IV - Concurrence

Communications en parallèle

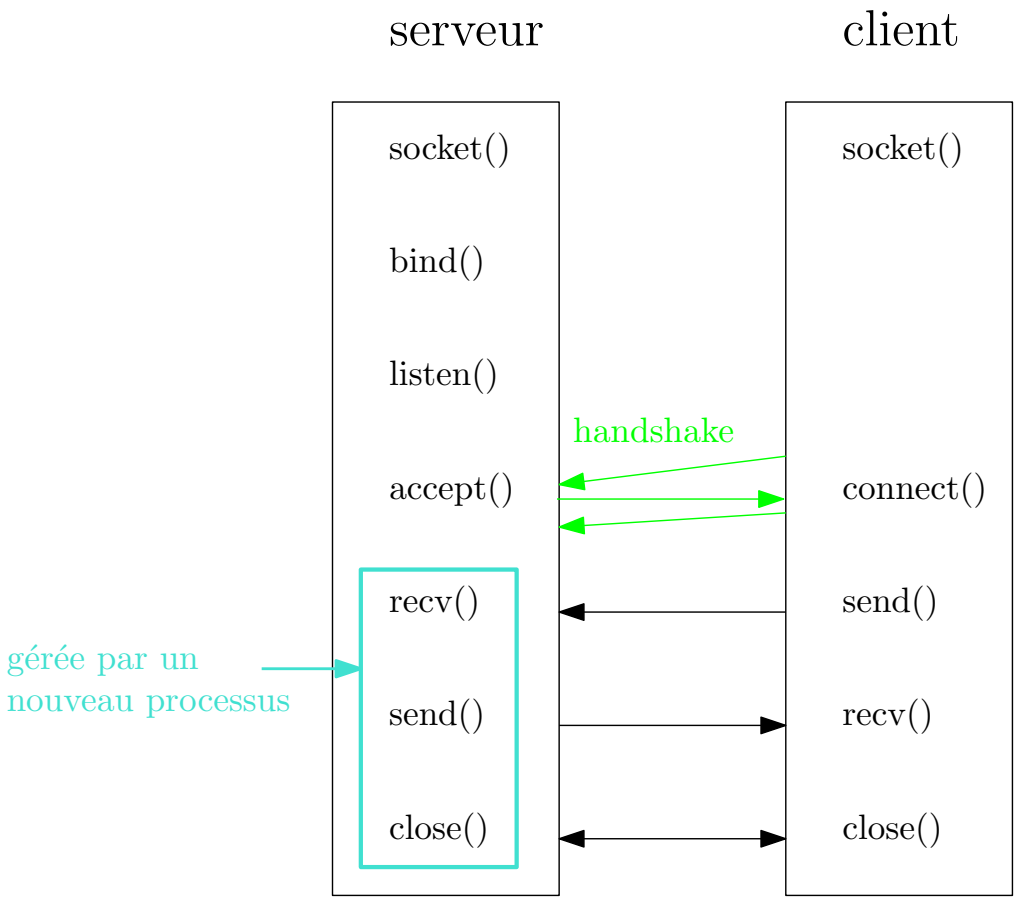
Un serveur qui ne peut s'occuper que d'un seul client à la fois peut vite être saturé, c'est-à-dire rejeter des demandes de connexion provenant de clients. Et les clients en attente de connexion peuvent s'impatienter...

Pour palier à ce problème, il y a plusieurs solutions :

- créer un nouveau processus (`fork`) à chaque nouvelle connexion d'un client,
- créer un nouveau processus léger (`thread`) à chaque nouvelle connexion d'un client,
- utiliser une socket non bloquante, c'est-à-dire, que les opérations d'acceptation d'une connexion, de reception et d'envoi sont non bloquantes.

Concurrence

Aujourd'hui, on s'intéresse aux deux premières solutions : créer un nouveau processus à chaque connexion d'un client.



Processus

Un processus est un programme en cours d'exécution.

Il est définit par

- un ensemble d'instructions à exécuter
- un environnement formé de
  - un espace d'adressage
  - ressources pour gérer les entrées/sorties de données

Plusieurs processus s'exécutent sur une même machine de façon quasi-simultanée.
C'est le système d'exploitation qui est chargé d'allouer les ressources mémoire, le temps processeur, les entrées/sorties.
D'un point de vue utilisateur, cela donne l'illusion du parallèlisme.

