INFO0604
PROGRAMMATION MULTI-THREADÉE

COURS 2

PROCESSUS ET THREADS, PROGRAMMATION ASYNCHRONE, PTHREADS, CYCLE DE VIE D'UN THREAD

Pierre Delisle
Département de Mathématiques, Mécanique et Informatique
Décembre 2020

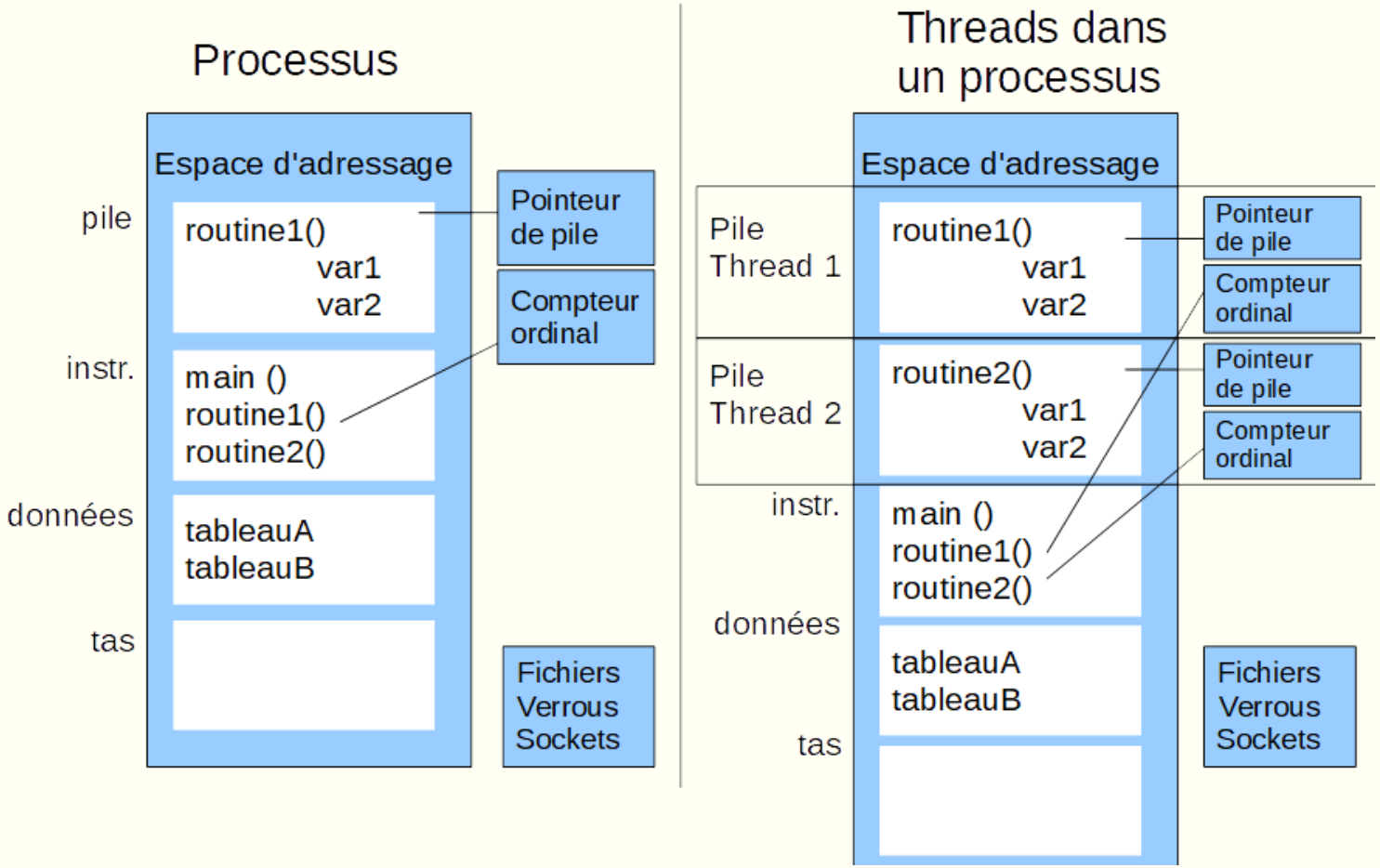
Plan de la séance

- Processus et threads
  - Concurrence et parallélisme
- La programmation asynchrone par l'exemple
  - Séquentiel, multi-processus, multi-thread
- Pthreads
  - Création d'un thread, exécution, terminaison
- Cycle de vie d'un thread

