Programmation système

Ressource R3.05 - API threads posix

monnerat@u-pec.fr

IUT de Fontainebleau

Sommaire

Rappels

Rappels

Mémoire virtuelle du processus

- Chaque processus est exécuté comme s'il avait toute la mémoire.
- Un processus peut accéder uniquement à sa mémoire.
- La mémoire virtuelle est décomposée en segments (facilite le partage et la protection)

Processus

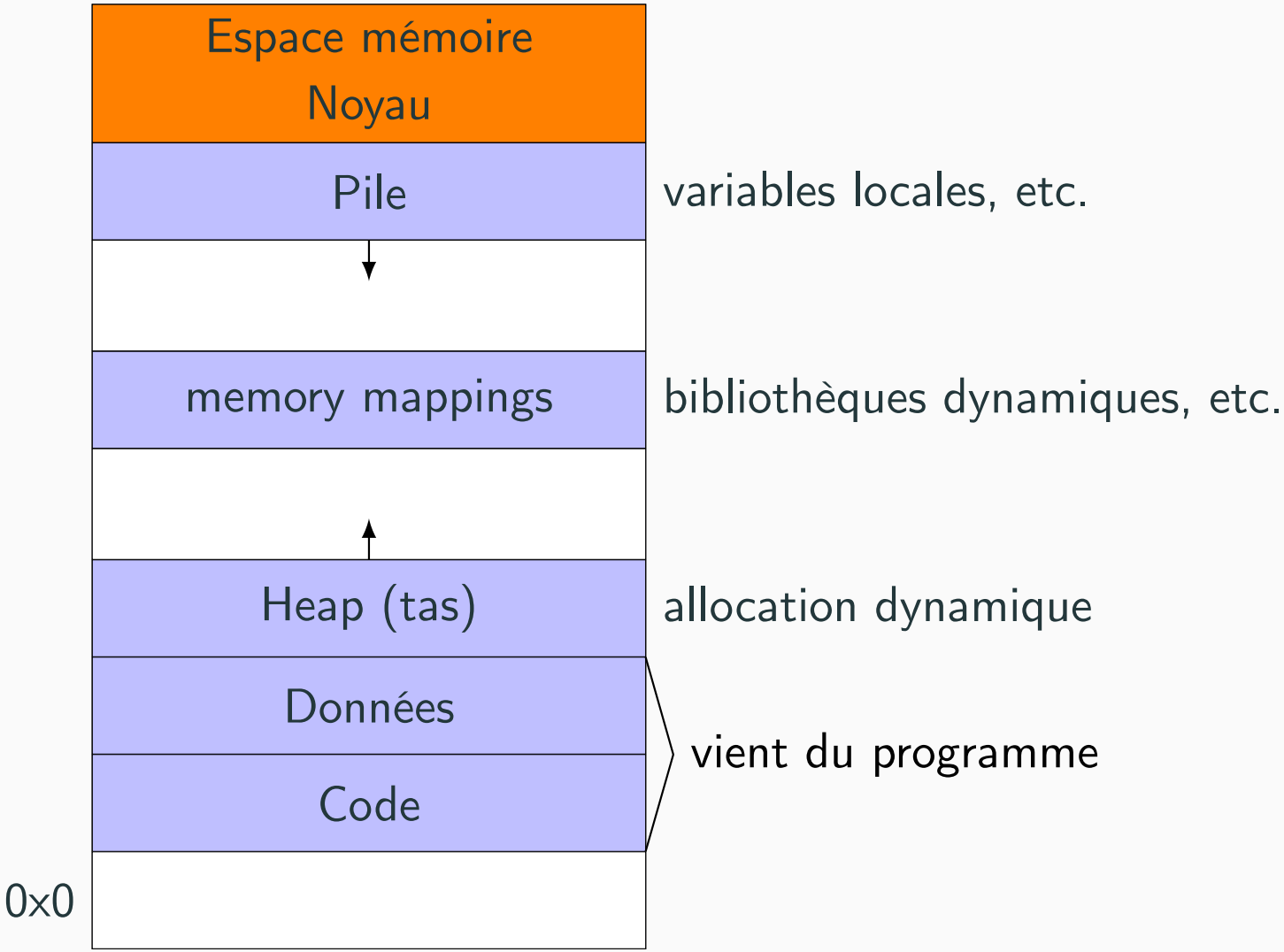
Image dynamique de l'exécution d'un programme.

