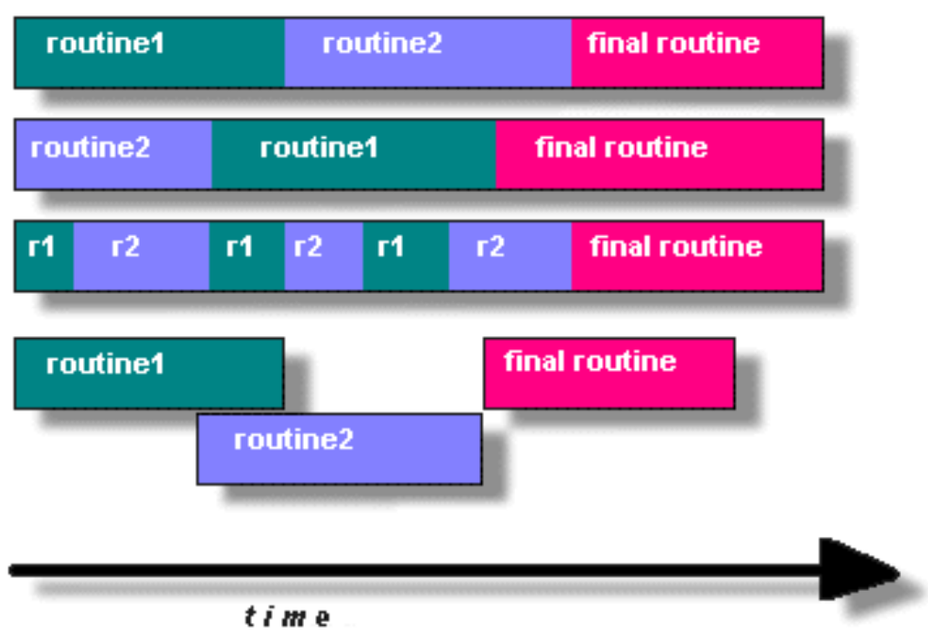
# Pthread

- Pthread => Bibliothèque POSIX
- Standardisé en 1995
- Définie un ensemble de structure, procédure dans le language C
- Disponible sur Linux, Windows, OS X…
- Les performances et résultats de la librairie dépendent de son implémentation. Par exemple, le nombre de thread maximal et la taille de la pile allouée peut varier grandement.

http://standards.ieee.org/findstds/standard/1003.1-2008.html
http://www.opengroup.org/austin/papers/posix_faq.html
http://www.unix.org/version3/ieee_std.html

# Pthread

- Pour qu'un programme soit un bon candidat pour l'utilisation de Pthread, il doit pouvoir s'organiser en tâches pouvant s'exécuter de façon concurrente.
- Concurrence : exécution de plusieurs flux d'instructions simultanément.




Travail pouvant être fait en parallèle.

Travail répondant à des événements asynchrones.

Travail aillant des priorités différentes.

Travail bloqué par des attentes d'entrées/sorties longues.

# Applications

- Worker/Manager : Un thread manager assigne du travail à d'autres threads (worker). Le manager gère les entrées du programme et redistribue le traitement.
  => Serveur web (Apache).

- Pipeline: Une tâche complexe est décomposée en tâches plus courtes. Ces tâches sont gérée en série mais de façon concurrente comme dans une chaine d'assemblage de voiture.
  => Compression vidéo.

- Peer: Similaire à Worker/Manager sauf que la manager participe au travail une fois celui-ci distribué.
  => Compression Image
  => Algorithme de tri
  => Algorithme de recherche

# API Pthread

4 groupes :

- Gestion des tâches : Création/Destruction de thread
  - `pthread_`
  - `pthread_attr_`
- Mutex : Mécanisme d'exclusion mutuelle
  - `pthread_mutex_`
  - `pthread_mutexattr_`
- Variable de condition : Communication sous condition
  - `pthread_cond_`
  - `pthread_condattr_`
- Synchronisation (barrière et lock) : Verrou, clé de stockage, barrière
  - `pthread_key_`
  - `pthread_rwlock_`
  - `pthread_barrier_`

# Compilation

Pour compiler et faire l'édition de lien :

```
$ gcc -pthread
$ icc -pthread
$ clang -lpthread
```

Pour exécuter :

```
$ ./executable
```

# API create 1/2

Créer un thread :

```
int pthread_create(pthread_t *tid,
                   pthread_attr_t *attr,
                   void * (*fonction) (void *),
                   void *arg);
```

`tid` : permet de stocker l'id du thread créé

`attr` : permet de définir les attributs du thread

`fonction` : pointeur vers la fonction qui sera le point d'entrée (et de sortie) du thread créé

`arg` : est le paramètre transmit à `fonction` lors de son appel

# API create 2/2

Erreurs renvoyées par `pthread_create` :

- `EAGAIN` : manque de ressource.
- `EPERM` : pas la permission pour le type d'ordonnancement demandé.
- `EINVAL` : attributs spécifiés par `attr` ne sont pas valables.