PROCESSUS ET THREADS

Thread

- Ensemble de propriétés permettant « la continuité et la séquence » dans un ordinateur
- Comprend l'état de la machine nécessaire pour exécuter une séquence d'instructions
  - Location de l'instruction courante
  - Registres d'adresses et de données
  - …
- Processus : thread + espace d'adressage + descripteurs de fichiers + …



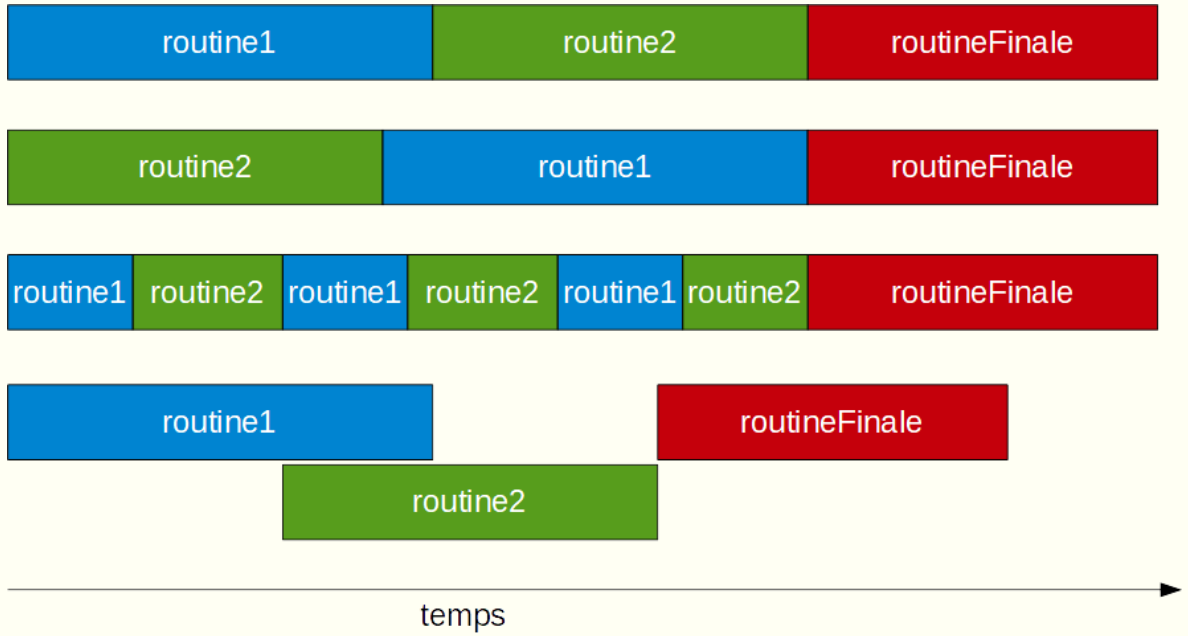
Asynchronisme, concurrence et parallélisme

- Asynchronisme
  - Deux opérations sont asynchrones lorsqu'elles peuvent être exécutées indépendamment l'une de l'autre
  - Les programmeurs à la dérive (« bailing programmers »)
    - Programmeur : thread
    - Seau : élément de synchronisation
    - Cris, coup d'épaule : mécanisme de communication
- Concurrence
  - Les instructions peuvent être exécutées en même temps, mais peuvent être exécutées en séquentiel
    - illusion du parallélisme
  - Comportement des processus/threads sur un ordinateur séquentiel
- Parallélisme
  - Les instructions sont effectivement exécutées en même temps
  - Comportement sur un multiprocesseur

Gestion de la concurrence

- Contexte d'exécution
  - État d'une entité concurrente
  - Le système doit permettre de
    - Créer/supprimer des contextes d'exécution
    - Maintenir l'état des contextes d'exécution indépendamment les uns des autres
- Ordonnancement (scheduling)
  - Détermine quel contexte doit être exécuté à un moment précis
  - Passe d'un contexte à l'autre si nécessaire
- Synchronisation
  - Mécanismes permettant aux contextes concurrents de coordonner leur utilisation des ressources partagées
  - Empêchent les contextes d'exécution de s'exécuter en même temps

- Exécution concurrente



Vue d'ensemble

	Contexte d'exécution	Ordonnancement	Synchronisation
Trafic routier	Automobile	Panneaux et feux	Clignotants et feux de freinage
UNIX (avant les threads)	Processus	Priorité (nice)	wait et pipes
Pthreads	thread	Priorités, règles	Mutex, sémaphores

Donc, un thread c'est ...