Rappels du cours de « Systèmes d'exploitation » :

- on crée un nouveau processus en faisant appel à la fonction `pid_t fork(void)`
- la fonction crée un nouveau processus et retourne
  - **0** pour le fils,
  - l'identifiant du nouveau processus (PID) pour le père
- à la création d'un processus fils, l'espace d'adressage du père est copié
- ensuite, les variables ne sont pas partagées entre les processus père et fils

Le serveur, après l'appel à

```
int sockclient = accept(sock, (struct sockaddr *) &adrclient, &size);
```

crée un processus fils avec `fork`.

- Le processus fils doit alors :
  - fermer son descripteur `sock`
  - communiquer avec le client via son descripteur `sockclient`
  - fermer `sockclient` à la fin de la communication
  - terminer son exécution par `exit`

- Le processus père doit :
  - fermer son descripteur `sockclient`
    Les espaces d'adressage étant copiés à la création du fils, si le père ferme son descripteur `sockclient`, cela ne ferme pas celui du fils
  - retourner sur `accept` pour attendre une nouvelle connexion
  - récupérer les processus zombies avec appel non bloquant à `waitpid`

Rappel : `pid_t waitpid(pid_t pid, int *wstatus, int options);`

- `pid = -1` si le père attend n'importe quel fils
- `wstatus = NULL` si on ne veut pas récupérer d'information sur la terminaison du processus fils
- `options = WNOHANG` afin que `waitpid` ne soit pas bloquant

Concurrence

Processus avec fork()

```
while(1){
  int sockclient = accept(sock, (struct sockaddr *) &adrclient, &size);
  if(sockclient == -1); //gérer l'erreur...

  switch(fork()){
  case -1 : break; //gérer l'erreur...
  case 0 :         //fils
    close(sock);
    int ret = communication(sockclient);
    exit(ret);
  default :        //père
    close(sockclient);
    affiche_connexion(adrclient);
    while(waitpid(-1, NULL, WNOHANG) > 0); //récupération des zombies
  }
}
```

Le problème avec `fork()` est que si les processus souhaitent partager des variables, ils doivent communiquer via une autre entité comme un tube.

Par exemple, dans ce programme, la variable `x` n'est pas partagée :

```
int main(){
  int x = 0;
  switch(fork()){
  case -1 : break; //gérer l'erreur...
  case 0 :         //fils
    x=25;
    printf("Valeur de x pour le fils %d\n",x);
    break;
  default :        //père
    sleep(2);
    printf("Valeur de x pour le pere %d\n",x);
    waitpid(-1, NULL, 0);
  }
  return 0;
}
```

L'exécution donne

```
Cours4$ ./pb_fork
Valeur de x pour le fils 25
Valeur de x pour le pere 0
```

→ les threads ou processus légers permettent le partage de variables.

Thread

Un thread est un fil d'exécution dans un programme, le programme étant lui même exécuté par un processus.

- Un processus peut avoir plusieurs threads $\rightarrow$ processus **multi-threadé**
- Chaque fil d'exécution est distinct des autres et est défini par
  - un **point courant d'exécution** (pointeur d'intstruction ou PC (Program Counter))
  - une **pile d'exécution** (stack)

Le processus principal et les threads qu'il a lancés, partagent :

- le **tas** (heap) $\rightarrow$ variables allouées avec `malloc`
- la **mémoire statique** (constantes, variables globales)
- le **code**

Un thread est donc un **processus léger** car le changement de contexte d'exécution est moins « lourd » que dans le cas d'un processus créé par `fork`. Le système à moins d'informations à charger lors du passage d'un fil d'exécution à l'autre.

Concurrence

création d'un thread

En C, la bibliothèque POSIX `pthread` permet d'utiliser des threads.

- parfois, pour compiler un programme incluant `pthread.h`, il faut compiler avec l'option `-pthread` : `gcc -Wall -pthread serveur.c -o serv`

Pour créer un thread, on a la fonction

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

- `thread` contient les données du thread créé
- `attr` contient les attributs donnés au thread à sa création (taille de la pile...). Mettre à **NULL** pour choisir les attributs par défaut
- la fonction `start_routine()` contient le code que va exécuter le thread. Elle prend en paramètre un `void *` et retourne un `void *`
- `arg` est un pointeur vers les arguments de la fonction `start_routine()`
- retourne **0** en cas de succès, un code d'erreur non nul sinon.

