Parallélisme en C

Bilal Chebaro

Patrice Torguet

Plan

- **Introduction**
- Notions de processus/thread
- Création de processus
- Communication entre processus
- Créations de threads
- Synchronisation entre threads

Introduction

- Une application est **multitâche** quand elle exécute (ou peut exécuter) plusieurs parties de son code en même temps
- A un instant donné, l'application comporte plusieurs **points d'exécution** liés aux différentes parties qui s'exécutent en parallèle.
- Tous les systèmes d'exploitation modernes sont multitâches et permettent l'exécution d'applications multitâches
- Sur une machine monoprocesseur/mono-coeur cette exécution en parallèle est simulée
- Le multitâche s'appuie sur les processus ou les threads (fils d'exécution en français)
- Si le système est **préemptif**, il peut à tout moment suspendre un processus/thread pour en exécuter un autre
- Sinon, les processus/threads doivent indiquer explicitement ou implicitement (par exemple quand ils sont bloqués par des entrées/sorties) qu'ils veulent passer la main à un autre processus/thread

Introduction

- Exemple de multitâche à l'intérieur d'une application :
  - l'interface graphique peut lancer un thread pour charger une image pendant qu'elle continue de traiter les événements générés par des actions de l'utilisateur
  - une application serveur qui attend les demandes de connexions venant des clients peut lancer un processus/thread pour traiter les demandes de plusieurs clients simultanément
  - la multiplication de 2 matrices (m,p) et (p,n) peut être effectuée en parallèle par m * n threads
- Utilité du multitâche :
  - amélioration des performances en répartissant les différentes tâches sur différents processeurs
  - profiter des temps de pause d'une tâche (attente d'entrées/sorties ou d'une action utilisateur) pour faire autre chose
  - réagir plus vite aux actions de l'utilisateur en rejetant une tâche longue et non-interactive dans un autre thread (exemple : correction de l'orthographe pour un éditeur de texte)

Introduction

- Problèmes du multitâche :
  - il est souvent plus difficile d'écrire un programme multitâche
  - et surtout, il est plus difficile de déboguer un programme qui utilise le multitâche

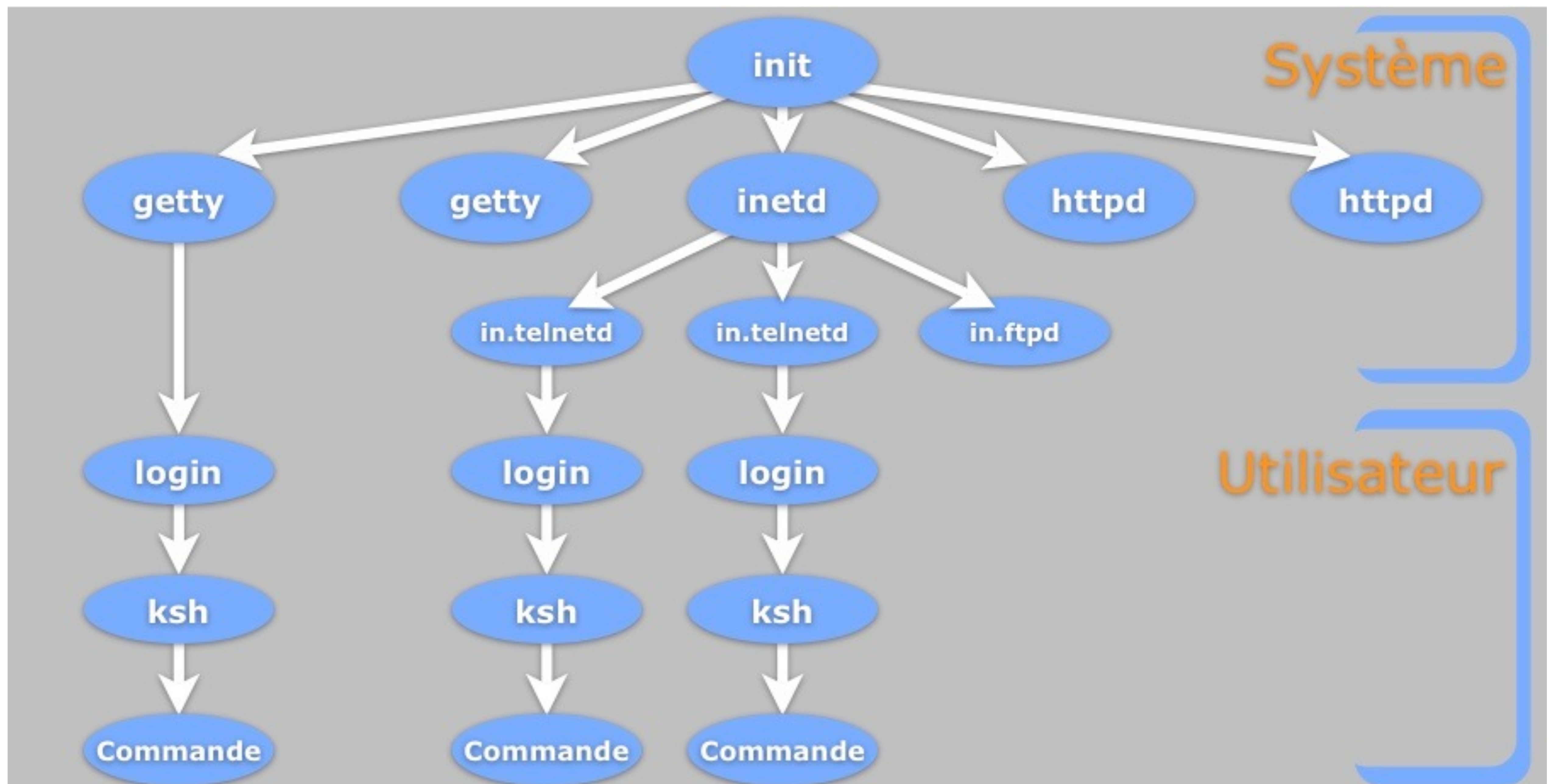
Plan

- Introduction
- **Notions de processus/thread**
- Création de processus
- Communication entre processus
- Créations de threads
- Synchronisation entre threads

Notion de Processus

- Un processus est l’image en mémoire d’un programme en cours d’exécution
- Un même programme lancé deux fois génère deux processus différents
- Chaque processus est identifié par un numéro unique appelé PID (Process IDentifier)
- Un processus peut créer d’autres processus. On les appelle ses fils. Ceci crée une hiérarchie père-fils.