- La partie d'un processus nécessaire pour exécuter du code
  - Pointeur sur l'instruction courante du thread (Program Counter)
  - Pointeur sur la pile du thread
  - Pointeur sur les registres (instructions/données)
- N'inclut pas
  - Descripteurs de fichiers, espace d'adressage
  - Ils sont partagés entre les threads du processus
- Les threads sont plus simples que des processus
  - Au niveau système, il est beaucoup plus rapide de passer d'un thread à l'autre que d'un processus à l'autre
  - Lorsqu'un processeur passe d'un processus à l'autre (changement de contexte), l'état matériel de la machine devient invalide
    - Pas dans le cas des threads
- La clé du succès en programmation multi-thread
  - Il faut suffisamment de threads, mais pas trop
  - Il faut suffisamment de communications, mais pas trop
  - Il faut apprendre à juger d'un bon équilibre selon la situation

PROGRAMMATION ASYNCHRONE PAR L'EXEMPLE

Exemple : programme d'alarme

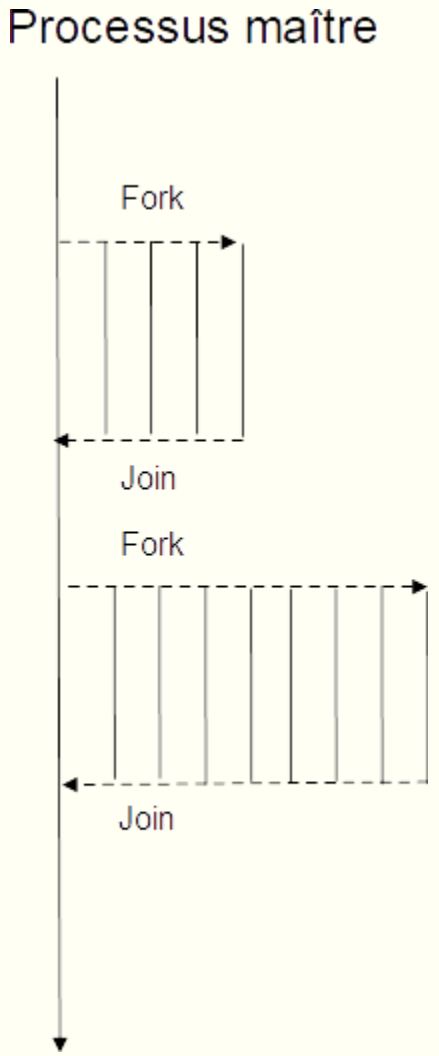
- On demande à l'utilisateur d'entrer, au clavier
  - Une durée à écouler avant l'affichage d'une alarme
  - Le message d'alarme à afficher
- Version 1 : alarme.c
  - Séquentiel
  - Synchrone

```
int main(int argc, char *argv) {
        int secondes;
        char message[30];

        while (1) {
                printf("Alarme>  ");
                scanf("%d %s", &secondes, message);
                sleep(secondes);
                printf ("(%d) %s\n", secondes, message);
        }
}
```

Modèle fork/join

- Processus maître
  - Exécution du code séquentiel
  - Débute et termine l'exécution
  - Fork : création de processus esclaves
- Processus esclaves
  - S'exécutent de façon concurrente
  - Join : destruction des esclaves et retour du contrôle au maître



Modèle fork/join

- Version 2 : alarme_fork.c
  - Asynchrone
  - Plusieurs processus

```
int main(int argc, char *argv) {
		int secondes;
		char message[30];
		pid_t pid;

		while (1) {
			printf("Alarme> ");
			scanf("%d %s", &secondes, message);

			pid = fork(); //creation d'un processus, retourne un id de processus
			if (pid == (pid_t) -1)
				printf("Probleme de fork");
			else {
				if (pid == (pid_t) 0 ) {
					/*Dans l'enfant, on attend et ensuite on affiche un message*/
					sleep(secondes);
					printf ("\n(%d) %s\nAlarme> ", secondes, message);
					exit(0);
				}
				else {
					/*Dans le parent, on appelle waitpid pour recuperer les enfants
					qui ont deja termine */
					do {
						pid = waitpid ((pid_t) -1, NULL, WNOHANG);
						if (pid == (pid_t) -1)
							printf("Probleme attente enfant");
					} while (pid != (pid_t) 0);
				}
			}
		}
}
```

alarme_fork.c

fork()

