INF2610
Noyau d’un système d’exploitation

Chapitre 3 - Threads (Fils d’exécution)

Sommaire

- Qu'est ce qu'un thread ?
- Avantages/inconvénients des threads
- Threads POSIX
  - Création d'un thread
  - Terminaison d'un thread
  - Annulation d'un thread
  - Attente de la fin d'un thread
  - Fonctions de nettoyage
  - Envoi de signaux à un thread
- Cas de Linux (appel système clone)

Qu’est-ce qu’un thread ?

- Les threads (ou fils d’exécution) sont un concept des systèmes d’exploitation modernes qui permet de lancer en concurrence plusieurs parties d’un même processus.

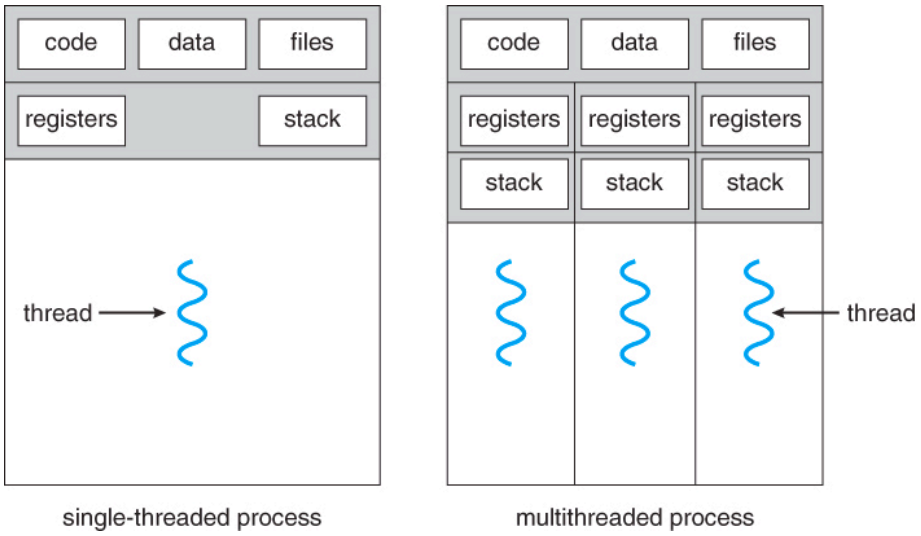
- Lors de sa création, un processus contient un seul thread principal qui exécute la fonction main. Ce thread principal peut créer d’autres threads qui, à leur tour, peuvent en créer d’autres (mais pas de relation hiérarchique).

- Tous les threads ainsi créés sont rattachés au processus et partagent les ressources du processus (espace d’adressage, table des descripteurs de fichiers, table des gestionnaires de signaux, etc.).

Qu’est-ce qu’un thread ? (2)

Chaque thread a :

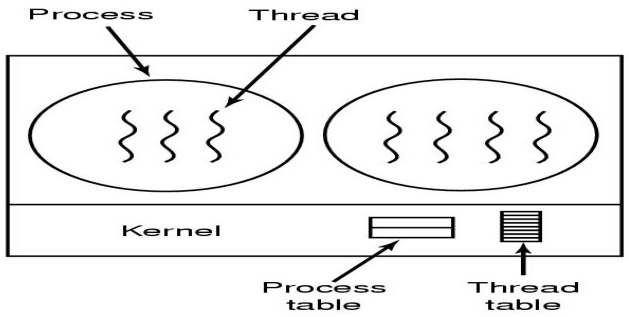
- un tid identifiant unique (thread identifier),
- une pile d’exécution (stack),
- des registres,
- un compteur ordinal,
- un état,
- une priorité,
- etc.



Qu’est-ce qu’un thread ? (3)

- Tous les systèmes d’exploitation de la famille Unix ainsi que Windows supportent les threads. Ils ordonnancent les threads et non les processus (allocation de processeurs aux threads).

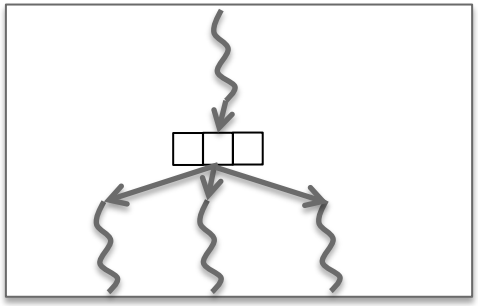
- Les threads ordonnancés par le noyau sont qualifiés de threads noyau.

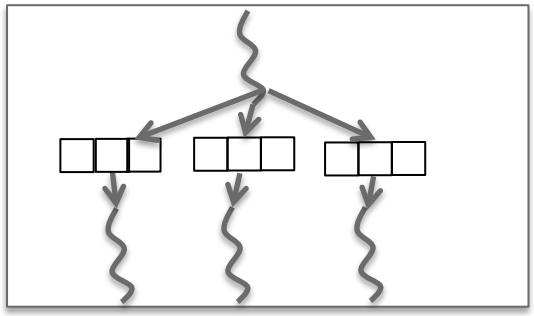


- Les threads créés et ordonnancés par une librairie au niveau utilisateur sont qualifiés de threads utilisateur. La librairie GNU pth est un exemple d’une telle librairie.

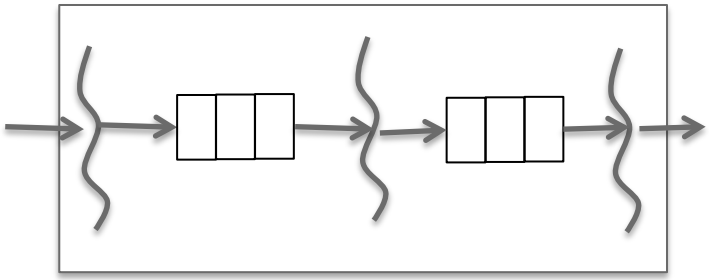
Qu’est-ce qu’un thread ? (4)

Quelques modèles de conception basés sur les threads (patrons de conception)





Répartition des tâches (avec création statique (thread pool) ou encore dynamique des threads)



Étapes/transformations successives (pipeline)

Avantages des threads

- **Réactivité :** si un thread d'un processus est bloqué, ses autres threads peuvent continuer à s'exécuter. Les threads d'un même processus s'exécutent en pseudo-parallèle ou en parallèle (dans un système multiprocesseur).