Notion de Processus



Notion de Processus

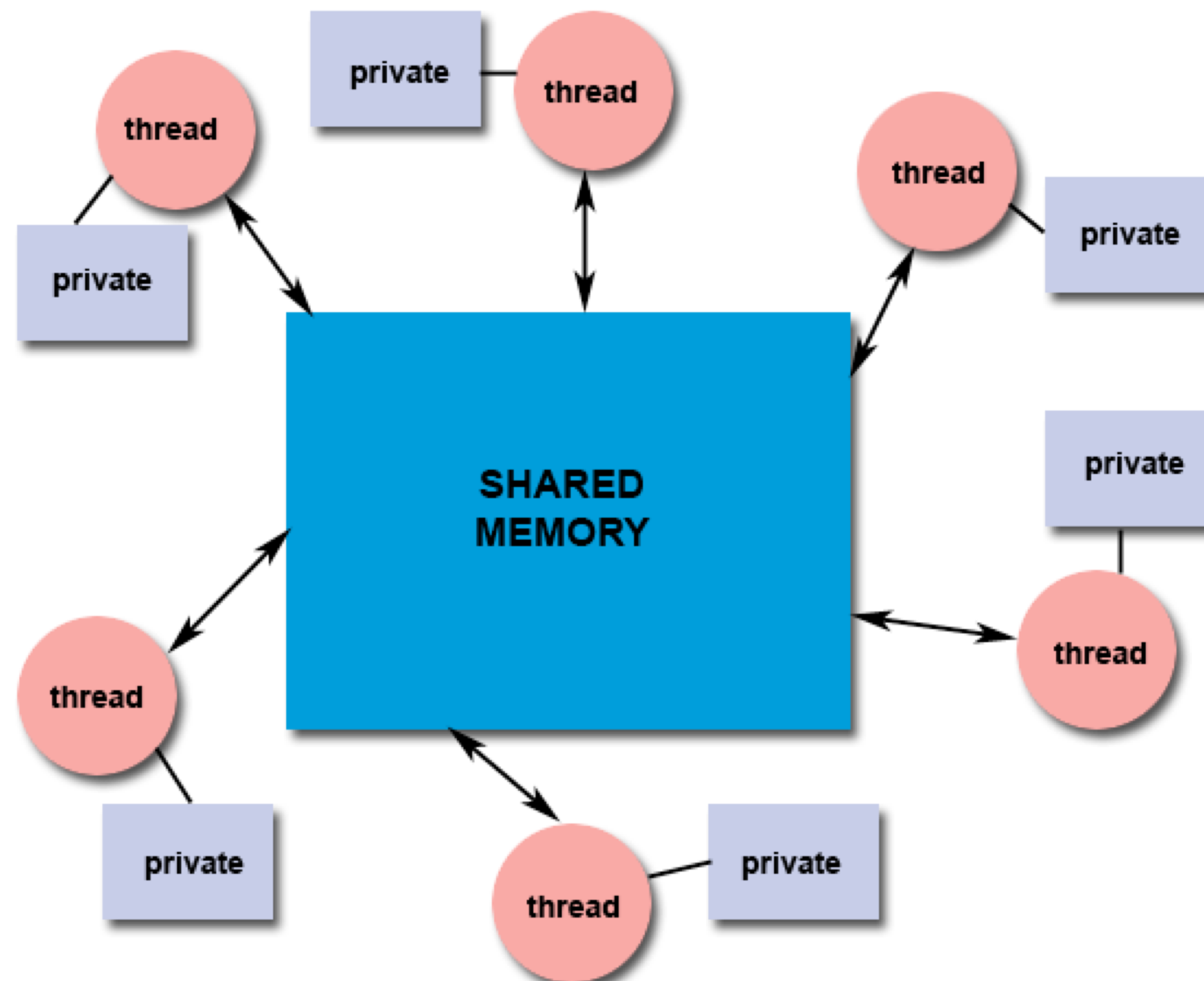
- Chaque processus a un espace d'adressage (espace mémoire où sont rangées les variables) distinct
- Avantage :
  - Pas de problème d'accès simultané à une même variable
- Inconvénient :
  - Il faut des outils spécifiques pour faire communiquer les processus : IPC (Inter Process Communications)
  - La création d'un processus et le passage d'un processus à un autre sont des opérations lourdes
- La communication entre processus peut être réalisée par :
  - des signaux
  - des tubes (pipes)
  - de la mémoire partagée
  - des sockets
  - ...

Notion de Thread

- Le mot thread peut se traduire par tâche ou fil d'exécution
  - Attention, Thread est différent du terme tâche dans «système multitâches»
- Permet de faire fonctionner plusieurs portions de code simultanément dans le même processus => partagent le même espace mémoire
- Avantages :
  - Correcteur grammatical d'un traitement de texte
  - Accélération de calculs matriciels sur un multi-processeur/coeur
  - ...
- Inconvénients :
  - Plusieurs threads peuvent modifier une même variable en même temps ce qui peut générer des incohérences
  - On doit donc les synchroniser avec des outils (mutex, conditions, sémaphores, barrières...)

Notion de Thread

- Chaque thread dispose d’une partie privée de mémoire composée de sa pile et éventuellement de données privées supplémentaires



Normalisation POSIX des Threads

- Historiquement : de nombreux OS permettaient de créer et de gérer des threads
  - Mais, il y avait de nombreuses différences
  - Comment faire des applications portables entre OS ?
  - Besoin de normalisation
    - Standard IEEE POSIX 1003.1c (1995)
    - Ensemble de types et de fonctions C
      - définis dans pthread.h
      - implémentés par une bibliothèque spécifique (en général libpthread) ou la bibliothèque standard (libc)

Comparaison Processus/Threads

Platform	fork()			pthread_create()		
	real	user	sys	real	user	sys
Intel 2.6 GHz Xeon E5-2670 (16cpus/node)	8.1	0.1	2.9	0.9	0.2	0.3
Intel 2.8 GHz Xeon 5660 (12cpus/node)	4.4	0.4	4.3	0.7	0.2	0.5
AMD 2.3 GHz Opteron (16cpus/node)	12.5	1.0	12.5	1.2	0.2	1.3
AMD 2.4 GHz Opteron (8cpus/node)	17.6	2.2	15.7	1.4	0.3	1.3
IBM 4.0 GHz POWER6 (8cpus/node)	9.5	0.6	8.8	1.6	0.1	0.4
IBM 1.9 GHz POWER5 p5-575 (8cpus/node)	64.2	30.7	27.6	1.7	0.6	1.1
IBM 1.5 GHz POWER4 (8cpus/node)	104.5	48.6	47.2	2.1	1.0	1.5
INTEL 2.4 GHz Xeon (2 cpus/node)	54.9	1.5	20.8	1.6	0.7	0.9
INTEL 1.4 GHz Itanium2 (4 cpus/node)	54.5	1.1	22.2	2.0	1.2	0.6

« Timings reflect 50,000 process/thread creations, were performed with the `time` utility, and units are in seconds, no optimization flags »

https://computing.llnl.gov/tutorials/pthreads/#WhyPthreads

Comparaison Processus/Threads

Platform	MPI Shared Memory Bandwidth (GB/sec)	Pthreads Worst Case Memory-to-CPU Bandwidth (GB/sec)
Intel 2.6 GHz Xeon E5-2670	4.5	51.2
Intel 2.8 GHz Xeon 5660	5.6	32
AMD 2.3 GHz Opteron	1.8	5.3
AMD 2.4 GHz Opteron	1.2	5.3
IBM 1.9 GHz POWER5 p5-575	4.1	16
IBM 1.5 GHz POWER4	2.1	4
Intel 2.4 GHz Xeon	0.3	4.3
Intel 1.4 GHz Itanium 2	1.8	6.4

« MPI libraries usually implement on-node task communication via shared memory, which involves at least one memory copy operation (process to process) »

https://computing.llnl.gov/tutorials/pthreads/#WhyPthreads

Plan

- Introduction
- Notions de processus/thread
- **Création de processus**
- Communication entre processus
- Créations de threads
- Synchronisation entre threads

Création de Processus

- Fonction pid_t fork(void)
- Cette fonction crée un nouveau processus qui est une copie quasi-identique du processus courant
- Toutes les variables globales, locales, dynamiques (crées avec malloc) sont dupliquées
- Le code exécuté est identique
- La table des fichiers ouverts est recopiée
- Les seules différences :
  - ils ont des PID différents
  - la valeur retour de la fonction fork est différente dans le père (il s'agit du pid du fils) et dans le fils (valeur 0)
- On peut utiliser les fonctions :
  - pid_t getpid(void) : pour récupérer le PID du processus courant
  - pid_t getppid(void) : pour récupérer le PID du père du processus courant

Création de Processus

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

int main (int argc, char *argv[])
{   pid_t pid;

    pid = fork();
    if (pid == -1) {
          /* erreur - impossible de créer un fils */
          perror("fork error");
          exit(1);
    } else if (pid == 0) {
          /* code du fils */
          printf("Fils >> Mon PID est : %d\n", getpid());
          printf("Fils >> Le PID de mon père est : %d\n", getppid());
          exit(0);
    } else {
          /* code du père */
          printf("Père >> PID du fils : %d\n", pid);
          printf("Père >> Mon PID est : %d\n", getpid());
          sleep(1);
          exit(0);
    }
}
```

Création de Processus

P1

```
void main() {
  ...
  pid = fork();
  ...
  if(pid == 0) {
    ...
    ...
    exit(0);
  } else {
    ...
    exit(0);
  }
}
```

P2