- Créée un nouveau processus
- Retourne
  - Une valeur négative si la création a échoué
  - 0 au processus enfant créé
  - Une valeur positive, de type pid_t (sys/types.h), au processus parent : le pid de l'enfant
- Une copie de l'espace d'adressage du parent est créée et fournie à l'enfant

waitpid()

- Attend la fin d'un processus
- pid_t waitpid(pid_t pid, int *status, int options);
  - pid = -1 : attendre la fin de n'importe quel fils
  - status = NULL : on ne récupère pas le statut de terminaison
  - options = WNOHANG : si aucun fils n'est terminé, retourne 0 et l'exécution du parent continue
- Lors du retour d'un enfant, libère les ressources

Modèle de threads

- Version 3 : alarme_thread.c
  - Asynchrone
  - Un seul processus
  - Plusieurs threads

```
typedef struct alarme_tag {
        int secondes;
        char message[30];
} alarme_t;

void *alarme_thread(void *arg) {
        alarme_t *alarme = (alarme_t *) arg;
        int statut;

        statut = pthread_detach (pthread_self());
        if (statut != 0)
                printf("Probleme detach\n");
        sleep (alarme->secondes);
        printf("\n(%d) %s\nAlarme>  ", alarme->secondes, alarme->message);
        fflush(stdout);
        free(alarme);
        return NULL;
}

int main(int argc, char *argv) {
        int statut;
        alarme_t *alarme;
        pthread_t thread;

        while (1) {
                printf("Alarme>  ");
                alarme = (alarme_t *) malloc (sizeof(alarme_t));
                scanf("%d %s", &alarme->secondes, alarme->message);

                statut = pthread_create(&thread, NULL, alarme_thread, alarme);
                if (statut != 0)
                        printf("Probleme creation thread alarme\n");
        }
}
```

Modèle de threads

pthread_create()

- Créé un nouveau thread
- int pthread_create(pthread_t * thrd,pthread_attr_t *attr, void * (*start_routine)(void *), void *arg);
  - S'exécute de façon concurrente avec l'appelant
  - thrd : stocke l'identifiant du thread créé (retour)
  - attr : attributs du nouveau thread (ex. joignable, ou non détaché)
  - Le nouveau thread exécute la fonction *start_routine* en lui passant *arg* comme premier argument
  - Retourne 0 si succès, code non nul si échec

Modèle de threads

pthread_detach()

- int pthread_detach(pthread_t thrd);
  - Place le thread thrd en mode *detach* (le programme n'a pas besoin de savoir quand le thread termine)
  - Les ressources associées à thrd sont libérées lorsque son exécution se termine
  - Retourne 0 si tout s'est passé normalement
  - Sinon, retourne un code d'erreur entier

- pthread_t pthread_self(void);
  - Renvoie l'identifiant du thread courant

Modèle de threads

pthread_exit()

- Termine le thread appelant
- void pthread_exit(void *retval);
  - retval : valeur de retour du thread
    - Pourra éventuellement être consulté par un autre thread (pthread_join())
  - Appelé implicitement lorsque le thread retourne de la routine qu'il exécute

Sommaire

- Version multi-processus
  - Chaque alarme a son propre espace d'adressage, qui est copié du programme principal
  - Les secondes et le message peuvent être placées dans les variables locales
  - La modification des valeurs par le parent n'affecte pas l'enfant
  - Le programme principal doit dire au kernel de libérer les ressources utilisées par chaque enfant (waitpid)
- Version multi-thread
  - Tous les threads partagent le même espace d'adressage
  - Il faut allouer une nouvelle structure pour chaque alarme et la passer au nouveau thread
  - Pas besoin d'attendre la fin d'un thread (à moins d'avoir besoin de la valeur de retour)
  - Detach : les ressources du thread vont être libérées immédiatement après la fin de son exécution