Concurrence

création d'un thread

```
int *sockclient = malloc(sizeof(int));
*sockclient = accept(sock, (struct sockaddr *) &caller, &size);
if (*sockclient < 0); //gérer l'erreur...

pthread_t thread;
if (pthread_create(&thread, NULL, serve, sockclient)){
  perror("pthread_create");
  continue;
}
```

Au retour de l'appel à `pthread_create`, le thread est créé et son exécution commence.

Le code exécuté par le thread est défini dans la fonction de prototype `void *serve(void *)`.

Dans notre exemple, cette fonction prend en paramètre un pointeur sur la socket `sockclient` de communication avec le client qui vient de se connecter.

Concurrence

création d'un thread

```
void *serve(void *arg) {
  int sock = *((int *) arg);      //on récupère le descripteur de socket
  char buf[SIZE_BUF+1];
  memset(buf, 0, sizeof(buf));

  int recu = recv(sock, buf, SIZE_BUF, 0);
  if (recu <= 0){
    close(sock); // fermer la socket client
    free(arg);   // libérer le pointeur de socket client
    return NULL;
  }

  printf("recu : %s\n", buf);
  char c = 'o';
  int ecrit = send(sock, &c, 1, 0);
  if(ecrit <= 0)
    perror("erreur ecriture");

  close(sock); // fermer la socket client
  free(arg);   // libérer le pointeur de socket client
  return NULL;
}
```

Attention, il faut passer en argument de la fonction `serve` une variable allouée sur le tas sinon cela peut créer des problèmes

Si on souhaite passer plusieurs arguments à la fonction `serve`

- si ces arguments sont de même type, on peut passer en argument un tableau dynamique les contenant,
- sinon, on peut passer en argument un pointeur sur une structure les contenant.

On peut également utiliser des variables globales qui sont partagées.

Si le programme principal termine avant des threads qu'il a lancés, ces derniers sont détruits
$\rightarrow$ il faut donc que le processus principal attende la fin d'exécution de ses threads. Il peut alors libérer la mémoire allouée sur le tas.

On utilise pour cela la fonction

```
int pthread_join(pthread_t thread, void **retval);
```

- la fonction est bloquante
- `thread` : thread attendu
- `retval` : si non **NULL**, permet de récupérer la valeur de retour du thread
- retourne **0** en cas de succès, un code d'erreur non nul sinon.

Concurrence

retour d'un thread

```
int compt = 0;
int *tsock[5];
pthread_t tpthread[5];
while(compt < 5){
  //*** on enregistre les pointeurs sur les descripteurs de socket client ***
  tsock[compt] = malloc(sizeof(int));
  *(tsock[compt]) = accept(sock, NULL, NULL);

  //*** on enregistre les threads ***
  if (*(tsock[compt]) >= 0) {
    if (pthread_create(&(tpthread[compt]), NULL, serve, tsock[compt])) {
      perror("pthread_create");
      continue;
    }
    compt++;
  }
}

//*** le processus principal attend les 5 threads et libère ***
//*** chaque pointeur de descripteur ***
for(int i=0; i<5; i++){
  pthread_join(tpthread[i], NULL);
  close(*tsock[i]);  // fermeture socket client
  free(tsock[i]);
}
```

Pour terminer un thread, on peut utiliser la fonction

```
void pthread_exit(void *retval);
```

- `retval` est la valeur retournée par le thread.
- ne libère pas les ressources partagées (descripteurs, verrous...)
  $\Rightarrow$ ne pas oublier de fermer la socket client et libérer la mémoire allouée
- équivalent à l'utilisation de `return` avec valeur de retour

Attention : un appel à `exit()` fait terminer le processus ! ! !
et donc tous les threads en cours qu'il a lancés...