```
void main() {
  ...
  pid = fork();
  ...
  if(pid == 0) {
    ...
    ...
    exit(0);
  } else {
    ...
    exit(0);
  }
}
```

Création de Processus

- Note : il est possible de remplacer l'image du programme du processus courant par celle d'un autre programme en utilisant les fonctions exec (cf. man 3 exec)
- Ceci est en général utilisé après un fork.
- Exemple : int execl(const char *path, const char *arg, ...);
  - path est le nom du nouveau programme
  - arg, ... permet de lui donner des paramètres
  - attention : le premier argument doit être le nom du programme et il faut terminer la liste par NULL
- Par exemple : execl("/bin/ls","ls","-l",NULL);
  - exécutera la commande ls -l dans le répertoire courant en affichant le résultat sur la sortie standard

Attente de la fin d'un processus

- pid_t wait(int *code_fin) : sert à attendre la fin d'un processus fils et à récupérer son code de fin (renvoyé par la fonction exit() et d'autres informations – voir transparent suivant)
- Si on n'attends pas la fin d'un processus fils celui-ci devient orphelin (si le père s'arrête avant) ou zombie (si le père continue de fonctionner).
- Un zombie disparait à chaque appel de wait.
- Les processus orphelins sont adoptés automatiquement par un processus système (init ou launchd de PID 1)
- On peut aussi appeler la fonction pid_t waitpid(pid_t pid, int *code_fin, int options), pour attendre uniquement la fin d'un fils en donnant son pid ou -1 pour tous les fils. En précisant WNOHANG comme option on n'attends pas et donc on récupère le code de fin d'un processus fils qui s'est déjà terminé (un zombie).
  Exemple : waitpid(-1, &code_fin, WNOHANG) récupère le code de fin de n'importe quel fils zombie.

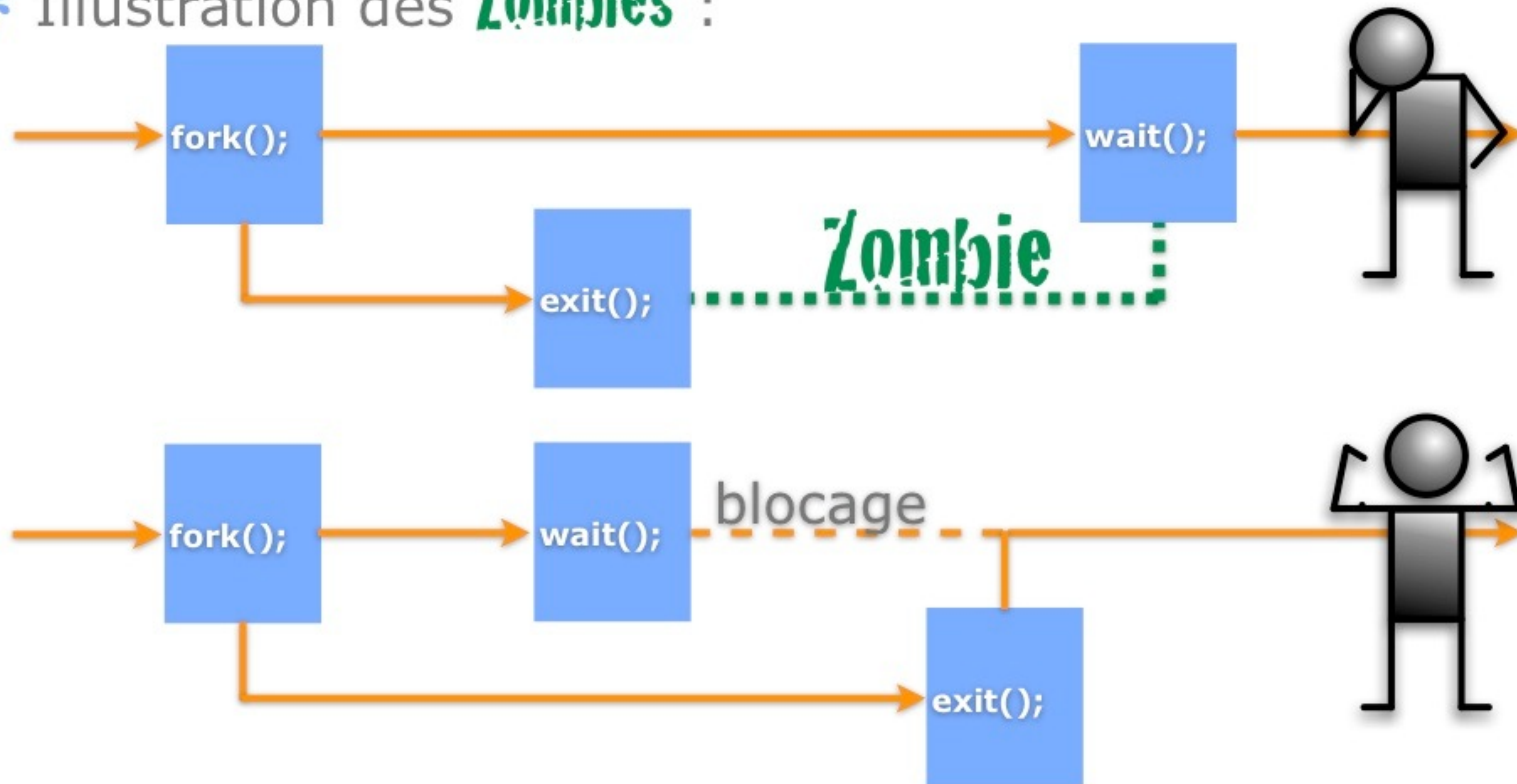
Attente de la fin d’un processus

- Le code_fin contient diverses informations (dans les différents octets de l’int) qu’il convient d’analyser avec les macros suivantes :

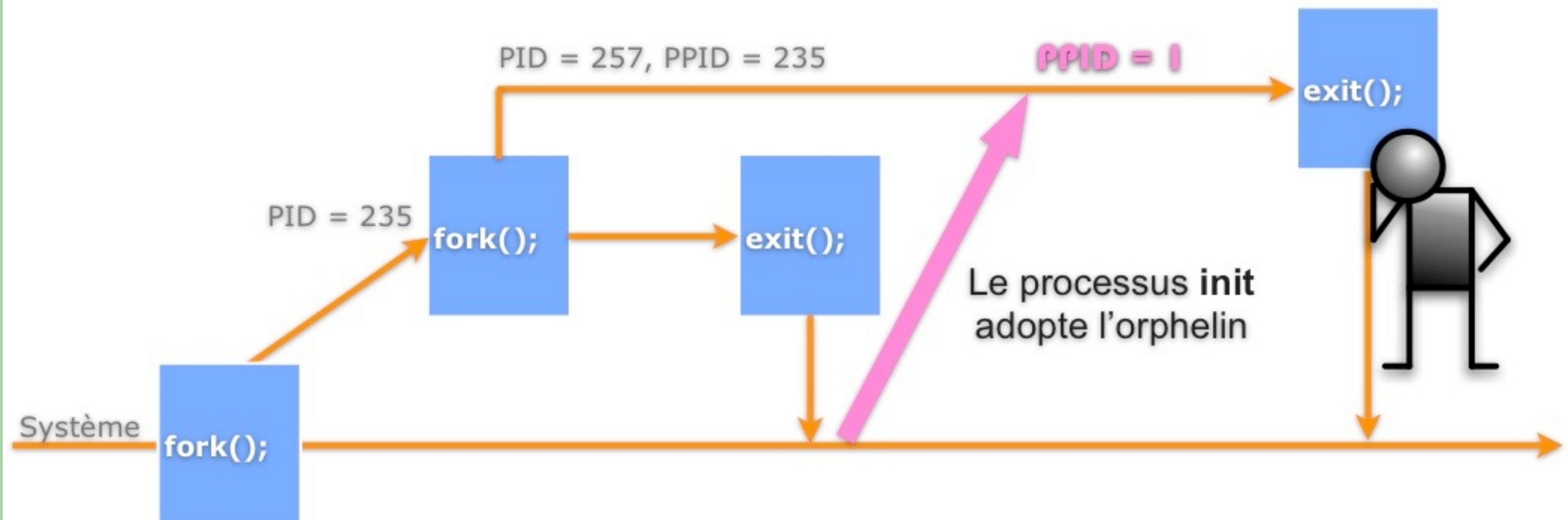
  - WIFEXITED(code_fin) : renvoie vrai si le fils s'est terminé normalement, c'est-à-dire par un appel à exit(3) ou _exit(2), ou bien par un retour de main().
  - WEXITSTATUS(code_fin) : renvoie le code de sortie du fils. Cette macro ne peut être évaluée que si WIFEXITED a renvoyé vrai.

  - WIFSIGNALED(status) : renvoie vrai si le fils s'est terminé à cause d'un signal. Dans ce cas, il y a d’autres macros pour savoir ce qui s’est produit (voir les pages du manuel : http://manpagesfr.free.fr/man/man2/wait.2.html)

Attente de la fin d'un processus



Attente de la fin d'un processus



Attente de la fin d’un processus

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

int main (int argc, char *argv[])
{    pid_t pid;
     int code_fin;

     pid = fork();
     if (pid == -1) {
          /* erreur - impossible de créer un fils */
          perror("fork error");
          exit(1);
     } else if (pid == 0) {
          /* code du fils */
          printf("Fils >> Mon PID est : %d\n", getpid());
          printf("Fils >> Le PID de mon père est : %d\n", getppid());
          exit(0);
     } else {
          /* code du père */
          printf("Père >> PID du fils : %d\n", pid);
          printf("Père >> Mon PID est : %d\n", getpid());
          wait(&code_fin);
          printf("Père >> Fin du fils\n");
          exit(0);
     }
}
```