- **Économie d'espace** : le partage des ressources (espace d'adressage, fichiers, etc.) économise l'espace mémoire mais aussi améliore les performances. Les threads peuvent communiquer sans passer par des appels système.

- **Économie de temps :**
  - Il est plus rapide de créer/terminer un thread que de créer/terminer un processus.
  - Le changement de contexte entre deux threads d'un même processus est plus rapide que le changement de contexte entre deux processus.

Inconvénients des threads

- **Conditions de concurrence :** les accès concurrents par des threads aux données partagées (en lecture et écriture) peut créer des incohérences. Il est donc nécessaire aux programmeurs de bien contrôler les accès aux données partagées ➔ utilisation de mécanismes de synchronisation.

- **Risque d'épuisement et gaspillage des ressources des processus :** la création d'un nombre important de threads au sein d'un même processus va épuiser les ressources du processus et affecter ses performances. Certaines ressources allouées à un thread ne sont pas libérées à la fin du thread (fichiers, allocations dynamique, etc.).

- **Cohérence des caches** des processeurs (assurée par le système) : pourrait avoir un effet négatif sur les performances des processus multithreads.

Threads POSIX (librairie pthread)

- Les threads POSIX peuvent être utilisés sous Windows, Linux, Solaris, macOS, etc.

- La librairie pthread permet de créer, terminer, se mettre en attente de la fin, synchroniser, forcer la terminaison des threads, etc.

Création d'un thread - pthread_create

int pthread_create(pthread_t* tid, const pthread_attr_t* attr, void* func, void* arg)

où :

- tid sert à récupérer l'identifiant du thread créé,
- attr spécifie les attributs de création du thread (NULL pour les attributs par défaut),
- func est le nom de la fonction exécutée par le thread, et
- arg est l'argument à passer à la fonction func.

Le prototype de la fonction func est : **void* func (void* arg);**

- Cet appel retourne la valeur 0 en cas de succès et une valeur différente de 0 en cas d'échec.

La fonction exécutée par un thread doit obligatoirement avoir un **seul paramètre**!
Si vous voulez que le thread exécute une fonction ayant plusieurs paramètres, vous devez les regrouper dans une **struct** et créer une fonction *wrapper* qui appellera la fonction ayant plusieurs paramètres avec les attributs de la **struct** (voir l'exemple dans codes/ ProcessusEtThreads/PthreadWrapper).

Terminaison d’un thread

- La fonction **pthread_exit()** termine le thread appelant :

  void pthread_exit(void* pvaleur)

  où pvaleur est un pointeur. Ce pointeur pourrait être récupéré par un autre thread via la fonction **pthread_join**.

- Un thread d’un processus peut forcer la terminaison d’un autre thread du processus en appelant la fonction **pthread_cancel()** :

  int pthread_cancel(pthread_t tid)

  Cette fonction retourne 0, en cas de succès et une autre valeur, en cas d’erreur. L’état de terminaison PTHREAD_CANCELED pourrait être récupéré par un autre thread via la fonction **pthread_join.**

Attendre la fin d'un thread - pthread_join

- La fonction **pthread_join()** met le thread appelant en attente de la fin d'un thread :

int pthread_join(pthread_t tid, void pstatus)**

où :

- tid est l'identifiant du thread à attendre (qu'il se termine),
- pstatus contient l'adresse de la variable où récupérer les informations de terminaison du thread tid. **Le double pointeur pstatus est égale à** :
  - **&pval**, si le thread tid s'est terminé avec **pthread_exit(pval)**, et
  - **PTHREAD_CANCELED**, si le thread tid s'est terminé anormalement (suite à un appel à **pthread_cancel**).

- Cet appel retourne 0 en cas de succès et une valeur non nulle en cas d'échec.

Exemple 1 : Pthread Race conditions

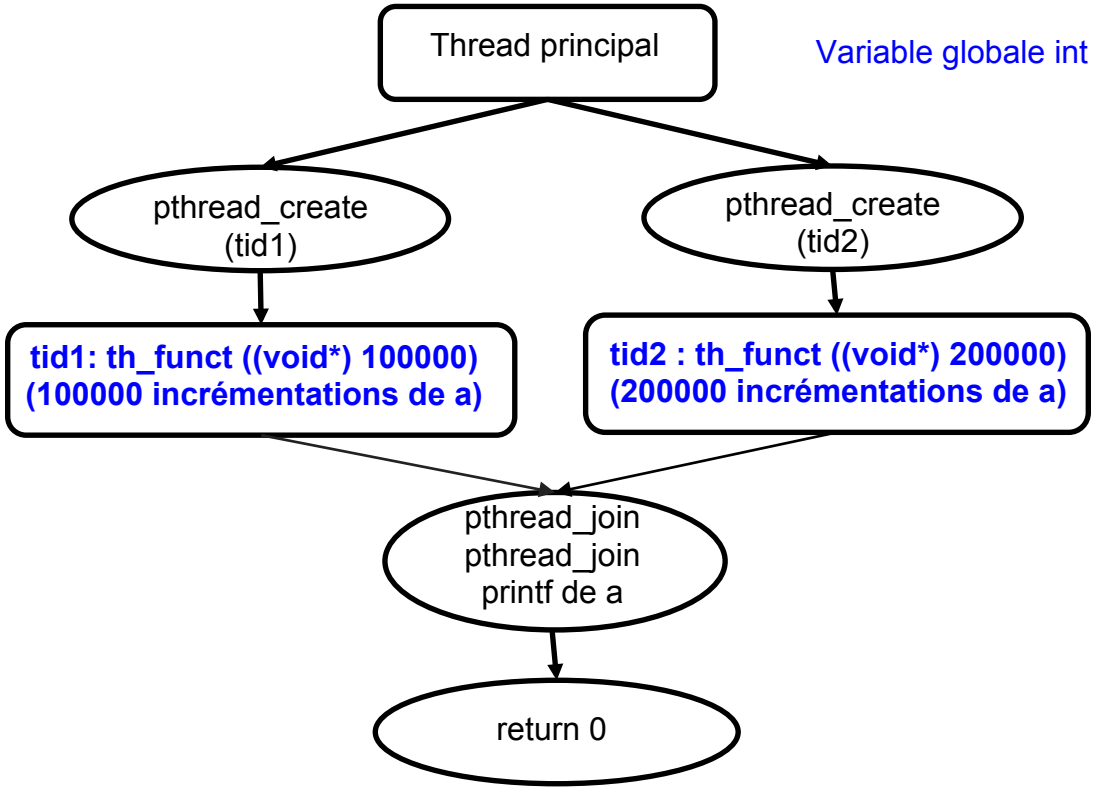
$ ./run.sh ProcessusEtThreads/ PthreadRaceCond