CYCLE DE VIE D'UN THREAD

Retour sur les threads

- De façon générale, dans un programme, un thread
  - Est représenté par un identifiant de thread, une variable de type *pthread_t*
  - Débute par un appel à une fonction (fournie par le programmeur) qui
    - Prend un seul paramètre de type void *
    - Retourne une valeur de type void *
  - Est créé en passant l'adresse de la fonction et la valeur du paramètre à *pthread_create()*
  - Se termine
    - À l'appel de *pthread_exit()*
    - Lorsque la fonction associée termine son exécution
  - Est détruit (ou recyclé)
    - Lors de sa terminaison s'il est détaché (mode *detach*)
    - Lors d'un appel à *pthread_join()*

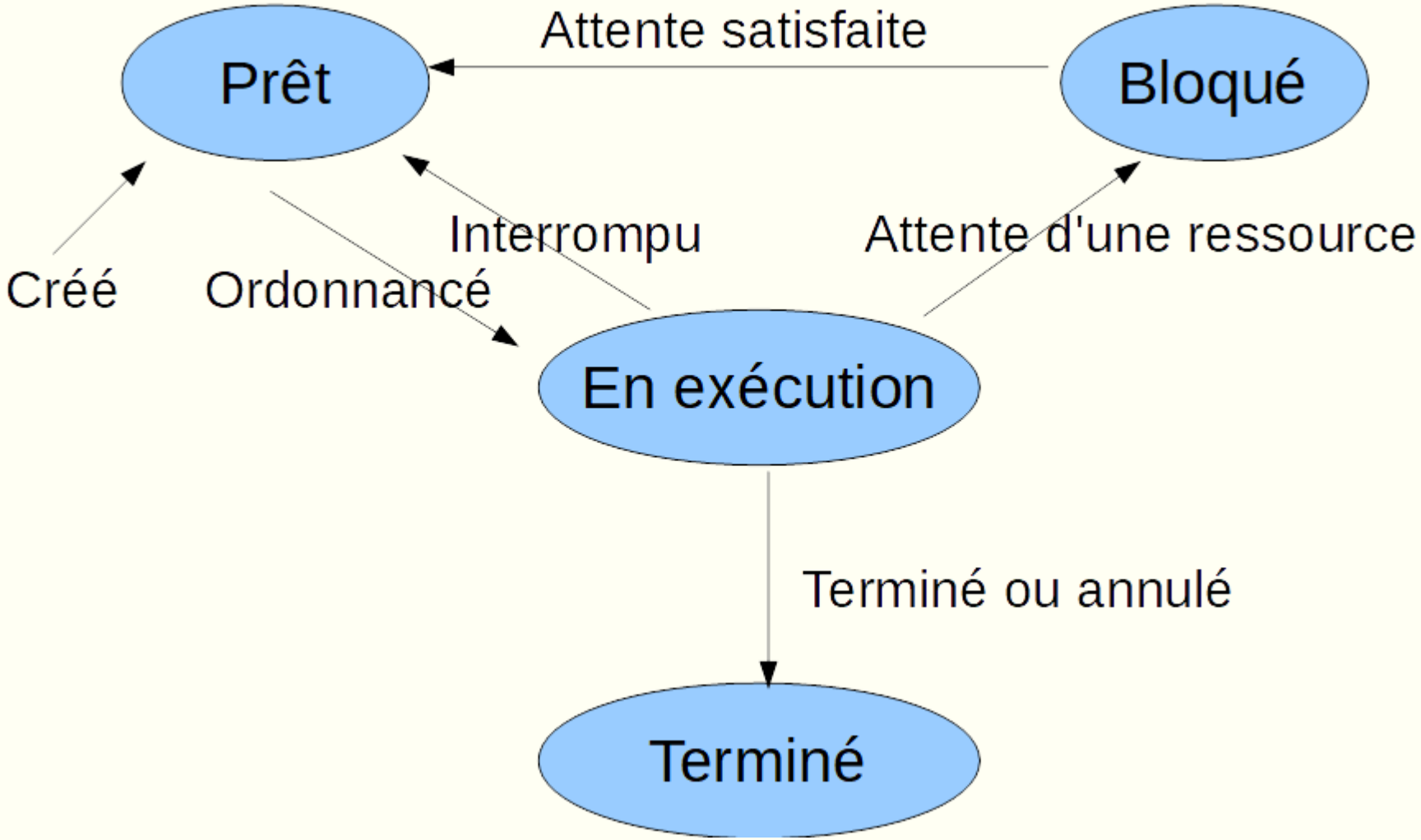
- Dans un programme multi-thread, le main est le *thread initial*, ou *thread principal*
- Comportement d'un processus traditionnel
  - Lorsque le main termine son exécution, tous les threads du programme sont détruits