Plan

- Introduction
- Notions de processus/thread
- Création de processus
- **Communication entre processus**
- Créations de threads
- Synchronisation entre threads

Communication entre processus : signaux

- Les signaux permettent une communication basique entre processus
- Il existe une trentaine de signaux définis dans signal.h (attention les valeurs changent d'un UNIX à un autre => utiliser les constantes)
- Par défaut, un processus qui reçoit un signal meurt, stoppe (équivalent d'un Ctrl-Z) ou l'ignore (en fonction du signal - cf. **man 7 signal** sous Linux ou **man signal** sous BSD/MacOSX)
- Mais, on peut modifier ce comportement pour la plupart des signaux (SIGUSR1, SIGUSR2, SIGALRM... mais pas SIGKILL et SIGSTOP) grâce à la fonction sigaction (ou signal qui est obsolète)
- Pour envoyer un signal en C on utilisera la fonction kill :
  int kill(pid_t pid, int sig);
  qui envoie le signal **sig** au processus identifié par **pid**.

Signaux

- La fonction sigaction est définie comme ceci :
  int sigaction(int sig, struct sigaction *p_action, struct sigaction *p_action_anc);
  Avec : struct sigaction {
  void (*sa_handler)(int); sigset_t sa_mask; int sa_flags;
  }
- void (*sa_handler)(int) est un pointeur vers une fonction qui prend un entier en paramètre
- sigaction permet donc de préciser une fonction qui sera appelée lors de la **prochaine** réception du signal **sig**
- On peut aussi préciser à la place d'une fonction : SIG_IGN pour ignorer un signal ou SIG_DFL pour revenir au comportement par défaut
- La fonction renvoie dans p_action_anc, la fonction précédemment positionnée (qui peut être SIG_IGN)

Signaux

- La fonction positionnée est exécutée au moment même de la réception du signal (interruption logicielle) puis ensuite le programme reprend son exécution normale (si il n'est pas mort ou stoppé)
- Si un signal identique est envoyé au processus pendant l'invocation de la fonction, ce signal est bloqué temporairement et sera traité dès que possible (on dit que le signal est masqué)
- Note : pour redémarrer un processus stoppé on peut regarder son numéro de job avec la commande **jobs** et taper **fg %1** (si c'était le job numéro 1) pour le redémarrer dans le terminal ou **bg %1** pour le redémarrer sans lien avec le terminal
- Pour attendre un signal on peut utiliser la fonction pause() qui bloque le processus courant jusqu'à la réception d'un signal

Signaux

- Les gestionnaires de signaux sont hérités par les processus fils
- Limites :
  - Aucune mémorisation du nombre de signaux reçus : si on reçoit 10 signaux SIGUSR1 alors que ce signal est masqué, il n'y aura qu'un seul appel du gestionnaire après le masquage
  - Aucune mémorisation de la date d'émission du signal : ils sont traités dans l'ordre des numéros de signaux
  - Aucun moyen de connaitre le pid du processus émetteur du signal
  - Aucun moyen de transmettre des données avec un signal
  - Le comportement de nombreuses fonctions est aléatoire dans le gestionnaire de signal : en général on ne fait que positionner des booléens

Signaux

```
...
#include <signal.h>
void sig_handler(int sig) {
     printf("Signal >> signal : %d reçu par le processus %d\n", sig, getpid());
}
int main (int argc, char *argv[])
{   pid_t pid; int code_fin;
    struct sigaction action;
    action.sa_handler=sig_handler;
    sigemptyset(&action.sa_mask);
    /* positionnement du gestionnaire de signal */
    if (sigaction(SIGUSR1,&action,(struct sigaction *)NULL) == -1) {
          perror("signal error"); exit(1);
    }
    pid = fork();
    if (pid == -1) {
          /* erreur - impossible de créer un fils */
          perror("fork error");
          exit(1);
    } else if (pid == 0) { /* code du fils */
          pause(); /* attente du signal */
          exit(0);
    } else { /* code du père */
          sleep(1);
          kill(pid, SIGUSR1); /* envoi du signal au fils */
          wait(&code_fin); exit(0);
    }
}
```

Signaux

P1

```
void sig_handler(int sig) {
   printf(...);
}

void main() {
  ...
  sigaction(...)
  pid = fork();
  ...
  sleep(1);
  kill(pid, SIGUSR1);
  ...
  exit(0);
}
```

P2

```
void sig_handler(int sig) {
   printf(...);
}

void main() {
  ...
  pid = fork();
  ...
  pause();
  exit(0);
}
```

Communication entre processus : tubes

- Pour dépasser les limites des signaux de nombreux outils ont été créés
- Ici on ne présente que les tubes de communication anonymes (pipes)
- Ils permettent le transfert de données entre deux processus parents en utilisant le modèle FIFO (First In/First Out) : les données sont lues dans l'ordre où elles ont été écrites
- Ils ne sont pas persistants, un tube n'existe que tant que le processus qui l'a créé existe
- Ils sont très utilisés par le shell (utilisation de «|»)
- Les accès à un tube sont synchronisés par le noyau : une seule lecture/écriture est autorisée à la fois, si le tube est vide la lecture est bloquante