- Le thread principal du programme pthreadRaceCond.c crée 2 threads puis se met en attente de la fin de ses threads. Il affiche ensuite la valeur de la variable globale **a** et se termine.

- Chaque thread créé va exécuter la fonction th_funct. Le paramètre de cette fonction indique le nombre d'incrémentations, de la variable partagée **a**, qu'elle a à réaliser.

➔ **Condition de concurrence (race condition) car les threads accèdent en concurrence (en lecture et en écriture) à la variable a.**

Exemple 1 : Pthread race conditions (2)



Variable globale int a=0

Accès concurrents à la variable a par les threads tid1 et tid2

Exemple 1 : Pthread race conditions (3)

```
// pthreadRaceCond.c ….
long a=0;
void* th_funct(void * p)
{   printf("thread exécutant thread_funct(%ld)\n”, (long)p);
    for(long i=0;i<(long)p;i++)   a++;
    pthread_exit(NULL);
}
int main( )
{    pthread_t tid1, tid2;
    if ( pthread_create(&tid1, NULL, th_funct, (void*)100000) != 0) return -1;
    if ( pthread_create(&tid2, NULL, th_funct, (void*)200000) != 0) return -1;
    pthread_join(tid1,NULL);
    pthread_join(tid2,NULL);
    printf("thread principal se termine avec a =%ld\n”, getpid(),a);
    return 0;
}
```

Exemple 1 : Pthread race conditions (4)

```
$ ./Run.sh ProcessusEtThreads/PthreadRaceCond
gcc -o a.out -pthread  pthreadRaceCond.c
thread exécutant th_funct(100000)
thread exécutant th_funct(200000)
thread principal se termine avec a =270281
```

```
$ ./Run.sh ProcessusEtThreads/PthreadRaceCond
gcc -o a.out -pthread  pthreadRaceCond.c
thread exécutant th_funct(100000)
thread exécutant th_funct(200000)
thread principal se termine avec a =243828
```

Valeur finale de a n'est pas celle attendue et varie d'une exécution à une autre.

Comment éviter les conditions de concurrence (race conditions) ?

Exemple 2
Fin normale/anormale

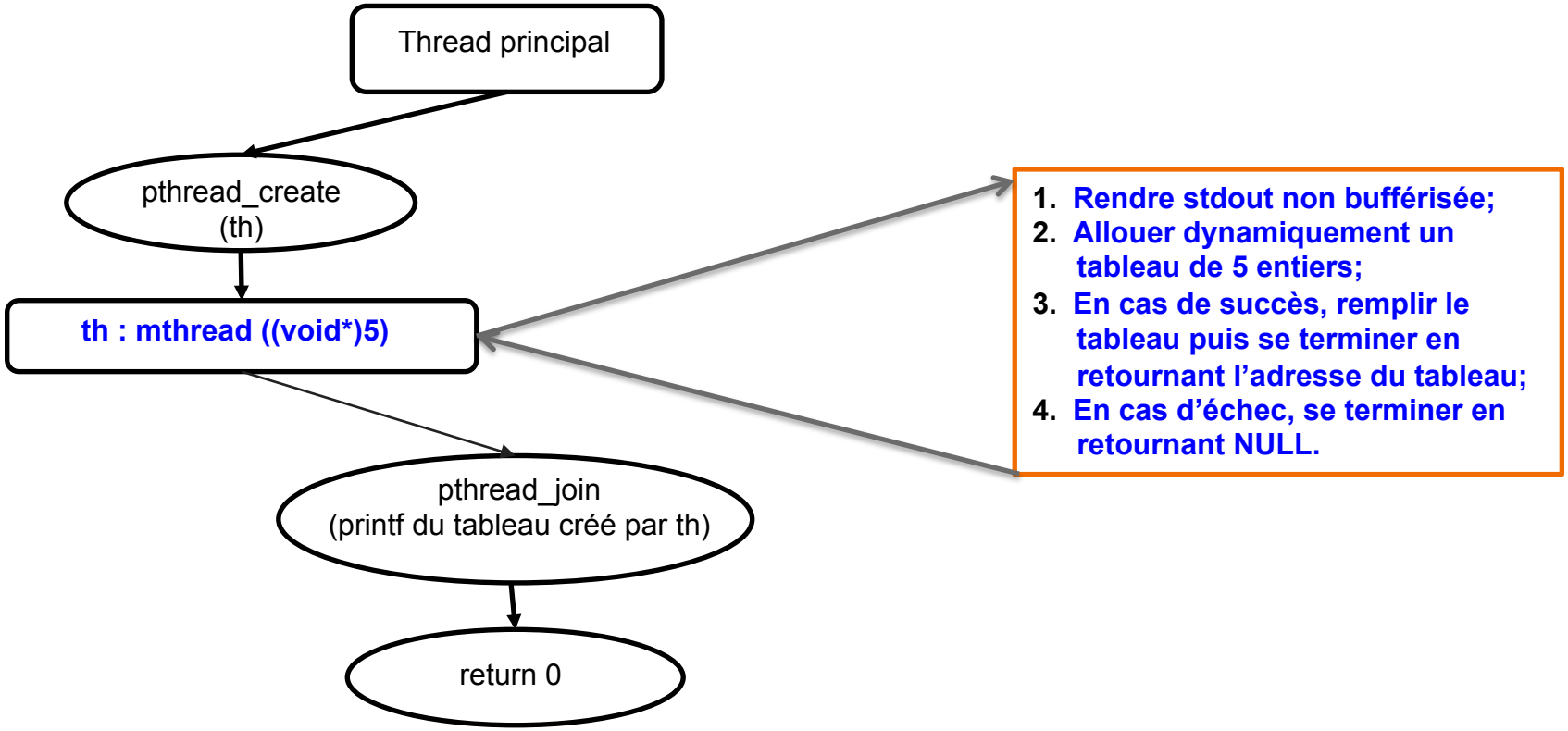
./run.sh ProcessusEtThreads/
PthreadCancel