Le cycle de vie d'un thread

- Un thread peut être dans un des états suivants

État	Signification
Prêt (Ready)	Le thread peut s'exécuter, mais il attend un processeur. Il peut avoir été débloqué ou interrompu (préempté) par un autre thread.
En exécution (Running)	Le thread est en cours d'exécution. Sur un multiprocesseur, il peut y avoir plus d'un thread en exécution sur un processus
Bloqué (Blocked)	Le thread est incapable de s'exécuter parce qu'il attend quelque chose (mutex, E/S, etc.)
Terminé (Terminated)	Le thead a terminé en retournant de sa fonction de démarrage, en appelant pthread_exit ou en ayant été annulé. Il n'est pas détaché et n'a pas été joint (join)

Transitions d'état d'un thread



Création

```
void *routine_thread(void *arg) {
        return arg;
}

int main(int argc, char *argv) {
        pthread_t thread_id;
        void *resultat_thread;
        int statut;

        statut = pthread_create(&thread_id, NULL, routine_thread, NULL);
        if (statut != 0)
                fprintf(stderr, "Probleme creation thread\n");

        statut = pthread_join(thread_id, &resultat_thread);
        if (statut != 0)
                fprintf(stderr, "Probleme thread join\n");
        if (resultat_thread == NULL)
                return 0;
        else
                return 1;
}
```

Crée un nouveau thread

Suite à l'appel, le thread créé devient en mode « prêt »

Pas de synchronisation entre
- Le retour de pthread_create() du thread créateur
- L'ordonnancement du thread créé

Le thread peut donc démarrer et même se terminer avant le retour du thread créateur !!!

Démarrage

```
void *routine_thread(void *arg) {
        return arg;
}

int main(int argc, char *argv) {
        pthread_t thread_id;
        void *resultat_thread;
        int statut;

        statut = pthread_create(&thread_id, NULL, routine_thread, NULL);
        if (statut != 0)
                fprintf(stderr, "Probleme creation thread\n");

        statut = pthread_join(thread_id, &resultat_thread);
        if (statut != 0)
                fprintf(stderr, "Probleme thread join\n");
        if (resultat_thread == NULL)
                return 0;
        else
                return 1;
}
```

Exécution de la routine spécifiée en paramètre de pthread_create()

La routine est appelée avec la valeur du paramètre spécifié lors de l'appel de pthread_create()

Exécution, blocage et terminaison

- Un thread ne peut généralement pas demeurer en exécution tout le temps de son existence
- Peut s'endormir
  - Parce qu'il nécessite une ressource qui n'est pas disponible
    - Mutex, variable de condition, E/S, opération système, ...
    - il tombe alors en mode *bloqué*
  - Parce que le système a assigné un autre thread au processeur sur lequel il s'exécute
    - il est interrompu, ou préempté, et tombe alors en mode *prêt*
- Terminaison
  - Retour de sa fonction associée
  - Appel de *pthread_exit()*
  - Appel de *pthread_cancel()* par un autre thread qui en possède l'identifiant
  - S'il a été détaché, il est immédiatement détruit
  - Sinon, il tombe en mode *terminé* et demeurera disponible pour être joint (*pthread_join()*) par un autre thread
    - *zombie* (mort mais toujours existant)

Destruction (ou recyclage)

- Se produit
  - À la fin de la routine si le thread est détaché
  - Demeure en mode terminé jusqu'à ce que son identifiant soit passé à *pthread_detach()* ou *pthread_join()*
- Libère les ressources
  - Mémoire locale (automatique), pile, etc...
  - Attention : ne pas retourner un pointeur sur une variable en mémoire locale !
  - Attention : libérer les ressources dynamiques !

PROCHAIN COURS

PTHREADS : LA SUITE
GESTION DES SYNCHRONISATIONS