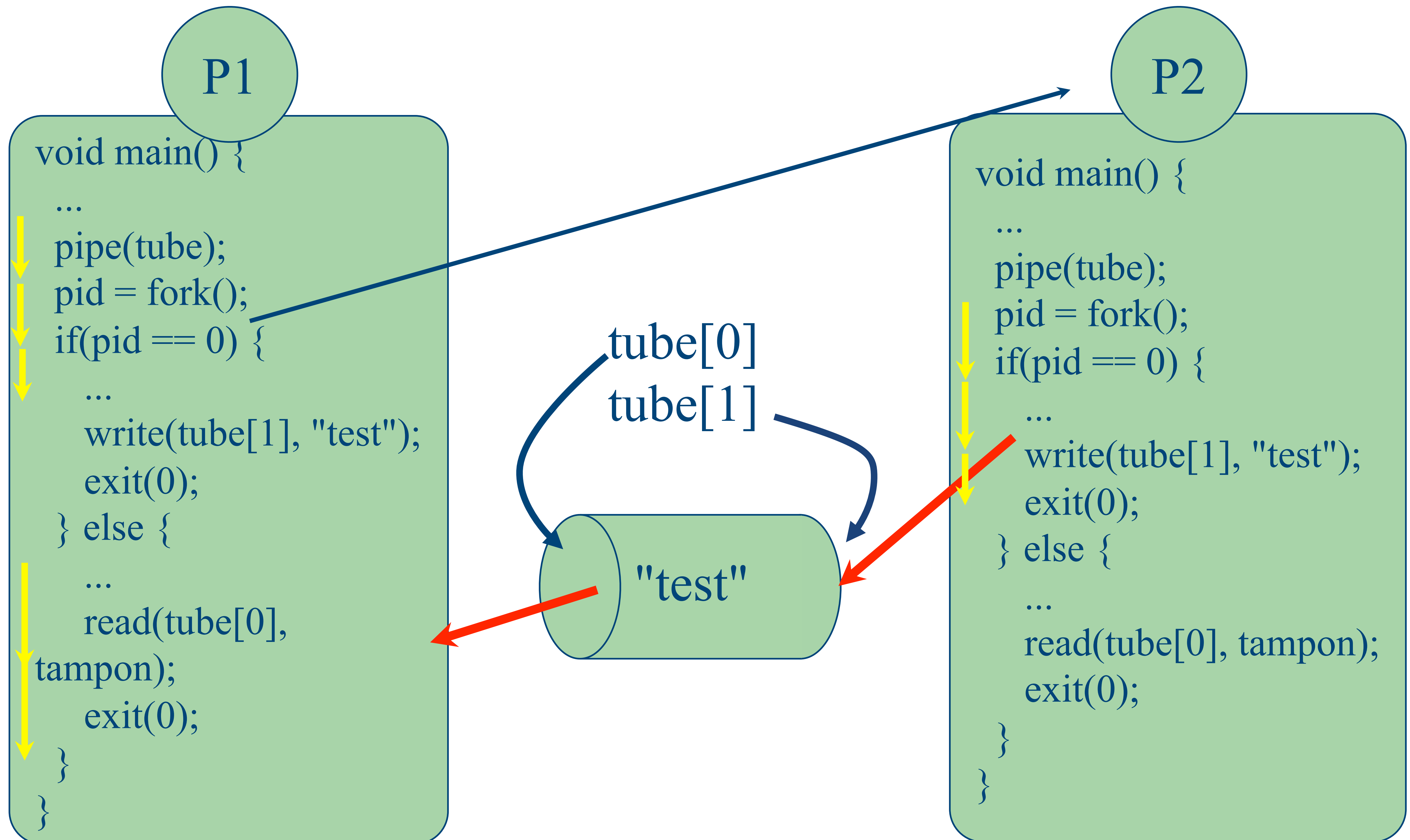
Tubes

- Les tubes ressemblent donc à des fichiers
- D'ailleurs on utilise les appels systèmes **read/write** pour lire et écrire dans un tube et **close** pour le fermer
- Mais : on ne peut lire l'information qu'une seule fois (pas de rewind ici) !
- La création d'un tube se fait avec la fonction pipe :
  int pipe(int pipefd[2]);
- Cette fonction crée deux descripteurs de fichiers (comme ceux renvoyés par open)
- le premier pipefd[0] est utilisé pour les lectures
- le second pipefd[1] est utilisé pour les écritures
- pipe renvoie 0 si ça a marché et -1 en cas d'erreur (et positionne errno)
- En général, un processus utilisera le tube en écriture (et fermera le premier descripteur) et un autre en lecture (en fermant le second)

Tubes

```
int main (int argc, char *argv[])
{   pid_t pid;
    int code_fin;
    int tube[2];
    char tampon[32];
    if(pipe(tube) == -1) {
     perror("pipe error");
          exit(1);
    }
    if ((pid = fork()) == -1) {
          /* erreur - impossible de créer un fils */
          perror("fork error");
          exit(1);
    } else if (pid == 0) { /* code du fils */
          close(tube[0]);    /* ferme le descripteur en lecture */
          write(tube[1], "test d'écriture", strlen("test d'écriture")+1);
          close(tube[1]);    /* ferme le descripteur en écriture */
          exit(0);
    } else {
          /* code du père */
          close(tube[1]);    /* ferme le descripteur en écriture */
          read(tube[0], tampon, 32);
          close(tube[0]);    /* ferme le descripteur en lecture */
          printf("J'ai lu : %s\n", tampon);
          wait(&code_fin);
          exit(0);
    }
}
```

Tubes



Tubes

- Remarques :
  - si tous les processus partageant un tube ont fermé le descripteur servant à lire, une écriture sur le tube provoque une exception SIGPIPE (qui par défaut termine le programme)
  - si tous les processus partageant un tube ont fermé le descripteur servant à écrire, une lecture sur le tube renverra 0 (indiquant la fin de fichier)

Plan

- Introduction
- Notions de processus/thread
- Création de processus
- Communication entre processus
- **Créations de threads**
- Synchronisation entre threads

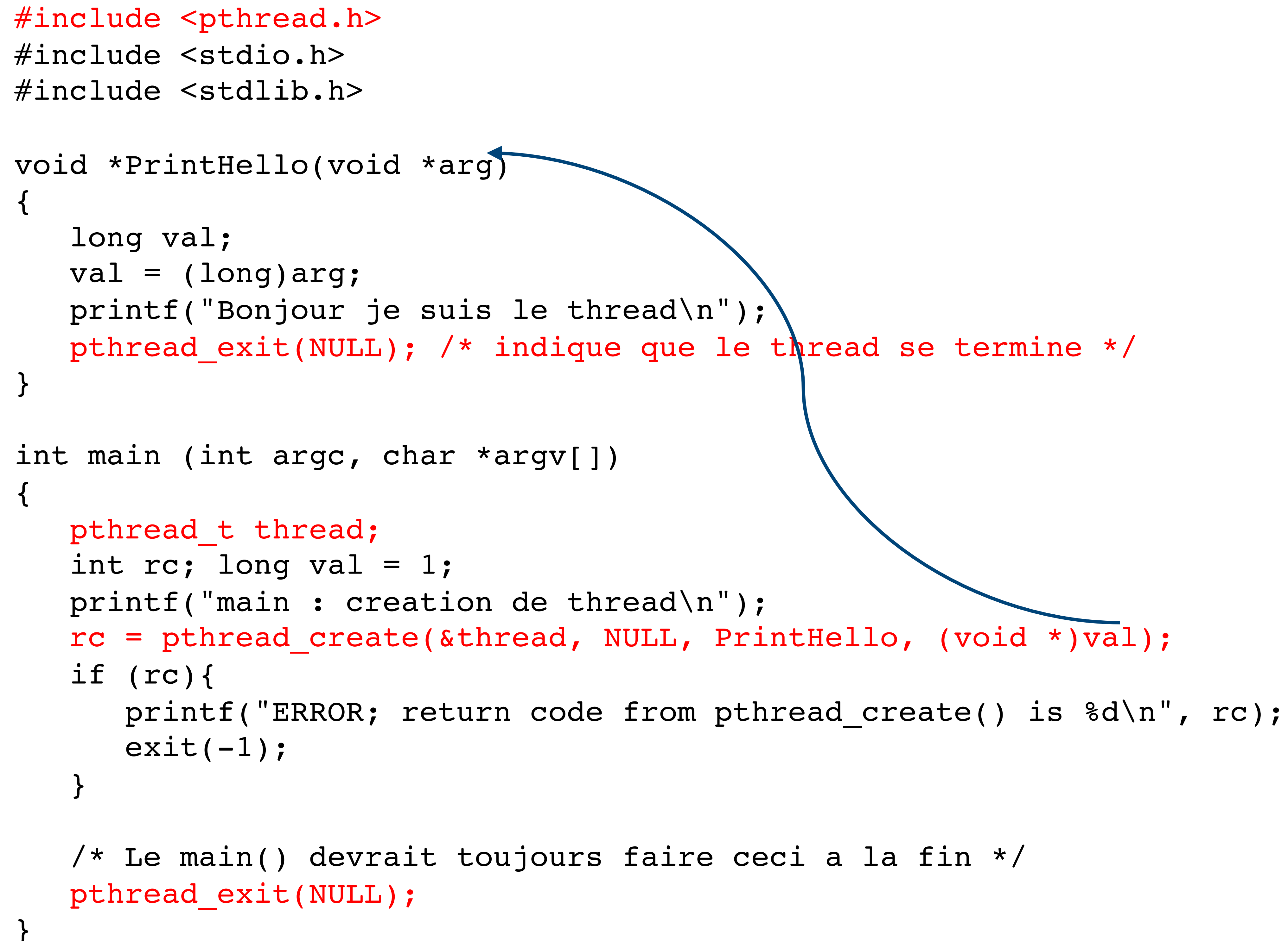
Création de Thread

- Fonction pthread_create :