Concurrence

retour d'un thread

```
void *serve(void *arg) {
  int sock = *((int *) arg);
  char buf[SIZE_BUF+1];
  memset(buf, 0, sizeof(buf));

  int recu = recv(sock, buf, SIZE_BUF, 0);
  if (recu <= 0) return NULL;

  close(sock);
  free(arg);

  int *ret = malloc(sizeof(int));

  if(buf[0] >= 'a' && buf[0] < 'p'){
    *ret = 1;
    pthread_exit(ret);
  }
  else{
    *ret = 2;
    pthread_exit(ret);
  }
}
```

Le processus principal attend la terminaison du thread

```
int *val;
pthread_join(thread1, (void **) &val);

if(val)
  printf("valeur de retour : %d\n", *val);
```

Tout programme a deux sections :

- une section non critique qui peut être exécutée en parallèle de n'importe quel thread,
- une section critique qui ne peut être exécutée que par un seul thread à la fois.

Dijkstra expose ce problème dans son article « *Solution of a Problem in Concurrent Programming Control*[1] » et liste les contraintes à satisfaire si on veut construire une solution au problème des sections critiques.

1. https://dl.acm.org/doi/pdf/10.1145/365559.365617

- La solution doit considérer tous les threads de la même façon et ne peut faire aucune hypothèse sur la priorité relative des différents threads.
- La solution ne peut faire aucune hypothèse sur la vitesse relative ou absolue d'exécution des différents threads. Elle doit rester valide quelle que soit la vitesse d'exécution non nulle de chaque thread.
- La solution doit permettre à un thread de s'arrêter en dehors de sa section critique sans bloquer l'accès à la section critique pour les autres thread.
- Si un ou plusieurs threads souhaitent entamer leur section critique, aucun de ces threads ne doit pouvoir être empêché indéfiniment d'accéder à sa section critique.

En C, on va utiliser des verrous ou **mutex** (abréviation de mutual exclusion) pour résoudre le problème de la section critique.

On peut schématiquement représenter un mutex comme étant une structure de données qui contient deux informations :

- la valeur actuelle du mutex (locked ou unlocked),
- une file contenant l'ensemble des threads qui sont bloqués en attente du mutex.

Les mutex sont fréquemment utilisés pour protéger l'accès à une zone de mémoire partagée. Ils sont partagés entre les threads.

Le principe est le suivant :

- un thread qui veut accéder à des données partagées demande le mutex.
- Si celui-ci est libre, il l'obtient et continue son exécution,
- sinon, il bloque jusqu'à ce que le mutex soit libéré.
- Lorsque le thread a terminé avec les données partagées, il libère le mutex.

En pratique, on commence par déclarer et initialiser un verrou :

```
pthread_mutex_t verrou = PTHREAD_MUTEX_INITIALIZER;
```

Puis, lorsqu'on veut protéger une section de code, on fait appel à

- `int pthread_mutex_lock(pthread_mutex_t *mutex);` pour prendre un verrou,
- `int pthread_mutex_unlock(pthread_mutex_t *mutex);` pour libérer un verrou.

Ce qui donne

```
pthread_mutex_lock(&verrou);
// section critique...
pthread_mutex_unlock(&verrou);
```

En pratique, une section critique étant une zone de mémoire partagée peut correspondre à :

- une variable globale,
- une variable sur le tas (pointeur),
- un fichier.

Attention, on utilise des verrous différents pour protéger les accès à différentes zones de mémoire.

Concurrence

Les mutex

```
#define LEN 15
int var = 0;
pthread_mutex_t verrou = PTHREAD_MUTEX_INITIALIZER;

void *serve(void *arg) {
  pthread_mutex_lock(&verrou);
  var += 1;
  pthread_mutex_unlock(&verrou);
  return NULL;
}

int main(void){
  int compt = 0;
  pthread_t tpthread[LEN];

  while(compt < LEN){

    if (pthread_create(&tpthread[compt], NULL, serve, NULL)){
      perror("pthread_create");
      continue;
    }
    compt++;
  }

  for(int i=0; i<LEN; i++)
    pthread_join(tpthread[i], NULL);

  return 0;
}
```

On détruit un mutex avec la fonction
`int pthread_mutex_destroy(pthread_mutex_t *mutex)`

Si on veut réinitialiser le mutex, il faut alors faire appel à
```
int pthread_mutex_init(pthread_mutex_t *restrict mutex,
          const pthread_mutexattr_t *restrict attr);
```

En pratique, pour ce cours, `attr` a la valeur **NULL**.

```
pthread_mutex_t verrou = PTHREAD_MUTEX_INITIALIZER;

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

 //utilisation du mutex verrou...

  pthread_mutex_destroy(&verrou);
  pthread_mutex_init(&verrou, NULL);

 //utilisation de mon nouveau mutex verrou...

}
```

Les mutex permettent de synchroniser des threads en controlant l'accès à certaines variables ou fichiers.