Le programme principal est le thread par défaut.

**—> exemple_1.c**

# Terminaison

Arrêter un thread :

```
void pthread_exit(void *state);
```

Faire attention à la valeur de `state`, elle doit pouvoir être accessible depuis d'autre thread (Pas de variable de pile !).

Le comportement de cette fonction dépend de l'attribut `detachstate`.

**—> exemple_2.c**

# Joignable/Détaché

Deux types de thread :

- Joignable (par défaut)
  - Attribut : `PTHREAD_CREATE_JOINABLE`
  - En se terminant suite à un appel à `pthread_exit`, les valeurs de son identité et de retour sont conservées jusqu'à ce qu'une autre thread en prenne connaissance (appel à `pthread_join`). Les ressources sont alors libérées.

- Détachée
  - Attribut : `PTHREAD_CREATE_DETACHED`
  - Lorsque la thread se termine toutes les ressources sont libérées.
  - Aucune autre thread ne peut les récupérer.

# API Detach

```
int pthread_detach(pthread_t tid);
```

Ou

```
pthread_attr_t attr;
pthread_attr_init(&attr);
pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
pthread_create(tid, &attr, func, NULL);
```

# API Join

Attendre la fin d'un thread:

```
int pthread_join (pthread_t tid, void **thread_return);
```

- `EINVAL` : le thread n'est pas joinable
- `ESRCH` : le thread a déjà été libérer

- Si le thread n'est pas terminé la fonction bloque
- Si le thread est terminé la fonction ne bloque pas

Les ressources du thread sont libérées après cet appel.

**—> exemple_3.c**

# API Attr

Attributs :

Initialise tous les attributs avec la valeur par défaut :
```
pthread_attr_init(&attr);
```

Lire/modifier la taille de la pile :
```
pthread_attr_getstacksize(&attr, &stacksize);
pthread_attr_setstacksize(&attr, stacksize);
```

Lire/modifier l'ordonnancement utilisé :
```
pthread_attr_setschedpolicy(&attr, SCHED_FIFO);
pthread_attr_getschedpolicy(&attr, &policy);
```

# Exemple

- Un patron
  - Dont le rôle est de fournir du travail

- Des travailleurs
  - Dont le rôle est de réaliser le travail

Chaque protagoniste sera un thread.
Le travail sera représenté par un entier accessible à tous. Le patron rajoutera une constante à cette variable partagée et les travailleurs soustrairont une constante.

**—> exemple_4.c**

# Cohérence mémoire

Problème de cohérence mémoire surviennent lorsque un processeur lit, modifie, puis écrit une valeur en mémoire.

Exemple :

| C | ASM |
|---|---|
| a-- | LD<br>DEC<br>ST |

Les opérations ne sont pas atomiques. Il faut rendre cette opération atomique

# Cohérence mémoire

CPU1

```
LD
DEC
ST
```

CPU2

```
LD
INC
ST
```

Valeur de a ?

```
a = 0x1
a = 0x0
a = 0xFFFFFFFF
```

Impossible de prédire

Certaines architectures les écritures se font en plusieurs cycles (poids fort puis poids faible):

a = 0xFFFF0001

# Mutex

Le mutex permet de créer un bloc d'instruction atomique. Un mutex est semblable à un sémaphore binaire (Sans garantie d'ordre fifo).
Il permet de se prévenir des "races conditions"

Attention à garder les blocs d'instructions les plus court possible :
Exemple Big Kernel Lock présent dans linux jusqu'à la version 2.6.37
Le noyau n'était pas parallèle. Un seul thread pouvait entrer dans le noyau à la fois.

# API Mutex

Initialisation:

```
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
int pthread_mutex_init(pthread_mutex_t *m,
                       pthread_mutex_attr *attr);
```

Destruction:

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

Verrouillage:

```
int pthread_mutex_lock(pthread_mutex_t *m); // Bloquant
int pthread_mutex_trylock(pthread_mutex_t *m);
```

Dévérouillage:

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

**—> exemple_5.c et exemple_6.c**

# API Conditions

Attente d'événement sur des données partagées par un mutex

Initialisation:

```
pthread_cond_t m = PTHREAD_COND_INITIALIZER;
int pthread_cond_init(pthread_cond_t *m,
                      pthread_cond_attr *attr);
```

Attente:

```
int pthread_cond_wait(pthread_cond_t *cond,
                      pthread_mutex_t *m);
```

Signalement:

```
int pthread_cond_signal(pthread_con_t *cond);
int pthread_cond_broadcast(pthread_con_t *cond);
```

**—> exemple_7.c et exemple_8.c**

# API Conditions

Destruction:

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

**Attention :**

Vérifier toujours que la condition pour laquelle le thread s'était endormi est valide !

```
while (condition) {
   pthread_cond_wait(&cond,&mutex);
}
```