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

- Ceci est un pointeur de fonction : void *(*start_routine)(void*)
- C'est cette fonction qui sera exécutée par le thread que l'on va créer
- thread est l'identificateur unique du thread qui sera renseigné en sortie de l'appel de pthread_create
- attr sert à choisir des attributs spécifiques pour le thread (en général on utilise NULL pour avoir les attributs par défaut) en utilisant les fonctions pthread_attr_setXXXX (état détaché ou joignable, ordonnancement, attributs de la pile...)
- arg est un paramètre optionnel qui sera passé à la fonction que va exécuter le thread
- Pour compiler un programme utilisant les pthreads sous Linux il faut taper **gcc -pthread main.c -o main**

Création de Thread



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

void *PrintHello(void *arg)
{
   long val;
   val = (long)arg;
   printf("Bonjour je suis le thread\n");
   pthread_exit(NULL); /* indique que le thread se termine */
}

int main (int argc, char *argv[])
{
   pthread_t thread;
   int rc; long val = 1;
   printf("main : creation de thread\n");
   rc = pthread_create(&thread, NULL, PrintHello, (void *)val);
   if (rc){
      printf("ERROR; return code from pthread_create() is %d\n", rc);
      exit(-1);
   }

   /* Le main() devrait toujours faire ceci a la fin */
   pthread_exit(NULL);
}
```

Création de Thread

T0

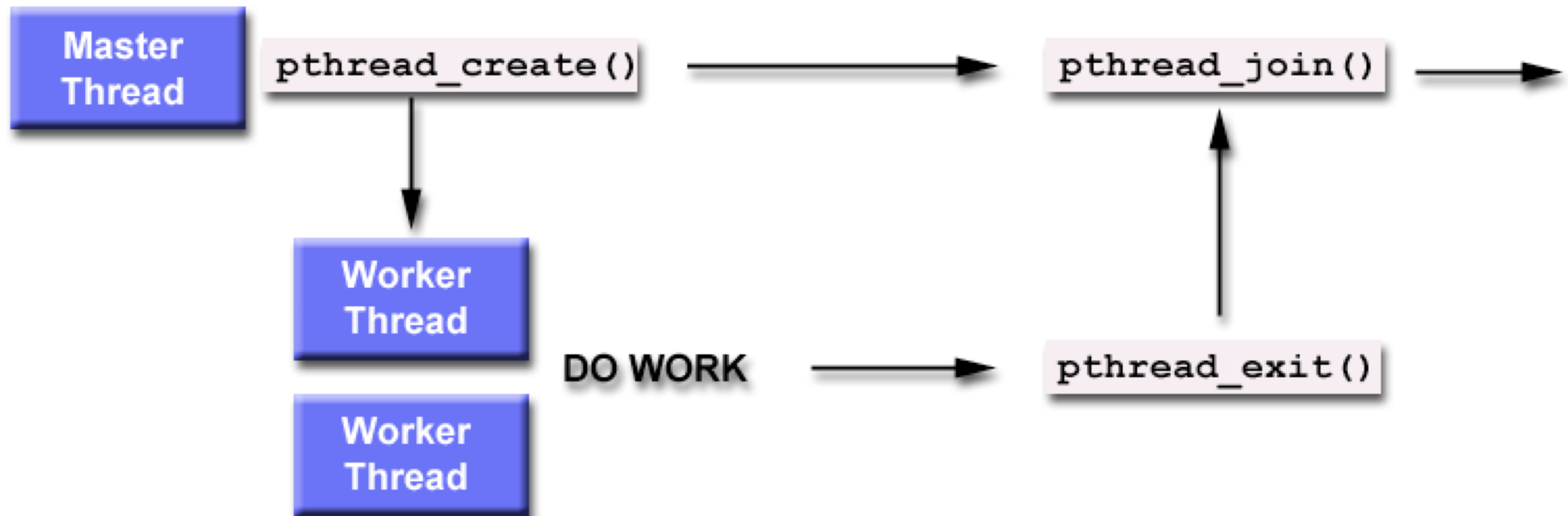
```
void main() {
  ...
  pthread_create(...);
  ...
  pthread_exit(NULL);
}
```

T1

```
void *PrintHello(void
*arg)
{
  long val;
  val = (long)arg;
  printf("Bonjour...");
  pthread_exit(NULL);
}
```

Attente de la fin d’un thread

- pthread_join sert à attendre la fin d’un thread
- seuls les threads créés avec l’attribut joignable peuvent être attendus
- il vaut mieux préciser cela à la création plutôt que de se baser sur les attributs par défaut.



Attente de la fin d’un thread

```
pthread_t thread;
pthread_attr_t attr;
int rc;
long t;
void *statut; /* sert à renvoyer quelque chose - ici ce sera un long */

/* Initialisation des attributs et choix de l'attribut état joignable */
pthread_attr_init(&attr);
pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);

rc = pthread_create(&thread[t], &attr, BusyWork, (void *)t);
if (rc) { printf("ERROR; return code from pthread_create()
             is %d\n", rc); exit(-1);
}