- Le thread principal du programme pthreadCancel.c crée un seul thread puis se met en attente de la fin de son thread.

- Le thread créé va exécuter la fonction mthread qui commence par rendre la sortie standard non bufférisée. Ensuite, elle alloue dynamiquement un tableau de 5 entiers. En cas de succès de l'allocation dynamique, elle remplit le tableau puis se termine en renvoyant l'adresse du tableau créé.
- En cas d'échec de l'allocation dynamique, le thread se termine en renvoyant NULL.

- Après la fin de son thread, le thread principal récupère l'état de terminaison du thread, affiche éventuellement le contenu du tableau créé par son thread puis se termine.

Exemple 2 : Fin normale/anormale (2)



Exemple 2 : Fin normale/anormale (3)

```
// pthreadCancel.c …..
void *mthread(void *taille)
{   setvbuf(stdout, (char *) NULL, _IONBF, 0);
    int*tab = (int*) malloc ( (long)taille*sizeof(int));
    if (tab!=NULL)
    {   for (long i=0; i<(long)taille; i++) {
            tab[i]=i;
            printf("%ld sur %ld ",i,(long)taille-1);
           usleep(50000); // pour ralentir le traitement attendre 5ms
        }
        printf("fin \n");
    }
    pthread_exit ((void*) tab);
}
```

Exemple 2 : Fin normale/anormale (4)

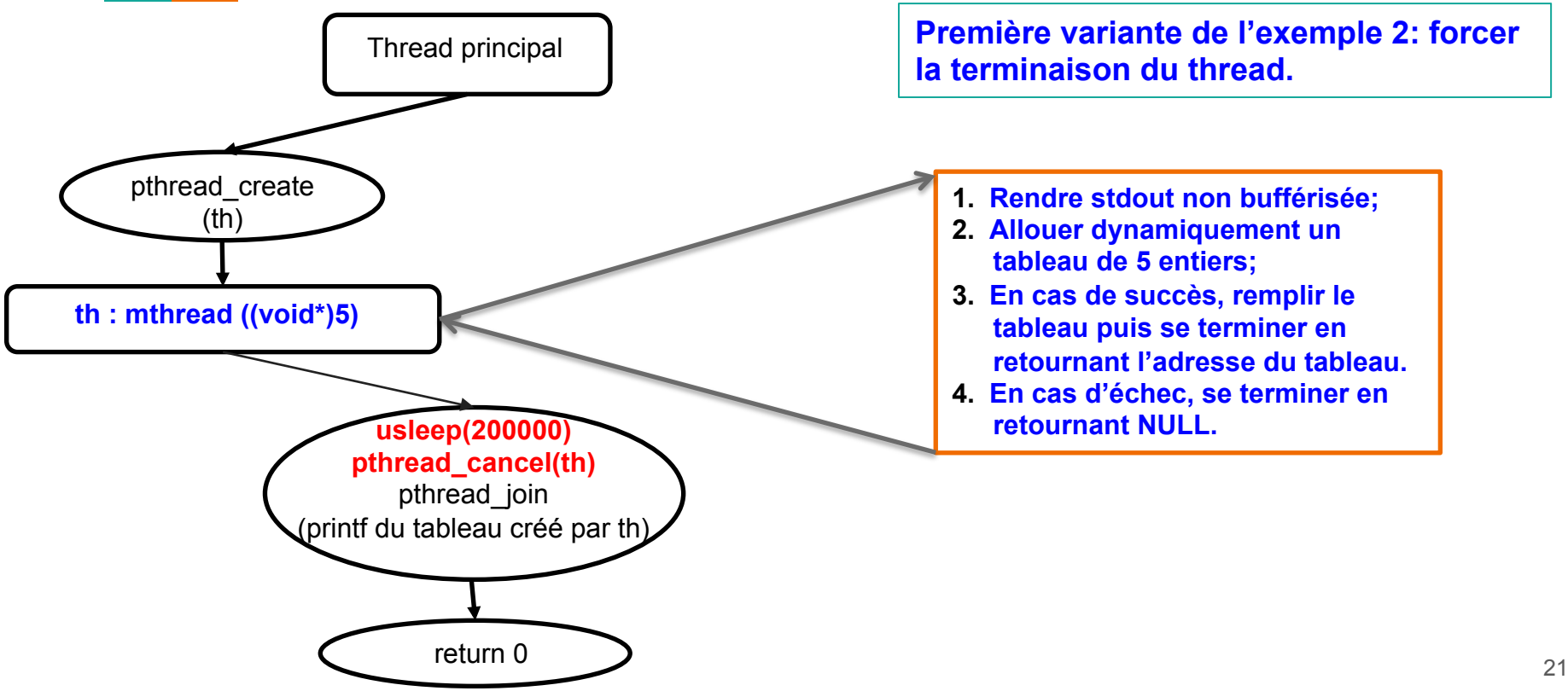
```
int main() {
   pthread_t th;
   int* pstatus;
   if ( pthread_create(&th,NULL,mthread,(void*)5) )   return 1;
   pthread_join(th, (void**) &pstatus);
   if (pstatus == PTHREAD_CANCELED) {    printf("Terminaison forcée \n"); }
   else   if (pstatus== NULL) {    printf("\n Terminaison normale avec NULL\n");}
          else {   printf("Terminaison normale avec %p.\n", pstatus);
                   for(int i=0; i<5;i++)
                   {     printf("tab[%d]=%d ", i, pstatus[i]);  }
                   printf("\n");
                   free(pstatus);
                }
   return 0;
}
```

```
$ ./run.sh ProcessusEtThreads/PthreadCancel
gcc -pthread -o a.out pthreadCancel.c
0 sur 4 1 sur 4 2 sur 4 3 sur 4 4 sur 4 fin
Terminaison normale avec 0x7f4718000b20.
tab[0]=0 tab[1]=1 tab[2]=2 tab[3]=3 tab[4]=4
```