Si on souhaite synchroniser les threads lorsqu'un certain état est atteint, on peut :

- faire de l'attente active pour surveiller l'état. Si, par exemple, on surveille la valeur d'une variable, on interroge en boucle sa valeur.
  $\rightarrow$ **à éviter** car consommatrice de ressources, énergie...
- utiliser une **variable condition**.

L'état à surveiller, que l'on nomme **condition**, peut être :

- la valeur d'une variable partagée par les différents threads,
- le contenu d'un fichier.

Une variable condition s'utilise en conjonction avec un mutex et une condition.

Pour initialiser la variable, on peut le faire :

- de façon statique :

```
pthread_cond_t vcond = PTHREAD_COND_INITIALIZER;
```

- de façon dynamique, en utilisant la fonction :
  `int pthread_cond_init(pthread_cond_t *cond, const pthread_condattr_t *attr);`
  En pratique, pour ce cours, `attr` a la valeur **NULL**.

```
if(pthread_cond_init(&vcond, NULL))
  perror("pthread_cond_init");
```

Pour détruire la variable condition, lorsqu'on en a plus besoin :
`int pthread_cond_destroy(pthread_cond_t *cond);`

Une fois le mutex et la variable condition initialisés, on utilise les fonctions suivantes pour la synchronisation :

- `int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);`
  bloque le thread courant en attente de la réalisation de la condition indiquée par la variable condition pointée par `cond`,

- `int pthread_cond_signal(pthread_cond_t *cond);`
  permet de débloquer *un thread* parmi les threads bloqués avec la variable condition pointée par `cond`,

- `int pthread_cond_broadcast(pthread_cond_t *cond);`
  permet de débloquer *tous les threads* bloqués avec la variable condition pointée par `cond`.

`pthread_cond_wait` :

- doit être appelée après avoir vérouillé le mutex associé à la variable condition,
- le mutex est automatiquement relâché pendant l'attente,
- lorsque le thread est réveillé par un autre thread invoquant `pthread_cond_signal` ou `pthread_cond_broadcast`, le mutex est automatiquement vérouillé. Le programme doit donc penser à le relâcher ensuite.

`pthread_cond_signal` ou `pthread_cond_broadcast` :

- doit être appelée après avoir vérouillé le mutex correspondant à la variable condition,
- doit donc relâcher le verrou juste après afin de permettre à `pthread_cond_wait` de terminer.
- doit être appelée après que les threads soient bloqués dans `pthread_cond_wait` sinon **des threads peuvent rester définitivement bloqués**.
  $\Rightarrow$ on teste la condition pour décider si le thread doit appeler `pthread_cond_wait`.

Concurrence

Variable condition

```
for(int i=0; i<10; i++)
  pthread_create(&pthread, NULL, serve, NULL);

  // ...

  pthread_mutex_lock(&verrou);

  pthread_cond_broadcast(&vcond);

  pthread_mutex_unlock(&verrou);
```

```
void *serve(void *arg) {
  pthread_mutex_lock(&verrou);

  // si la condition n'est pas réalisée
  // le thread est mis en attente
  if(...)
    pthread_cond_wait(&vcond, &verrou);

  // modification ou consultation
  // de la variable partagée ou du fichier
  // ...

  pthread_mutex_unlock(&verrou);

  // ...

  return NULL;
}
```

Ici, tous les threads sont réveillés simulatnément.

Pour réveiller les threads l'un après l'autre :

```
for(int i=0; i<10; i++)
  pthread_create(&pthread, NULL, serve, NULL);

  // ...

  pthread_mutex_lock(&verrou);

  for(int i=0; i<len; i++)
      pthread_cond_signal(&vcond);

  pthread_mutex_unlock(&verrou);
```

```
void *serve(void *arg) {
  pthread_mutex_lock(&verrou);

  // si la condition n'est pas réalisée
  // le thread est mis en attente
  if(...)
    pthread_cond_wait(&vcond, &verrou);

  // modification ou consultation
  // de la variable partagée ou du fichier
  // ...

  pthread_mutex_unlock(&verrou);

  // ...

  return NULL;
}
```

L'ordre dans lequel les threads sont réveillés est aléatoire.