/* On libère la mémoire des attributs et on attends la fin du thread */
pthread_attr_destroy(&attr);
rc = pthread_join(thread, &statut);
if (rc) { printf("ERROR; return code from pthread_join()
             is %d\n", rc); exit(-1);
}
printf("Main: le thread est terminé, sont statut était : %ld\n",
       (long)statut);
```

Plan

- Introduction
- Notions de processus/thread
- Création de processus
- Communication entre processus
- Créations de threads
- **Synchronisation entre threads**

Synchronisation entre Threads

- L'utilisation de threads peut entraîner des besoins de synchronisation pour éviter les problèmes liés aux accès simultanés aux variables

- En programmation parallèle, on appelle **section critique**, une partie du code qui ne peut être exécutée en même temps par plusieurs threads sans risquer de provoquer des anomalies de fonctionnement

Exemple de problème

- Si x = 2, le code x = x + 1;
  exécuté par 2 threads, peut donner en fin d'exécution 3 ou 4 suivant l'ordre d'exécution :

1. T1 : lit la valeur de x (2)
2. T2 : lit la valeur de x (2)
3. T1 : calcule x + 1 (3)
4. T2 : calcule x + 1 (3)
5. T1 : range la valeur calculée dans x (3)
6. T2 : range la valeur calculée dans x (3)

- x contient 3 au lieu de 4 !

Nécessité de synchroniser

- Il faut donc éviter l'exécution simultanée de sections critiques par plusieurs threads
- Il existe plusieurs outils de synchronisation
- Ici on présente : mutex et condition

Mutex

- Un mutex sert à protéger une section critique simple grâce à une exclusion mutuelle (un seul thread peut exécuter la section critique à un instant donné)
- Un mutex peut être verrouillé ou déverrouillé
- Si on essaie de verrouiller un mutex déjà verrouillé, l'opération est bloquante jusqu'à ce que le mutex soit déverrouillé
- Seul le thread qui a verrouillé un mutex peut le déverrouiller (on dit qu'il le possède)
- Lors du déverrouillage, si un ou plusieurs threads sont bloqués en attente du mutex, l'un d'eux est réveillé (s'il y en a plusieurs le choix du thread est plus ou moins fait au hasard)
- Quand il est créé, un mutex est déverrouillé

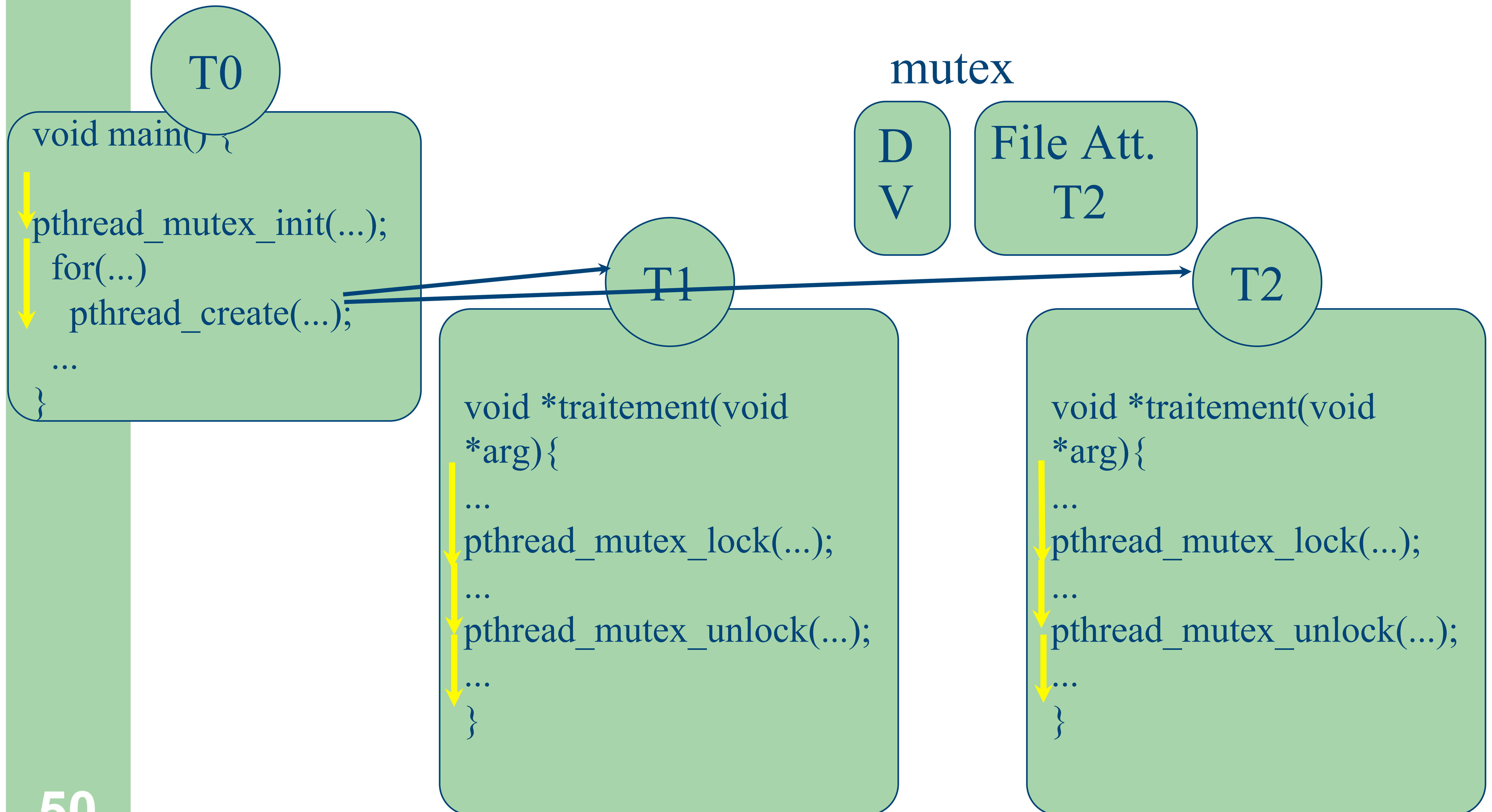
Mutex

- La création d’un mutex peut se faire de deux façons :
  - statiquement (lors de sa déclaration) :
    ```
    pthread_mutex_t mymutex =
            PTHREAD_MUTEX_INITIALIZER;
    ```
  - dynamiquement en utilisant la fonction :
    ```
    int pthread_mutex_init(pthread_mutex_t *mutex,
    pthread_mutexattr_t *attr);
    ```
    ce qui permet de choisir des attributs pour le mutex
- Pour détruire un mutex on utilisera :
  ```
  int pthread_mutex_destroy(pthread_mutex_t *mutex);
  ```
- Pour le verrouiller :
  ```
  int pthread_mutex_lock(pthread_mutex_t *mutex);
  ```
- Pour le déverrouiller :
  ```
  int pthread_mutex_unlock(pthread_mutex_t *mutex);
  ```

Mutex