Remarque : Le thread th s'est terminé et l'espace alloué dynamiquement au tableau n'a pas été libéré.

Que se passera-t-il en cas d'un arrêt forcé du thread ?

Exemple 2 : Fin normale/anormale – Variante 1



Exemple 2 : Fin normale/anormale – Variante 1 (2)

```
int main() {
   pthread_t th;
   int* pstatus;
   if ( pthread_create(&th,NULL,mthread,(void*)5) )   return 1;
    usleep(200000);  pthread_cancel(th);
   pthread_join(th, (void**) &pstatus);
   if ( pstatus == PTHREAD_CANCELED ) {    printf("\nTerminaison forcee \n"); }
   else   if ( pstatus== NULL) {    printf("Terminaison normale avec NULL\n");}
          else {   printf("Terminaison normale avec %p.\n", pstatus);
                   for(int i=0; i<5;i++)  { printf("tab[%d]=%d ", i, pstatus[i]);}
                    printf("\n");
                    free(pstatus);
                  }
   return 0;
}
```

```
$ ./run.sh ProcessusEtThreads/PthreadCancel
gcc -pthread -o a.out pthreadCancel.c
0 sur 4 1 sur 4 2 sur 4 3 sur 4
Terminaison forcée
```

La terminaison du thread th a été forcée et l'espace alloué dynamiquement au tableau n'a pas été libéré.
➔ Ajouter une fonction de nettoyage au niveau du thread en prévision d'un arrêt forcé.

Fonctions de nettoyage

- Chaque thread dispose d'une pile de nettoyage dans laquelle le thread peut empiler les fonctions à exécuter lors de sa terminaison :

void pthread_cleanup_push(void func, void* args)

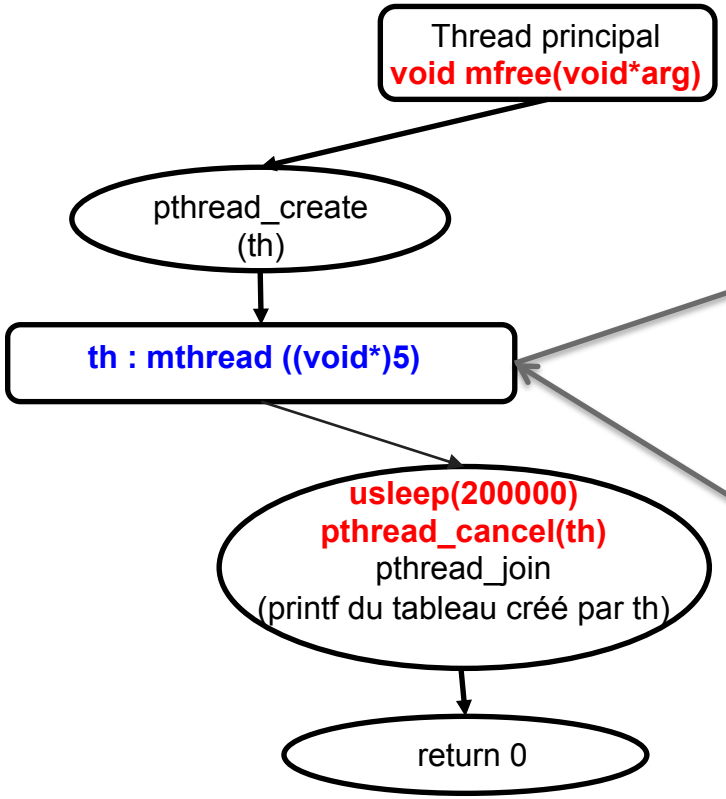
- ajoute la fonction func dans la pile de nettoyage. La signature de la fonction func est : **void func (void* args)**.

void pthread_cleanup_pop(int execute)

- dépile la fonction en tête de la pile de nettoyage et exécute celle-ci si le paramètre **execute** n'est pas 0.

- Lors de la terminaison d'un thread, **les fonctions figurant dans sa pile de nettoyage sont dépilées et exécutées séquentiellement**.

Exemple 2 : Fin normale/anormale – Variante 2



Deuxième variante de l'exemple 2: forcer la terminaison du thread avec ajout d'une fonction de nettoyage.