```
long global=0; /* variable globale */
pthread_mutex_t mutex; /* mutex pour protéger la section critique */
void *traitement(void *arg)
{
   long val; int i; long tempo;
   val = (long)arg;
   for(i = 0; i < val; i++) { /* section critique */
        pthread_mutex_lock (&mutex);
        tempo = global;
        pthread_yield(); /* pour forcer une dé-synchronisation (en BSD : usleep(1);)*/
        global = tempo + i*val;
        printf("global vaut %ld\n", global);
        pthread_mutex_unlock (&mutex);
   }
   pthread_exit(NULL); /* indique que le thread se termine */
}
int main (int argc, char *argv[])
{
   pthread_t thread; int rc; long val = 10; int i;
   pthread_mutex_init(&mutex, NULL);
   for(i = 0; i<10; i++) { /* on crée 10 threads */
        pthread_create(&thread, NULL, traitement, (void *)val);
   }
   /* Le main() devrait toujours faire ceci à la fin */
   pthread_exit(NULL);
}
```

Mutex



Conditions

- Lorsqu'un programme est multi-tâche, la situation suivante peut se rencontrer :
  - Un thread t1 ne peut continuer son exécution que si une condition est remplie
  - Le fait que la condition soit remplie ou non dépend d'un autre thread t2
- Par exemple, t1 a besoin du résultat d'un calcul effectué par t2
- Une solution coûteuse serait que t1 teste la condition à intervalles réguliers
- Les conditions permettent de programmer plus efficacement ce genre de situation

Conditions

- L'utilisation des conditions demande un travail coopératif entre les threads t1 et t2 :
  1. Ils utilisent tout deux une même condition et un même mutex. La condition est créée :
     - statiquement :
       pthread_cond_t myconvar =
       PTHREAD_COND_INITIALIZER;
     - dynamiquement en utilisant :
       int pthread_cond_init(pthread_cond_t *cond, pthread_condattr_t *attr);
       ce qui permet de choisir des attributs
  2. Arrivé à l'endroit où il ne peut continuer que si la condition est remplie, t1 se met en attente en appelant la fonction :
     int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
  3. Quand t2 a effectué le travail pour que la condition soit remplie, il le notifie en utilisant la fonction :
     int pthread_cond_signal(pthread_cond_t *cond);
  4. ce qui débloque t1

Conditions

- Le mécanisme d’attente-notification met en jeu l’état interne de la condition ; pour éviter des accès concurrents à cet état interne, une synchronisation est nécessaire avec un mutex
- Autrement dit, les fonctions d’attente et de signalisation ne peuvent être lancés que dans une section critique
- Le thread qui appelle la fonction d’attente doit auparavant avoir verrouillé le mutex, le blocage du thread déverrouillant automatiquement le mutex. Quand le signal sera reçu, le mutex se re-vérrouillera automatiquement lors du réveil du thread. Le thread devra plus tard déverrouiller le mutex
- De même, pour appeler la méthode de signalisation, un thread doit auparavant avoir verrouillé le mutex. Après l’appel, il devra déverrouiller le mutex

Conditions

- Si plusieurs threads sont bloqués sur la même condition on peut utiliser la fonction :
  int pthread_cond_broadcast(pthread_cond_t *cond);
- Cette fonction débloque tous les threads alors que la fonction signal débloque aléatoirement un seul des threads bloqués
- Attention :
  - c'est une erreur d'appeler signal avant qu'un wait ait été appelé
  - si on ne verrouille/déverrouille pas correctement le mutex lors des appels à wait et signal le comportement sera faussé
    - par exemple, si on appelle wait sans avoir verrouillé le mutex le thread ne sera pas bloqué !
    - de même, si on oublie de déverrouiller après avoir appelé signal, l'autre thread restera bloqué !

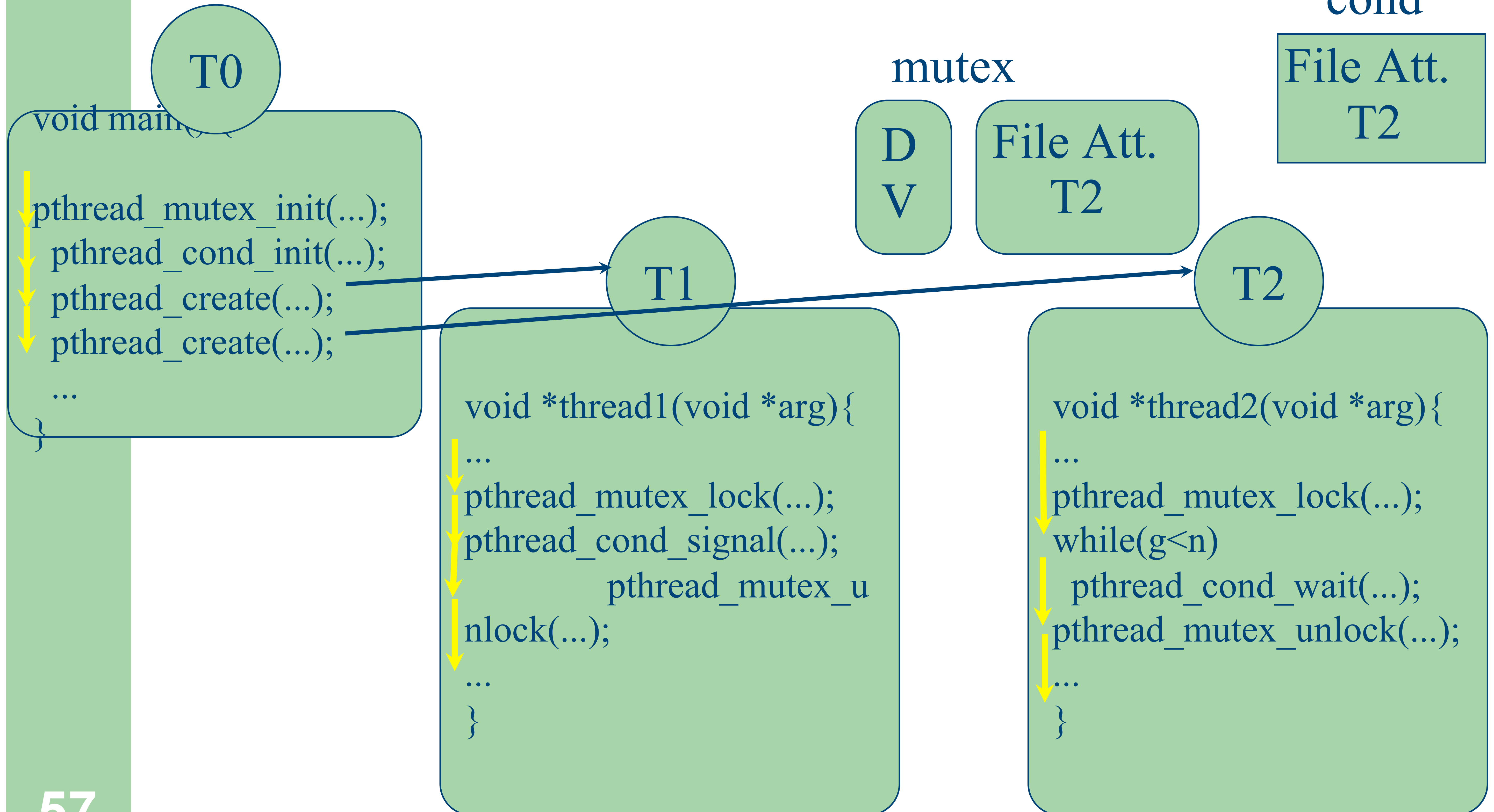
Conditions

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

long g = 0; /* variable globale */
pthread_mutex_t mutex; /* mutex pour protéger la variable globale et la condition */
pthread_cond_t cond; /* condition */

void *thread1(void *arg) {
     long n = (long)arg;
     long i;
     for (i = 0; i < n; i++) {
          pthread_mutex_lock(&mutex); /* section critique : on touche à g */
          g = i;
          pthread_mutex_unlock(&mutex); /* fin de section critique */
     }
     pthread_mutex_lock(&mutex); /* section critique : on touche à g et à cond */
     g = i;
     pthread_cond_signal(&cond); /* on dit à thread 2 qu'on a fini */
     pthread_mutex_unlock(&mutex); /* fin de section critique */
     pthread_exit(NULL);
}
```

Conditions

```
void *thread2(void *arg) {
     long n = (long)arg;
     pthread_mutex_lock(&mutex); /* section critique : on touche à g et à cond */
     while (g<n) {
          pthread_cond_wait(&cond, &mutex); /* on attend la condition */
     }
     printf("Le thread 1 a fini et g = %ld\n", g);
     pthread_mutex_unlock(&mutex); /* fin de section critique */
     pthread_mutex_destroy(&mutex); /* tout est fini, on nettoie */
     pthread_cond_destroy(&cond);
     pthread_exit(NULL);
}

int main (int argc, char *argv[]) {
     pthread_t th1, th2;
     long n = 10;
     pthread_mutex_init(&mutex, NULL); /* init. du mutex */
     pthread_cond_init(&cond, NULL); /* init. de la condition */
     pthread_create(&th1, NULL, thread1, (void *)n);
     pthread_create(&th2, NULL, thread2, (void *)n);
     pthread_exit(NULL);
}
```

Condition



Difficultés liées aux conditions

- Si un thread t1 doit attendre 2 signalisation de 2 autres threads (t2 et t3), ce serait faux de coder deux appels à la fonction wait
- En effet, les 2 signalisation peuvent arriver «presque en même temps» :
  - le thread t2 envoie un 1er signal qui débloque le 1er wait ; il relâche ensuite le mutex et permet ainsi ...
  - ... au thread t3 d'envoyer le 2ème signal avant que le 2ème wait ne soit exécuté
  - le thread t1 reste bloqué éternellement sur le 2ème wait (qui ne recevra jamais de signal)

Comment se débrouiller

- On compte le nombre de signalisation avec une variable qui est incrémentée dans la section critique qui contient le signal (pour être certain que la variable représente vraiment le nombre de signalisation) :

```
pthread_mutex_lock(&mutex);
nbSignalisation++;
if (nbSignalisation==2) pthread_cond_signal(&cond);
pthread_mutex_unlock(&mutex);
```

- Et on se met en attente dans une boucle (dans une section critique) :

```
 pthread_mutex_lock(&mutex);
 while (nbSignalisation<2) {
    pthread_cond_wait(&cond, &mutex);
 }
 pthread_mutex_unlock(&count_mutex);
```

- Si on reçoit 1 signal entre les 2 wait, nbSignalisation sera égal à 2 et on sortira de la boucle sans faire le 2ème wait()

Eviter la sur-synchronisation

- Comme la synchronisation a un coût non négligeable, il faut essayer de l'éviter quand c'est possible
- Par exemple, si un seul thread écrit une valeur de type int qui est lue par plusieurs autres threads, on peut se passer de synchronisation car les opérations de lecture/écriture de int sont toujours atomiques (processeurs 32 et 64 bits)

Résumé

- 2 solutions pour le parallélisme : processus et threads
- Processus : mémoire privée => communication entre processus
  - Signaux : communication simple d’événements
  - Tubes anonymes : communication d’information
  - Nombreux autres outils : tubes nommés, sémaphores, mémoire partagée, boites aux lettres, sockets...
- Threads : mémoire partagée => synchronisation entre threads
  - Mutex : exclusion mutuelle simple
  - Condition : attente et signalisation d’un évènement
- Choix :
  - protection : processus (exemple : Apache/PHP)
  - efficacité : threads (exemple : SGBD)

Remerciements

- Ce document est basé sur les sources suivantes
  - man pages Linux et MacOS X
  - cours de C. Teyssié en M1 MIAGE (cedric-teyssie.info)
  - cours de C. Drocourt en IUT à Amiens (drocourt.info)
  - cours de W. Holzmann, University of Lethbridge (www.cs.uleth.ca/~holzmann/C/system/)
  - tutoriel de Blaise Barney, Lawrence Livermore National Laboratory (computing.llnl.gov/tutorials/pthreads/)