1. **Rendre stdout non bufférisée;**
2. **Allouer dynamiquement un tableau de 5 entiers;**
3. **En cas de succès,**
   3.1 Empiler la fonction mfree(tab) dans la pile de nettoyage du thread;
   3.2 remplir le tableau;
   3.3 dépiler (sans l'exécuter) la fonction mfree(tab) de la pile de nettoyage du thread;
   3.4 se terminer en retournant l'adresse du tableau;
4. **En cas d'échec, se terminer en retournant NULL.**

Exemple 2 : Fin normale/anormale – Variante 2 (2)

```
// pthreadCancel.c …..
void mfree (void *arg)  {  printf("free\n"); free(arg); }
void *mthread(void *taille)
{    setvbuf(stdout, (char *) NULL, _IONBF, 0);
   int*tab = (int*) malloc ( (long)taille*sizeof(int));
   if (tab!=NULL)
   {  pthread_cleanup_push(mfree, (void*) tab);
      for (long i=0; i<(long)taille; i++) {   tab[i]=i;
         printf("%ld sur %ld ",i,(long)taille-1);
         usleep(50000); // pour ralentir le trait. attendre 5ms
      }
      printf("fin \n");
      pthread_cleanup_pop(0);
    }
   pthread_exit ((void*) tab);
}
```

```
$ ./run.sh ProcessusEtThreads/PthreadCancel
gcc -pthread -o a.out pthreadCancel.c
0 sur 4 1 sur 4 2 sur 4 3 sur 4 free

Terminaison forcée
```

Qu'arrivera-t-il, si la terminaison forcée se produit entre pthread_cleanup_pop(0) et pthread_exit((void*) tab) ?

➔ différer l'annulation

Différer la terminaison forcée

- Chaque thread a un bit d'annulation qui vaut PTHREAD_CANCEL_ENABLE ou PTHREAD_CANCEL_DISABLE.

- **Par défaut**, le bit d'annulation est fixé à PTHREAD_CANCEL_ENABLE. Ce qui signifie que la terminaison du thread peut être forcée par la fonction **pthread_cancel**.

- Un thread peut bloquer (différer) la terminaison forcée par **pthread_cancel** en modifiant son bit d'annulation à **PTHREAD_CANCEL_DISABLE**.

- La fonction qui permet de modifier ce bit est :

  int pthread_setcancelstate(int state, int *oldstate);

Exemple 2 : Fin normale/anormale – Variante 3

```
// pthreadCancel.c …..
void mfree (void *arg)  {  printf("free\n"); free(arg); }
void *mthread(void *taille)
{    setvbuf(stdout, (char *) NULL, _IONBF, 0);
   int*tab = (int*) malloc ( (long)taille*sizeof(int));
   if (tab!=NULL)
   {  pthread_cleanup_push(mfree, (void*) tab);
      for (long i=0; i<(long)taille; i++) {   tab[i]=i;
      printf("%ld sur %ld ",i,(long)taille-1);
      usleep(50000); // pour ralentir le traitement attendre 5ms
    }
     printf("fin \n");    pthread_setcancelstate(PTHREAD_CANCEL_DISABLE,NULL);
     pthread_cleanup_pop(0);   sleep(5);  // retarder sa terminaison de 5 secondes
   }
      pthread_exit ((void*) tab);
}
```

Troisième variante de l'exemple 2: empêcher l'annulation du thread, s'il parvient à remplir le tableau.

Exemple 2 : Fin normale/anormale – Variante 3 (2)

```
int main() {
   pthread_t th;
   int* pstatus;
   if ( pthread_create(&th,NULL,mthread,(void*)5) )   return 1;
    usleep(280000);  pthread_cancel(th);
   pthread_join(th, (void**) &pstatus);
   if ( pstatus == PTHREAD_CANCELED ) {    printf("\nTerminaison forcee \n"); }
   else   if ( pstatus== NULL) {    printf("Terminaison normale avec NULL\n");}
          else {   printf("Terminaison normale avec %p.\n", pstatus);
                   for(int i=0; i<5;i++)  { printf("tab[%d]=%d ", i, pstatus[i]);}
                   printf("\n");
                   free(pstatus);
                }
   return 0;
}
```

```
$ ./run.sh ProcessusEtThreads/PthreadCancel
gcc -pthread -o a.out pthreadCancel.c
0 sur 4 1 sur 4 2 sur 4 3 sur 4 4 sur 4 fin
Terminaison normale avec 0x7fc524000b20.
tab[0]=0 tab[1]=1 tab[2]=2 tab[3]=3 tab[4]=4
```

Exemple 2 : Fin normale/anormale – Variante 3 (3)

```
int main() {
   pthread_t th;
   int* pstatus;
   if ( pthread_create(&th,NULL,mthread,(void*)5) )   return 1;
    usleep(200000);  pthread_cancel(th);
   pthread_join(th, (void**) &pstatus);
   if ( pstatus == PTHREAD_CANCELED ) {    printf("\nTerminaison forcee \n"); }
   else   if ( pstatus== NULL ) {    printf("Terminaison normale avec NULL\n");}
          else {   printf("Terminaison normale avec %p.\n", pstatus);
                   for(int i=0; i<5;i++)  { printf("tab[%d]=%d ", i, pstatus[i]);}
                   printf("\n");
                  free(pstatus);
                }
   return 0;
}
```

```
$ ./run.sh ProcessusEtThreads/PthreadCancel
gcc -pthread -o a.out pthreadCancel.c
0 sur 4 1 sur 4 2 sur 4 3 sur 4 free

 Terminaison forcée
```

Envoi de signaux - pthread_kill / kill

- La fonction **pthread_kill()** permet d'envoyer un signal à un thread :

int pthread_kill(pthread_t tid, int sig)

- Elle retourne 0 en cas de succès, une valeur non nulle en cas d'erreur.
- Chaque thread a un **masque de signaux privé** (man **pthread_sigmask**).
- La fonction **kill()** envoie un signal à un processus. **Ce signal peut être traité par n'importe quel thread du processus qui n'a pas bloqué ce signal.**

Exemple 3
Envoi d'un signal à un thread

```
./run.sh ProcessusEtThreads/
PthreadKill
```

- Le thread principal crée un thread pour exécuter la fonction **action**.
- Cette fonction commence par remplacer le gestionnaire par défaut du signal SIGINT par la fonction **sigIntHandle**. Elle se met en attente d'un signal.
- Le thread principal attend 1 seconde, envoie le signal SIGINT au thread qu'il a créé, se met attente de la fin du thread puis se termine.

Exemple 3 - Envoi d'un signal (2)

```
// threadkill.c
….
void sigIntHandler (int signum) {
    printf( "signal %d recu  par %ld \n", signum, syscall(SYS_gettid));
}

void * action (void * unused) {
  signal(SIGINT, sigIntHandler);
  printf("Thread %ld se met en pause \n", syscall(SYS_gettid));
  pause();
  printf("Thread %ld se termine \n", syscall(SYS_gettid));
  pthread_exit(NULL);
}
```

Exemple 3 - Envoi d'un signal (3)

```
// threadkill.c …
int main(){
    void *pstatus; pthread_t tid;
    pthread_create (&tid, NULL, action, NULL);
    sleep(1);
    printf("Thread %ld envoie SIGINT (%d) au thread  \n", syscall(SYS_gettid), SIGINT);
    pthread_kill(tid, SIGINT);
    printf("Thread %ld attend la fin du thread \n",  syscall(SYS_gettid) );
    pthread_join(tid,&pstatus);
    if (pstatus==NULL || pstatus!=PTHREAD_CANCELED)
    {       printf(" Fin normale du thread créé \n");    }
   else    {    printf(" Fin forcee du thread créé \n"); }
    return 0;
}
```

```
$ ./run.sh ProcessusEtThreads/PthreadKill
gcc -pthread -o a.out threadkill.c
Thread 14 se met en pause
Thread 13 envoie SIGINT de num 2 au thread
Thread 13 attend la fin du thread
signal 2 recu  par 14
Thread 14 se termine
 Terminaison normale du thread créé
```

Exemple 3 - Envoi d'un signal – Variante 2

```
// threadkill.c …
int main(){
    void *pstatus; pthread_t tid;
    pthread_create (&tid, NULL, action, NULL);
   // sleep(1);
    printf("Thread %ld envoie SIGINT (%d) au thread  \n", syscall(SYS_gettid), SIGINT);
    pthread_kill(tid, SIGINT);
    printf("Thread %ld attend la fin du thread \n",  syscall(SYS_gettid) );
    pthread_join(tid,&pstatus);
    if (pstatus==NULL || pstatus!=PTHREAD_CANCELED)
    { printf(" Fin normale du thread créé \n"); }
    else { printf(" Fin forcee du thread créé \n"); }
    return 0;
}
```

```
$ ./run.sh ProcessusEtThreads/PthreadKill
gcc -pthread -o a.out threadkill.c
Thread 13 envoie SIGINT (2) au thread
Thread 13 attend la fin du thread
signal 2 recu  par 14
Thread 14 se met en pause
```

Signal reçu avant de se mettre en pause

Exemple 3 - Envoi d'un signal – Variante 3

```
$ ./run.sh ProcessusEtThreads/PthreadKill
gcc -pthread -o a.out threadkill.c
Thread 14 se met en pause
Thread 13 envoie SIGINT (2) au thread
```

```
// threadkill.c
….
void sigIntHandler (int signum) {
    printf( "signal %d recu  par %ld \n", signum, syscall(SYS_gettid));
 }
void * action (void * unused) {
 // signal(SIGINT, sigIntHandler);
  printf("Thread %ld se met en pause \n", syscall(SYS_gettid));
  pause();
  printf("Thread %ld se termine \n", syscall(SYS_gettid));
  pthread_exit(NULL);
}
```

Cas de Linux

- Linux ne fait pas de distinction entre un **processus** et un **thread**. Ils sont appelés communément **tâche** et représentés par la structure de données **task_struct.**

- Une tâche Linux peut être créée via les appels systèmes **fork**, **vfork** mais aussi via l'appel système **clone**.

- La fonction vfork est similaire à fork, à l'exception que le processus créateur reste bloqué jusqu'à ce que le processus fils se termine ou remplace son code (ex: avec execl())

```
$ man fork et man vfork
```

Cas de Linux (2)

- L'appel système **clone** (propre à Linux) permet de créer une tâche en spécifiant les **ressources à partager entre la tâche créatrice et la tâche créée** (espace d'adressage, table des descripteurs de fichiers, table des gestionnaires de signaux, etc.)

- Il permet ainsi de créer des processus et des threads au sens de POSIX.

- Il permet également de créer des tâches qui partagent plus de ressources que les processus mais moins de ressources que les threads.

- Il est possible aussi de créer des threads utilisateur en utilisant, par exemple, la librairie pth (GNU Portable Threads). https://www.gnu.org/software/pth/pth-manual.html

Threads niveau noyau
VS
Threads niveau utilisateur

Exemple 4
Librairie pth

./run.sh ProcessusEtThreads/
UserThreads

- Le thread principal commence par appeler la fonction pth_init, de la librairie pth, pour créer, notamment, l'ordonnanceur des threads utilisateur.

- Il va ensuite créer deux threads pth qui exécuteront la fonction count avec comme paramètre la variable counter. Cette fonction va incrémenter 1 milliard de fois la variable counter puis se termine.

- Il attend la fin de ses deux threads pth, affiche la valeur counter puis se termine.

Exemple 4 – Librairie pth

```
//uthreads.c
unsigned long long counter = 0;
#define MAX 1000000000
void* count(void *arg) {
    unsigned long long*receivedValue = (unsigned long long*) arg;
    unsigned long long i;
    for (i = 0; i < MAX; i++) {    *receivedValue += 1; }
    pth_exit (NULL);
}
```

Exemple 4 – Librairie pth (2)

```
//uthreads.c

int main( ) {
    pth_t t1, t2;
    pth_init();
    t1=pth_spawn(PTH_ATTR_DEFAULT,count, (void*)&counter);
    t2=pth_spawn(PTH_ATTR_DEFAULT,count, (void*)&counter);
    pth_join(t1,NULL);
    pth_join(t2,NULL);
    printf("pid= %d counter=%llu\n", getpid(), counter);

    return 0;
}
```

```
$ ./run.sh ProcessusEtThreads/UserThreads
cc `pth-config --cflags` -c uthreads.c
cc `pth-config --ldflags` -o a.out uthreads.o `pth-config --libs`
pid= 18 counter=2000000000
```

```
$ gcc uthreads.c  -o uthreads –lpth
$ strace -e trace=clone ./uthreads
pid= 2592 counter=2000000000
+++ exited with 0 +++
```

Exemple 5
Librairie pthread

./run.sh ProcessusEtThreads/NoyThreads

- Ce programme réalise le même traitement que le programme précédent en utilisant la librairie pthread (au lieu de la librairie pth).

Exemple 5 - Librairie pthread

```
//pthreads.c
unsigned long long counter = 0;
#define MAX 1000000000
void* count(void *arg) {
    unsigned long long*receivedValue = (unsigned long long*) arg;
    unsigned long long i;
    for (i = 0; i < MAX; i++) {    *receivedValue += 1; }
    pthread_exit (NULL);
}
```

Exemple 5 - Librairie pthread

```
//pthreads.c
int main( ) {
    pthread_t t1, t2;

    pthread_create(&t1,NULL,count, (void*)&counter);
    pthread_create(&t2,NULL,count, (void*)&counter);

    pthread_join(t1,NULL);
    pthread_join(t2,NULL);

    printf("pid= %d counter=%llu\n", getpid(), counter);

    return 0;
}
```

Exemple 5 - Librairie pthread

```
$ strace -e trace=clone ./pthreads
clone(child_stack=0x7f1678fb1fb0, flags=CLONE_VM|CLONE_FS|CLONE_FILES|
CLONE_SIGHAND|CLONE_THREAD|CLONE_SYSVSEM|CLONE_SETTLS|
CLONE_PARENT_SETTID|CLONE_CHILD_CLEARTID, parent_tid=[11772],
tls=0x7f1678fb2700, child_tidptr=0x7f1678fb29d0) = 11772
clone(child_stack=0x7f16787b0fb0, flags=CLONE_VM|CLONE_FS|CLONE_FILES|
CLONE_SIGHAND|CLONE_THREAD|CLONE_SYSVSEM|CLONE_SETTLS|
CLONE_PARENT_SETTID|CLONE_CHILD_CLEARTID, parent_tid=[11773],
tls=0x7f16787b1700, child_tidptr=0x7f16787b19d0) = 11773
pid= 11771 counter=1030616071
+++ exited with 0 +++
```

Exercice 1

- Dans les exemples 4 et 5, quel serait l'effet de l' ajout des instructions suivantes :

  printf("going to pause when *receivedValue=%lld\n", *receivedValue);
  pause();

  dans la fonction **count() à la sortie de la boucle for** ?

- Quelle serait la valeur de counter afficher à la fin du programme, s'il créait deux processus fils au lieu de deux threads ?

Exercice 2 (QCM)

Considérez le code suivant (sans les directives d'inclusion) :

```
int a=0;
void *increment(void *) { a++; return NULL;}
int main ()
 { pthread_t th[3];
   int i;
   for(i=0; i <3; i++)
         pthread_create(&th[i], NULL, increment, NULL);
   for(i=0; i <3; i++)
         pthread_join(th[i], NULL);
   printf(" a = %d\n", a);
   return 0;
}
```

Le programme est-il déterministe ?

a) Oui, car il affichera toujours la valeur 3.
b) Non, car il affichera 1, 2 ou 3.
c) Non, car il affichera 0, 1, 2 ou 3.
d) Oui, car il affichera toujours 0.
e) aucune de ces réponses.

Exercice 3 (QCM)

Considérez le code suivant (sans les directives d'inclusion) :

```
int a=0;
void *increment(void *) { a++; return NULL;}
int main ()
 { pthread_t th[3]; int i;
   for(i=0; i <3; i++)
         pthread_create(&th[i], NULL, increment, NULL);
   printf(" a = %d\n", a);
   return 0;
}
```

Le programme est-il déterministe ?

a) Oui, car il affichera toujours la valeur 3.
b) Non, car il affichera 1, 2 ou 3.
c) Non, car il affichera 0, 1, 2 ou 3.
d) Oui, car il affichera toujours 0.
e) aucune de ces réponses.

Exercice 4 (QCM)

Si un thread d'un processus exécute cette instruction « dup2(fd=open("*fich1*",O_WRONLY),1); » alors

a) la sortie standard du processus devient le fichier *fich1.*
b) l'écriture dans *fich1* par un thread du processus est redirigée vers l'écran.
c) l'écriture dans *fich1* par un thread du processus peut être réalisé en utilisant le descripteur fd ou 1.
d) tous les threads du processus n'ont plus accès à l'écran via le descripteur 1.
e) aucune de ces réponses.

Exercice 5 (QCM)

On veut faire communiquer deux threads *utilisateur* d'un même processus via un tube anonyme (*pipe*). À quel niveau doit-on créer le tube anonyme ?

Le tube anonyme doit être créé :

a) avant la création du premier thread.

b) après la création du premier thread et avant la création du second thread.

c) après la création du second thread.

d) dans chacun des deux threads.

e) aucune de ces réponses car les threads ne pourront pas communiquer en utilisant les appels système « *read* » et « *write* ».

Exercice 6 (QCM)

On veut faire communiquer deux threads utilisateur d'un même processus via un tube nommé. À quel niveau doit-on ouvrir le tube nommé ? Le tube nommé doit être ouvert :

a) avant la création du premier thread.
b) dans le premier thread.
c) dans le second thread.
d) dans chacun des deux threads (une en lecture et une autre en écriture).
e) aucune de ces réponses car les threads ne pourront pas ouvrir le tube.

Lecture suggérée

- **Chapitre 2.2 : Threads**

Introduction aux systèmes d'exploitation - Cours et exercices en GNU/Linux, Hanifa Boucheneb & Juan-Manuel Torres-Moreno, 216 pages, édition ellypses, 2019, ISBN : 9782340029651.