- Un programme (fichier exécutable - format ELF) est statique. Il fournit :
  - Le code (instructions) - Code Segment
  - Les données - Data Segment
- Son exécution par le SE est dynamique $\Rightarrow$ Processus

Durant toute sa vie, l'état d'un processus comprend :

- mémoire : code, data, tas, pile.
- contexte d'exécution : registres, compteur ordinal, sommet de pile, etc.
- Process control Bloc (PCB) : état vis à vis du SE.

Mémoire du processus



cat /proc/pid/maps

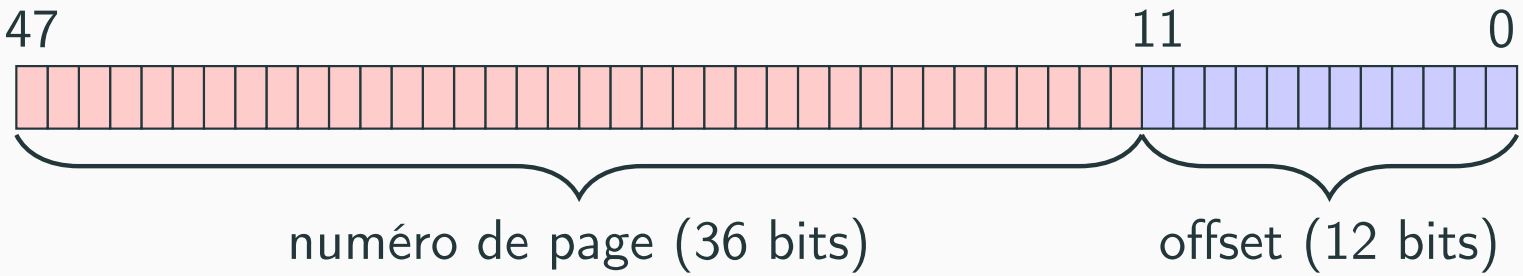
```
558d6b6fa000-558d6b6fb000 r--p 00000000 08:04 541510                     a.out
558d6b6fb000-558d6b6fc000 r-xp 00001000 08:04 541510                     a.out
558d6b6fc000-558d6b6fd000 r--p 00002000 08:04 541510                     a.out
558d6b6fd000-558d6b6fe000 r--p 00002000 08:04 541510                     a.out
558d6b6fe000-558d6b6ff000 rw-p 00003000 08:04 541510                     a.out
558d6ce5f000-558d6ce80000 rw-p 00000000 00:00 0                          [heap]
7f0fc83ed000-7f0fc83ef000 rw-p 00000000 00:00 0
7f0fc83ef000-7f0fc8415000 r--p 00000000 08:03 1969089                    /usr/lib/libc-2.32.so
7f0fc8415000-7f0fc8562000 r-xp 00026000 08:03 1969089                    /usr/lib/libc-2.32.so
7f0fc8562000-7f0fc85ae000 r--p 00173000 08:03 1969089                    /usr/lib/libc-2.32.so
7f0fc85ae000-7f0fc85b1000 r--p 001be000 08:03 1969089                    /usr/lib/libc-2.32.so
7f0fc85b1000-7f0fc85b4000 rw-p 001c1000 08:03 1969089                    /usr/lib/libc-2.32.so
7f0fc85b4000-7f0fc85ba000 rw-p 00000000 00:00 0
7f0fc8600000-7f0fc8602000 r--p 00000000 08:03 1969032                    /usr/lib/ld-2.32.so
7f0fc8602000-7f0fc8623000 r-xp 00002000 08:03 1969032                    /usr/lib/ld-2.32.so
7f0fc8623000-7f0fc862c000 r--p 00023000 08:03 1969032                    /usr/lib/ld-2.32.so
7f0fc862c000-7f0fc862d000 r--p 0002b000 08:03 1969032                    /usr/lib/ld-2.32.so
7f0fc862d000-7f0fc862f000 rw-p 0002c000 08:03 1969032                    /usr/lib/ld-2.32.so
7fffbe4a3000-7fffbe4c4000 rw-p 00000000 00:00 0                          [stack]
7fffbe59b000-7fffbe59f000 r--p 00000000 00:00 0                          [vvar]
7fffbe59f000-7fffbe5a1000 r-xp 00000000 00:00 0                          [vdso]
ffffffffff600000-ffffffffff601000 --xp 00000000 00:00 0                  [vsyscall]
```

Il s'agit d'adresses virtuelles !

Table de pages

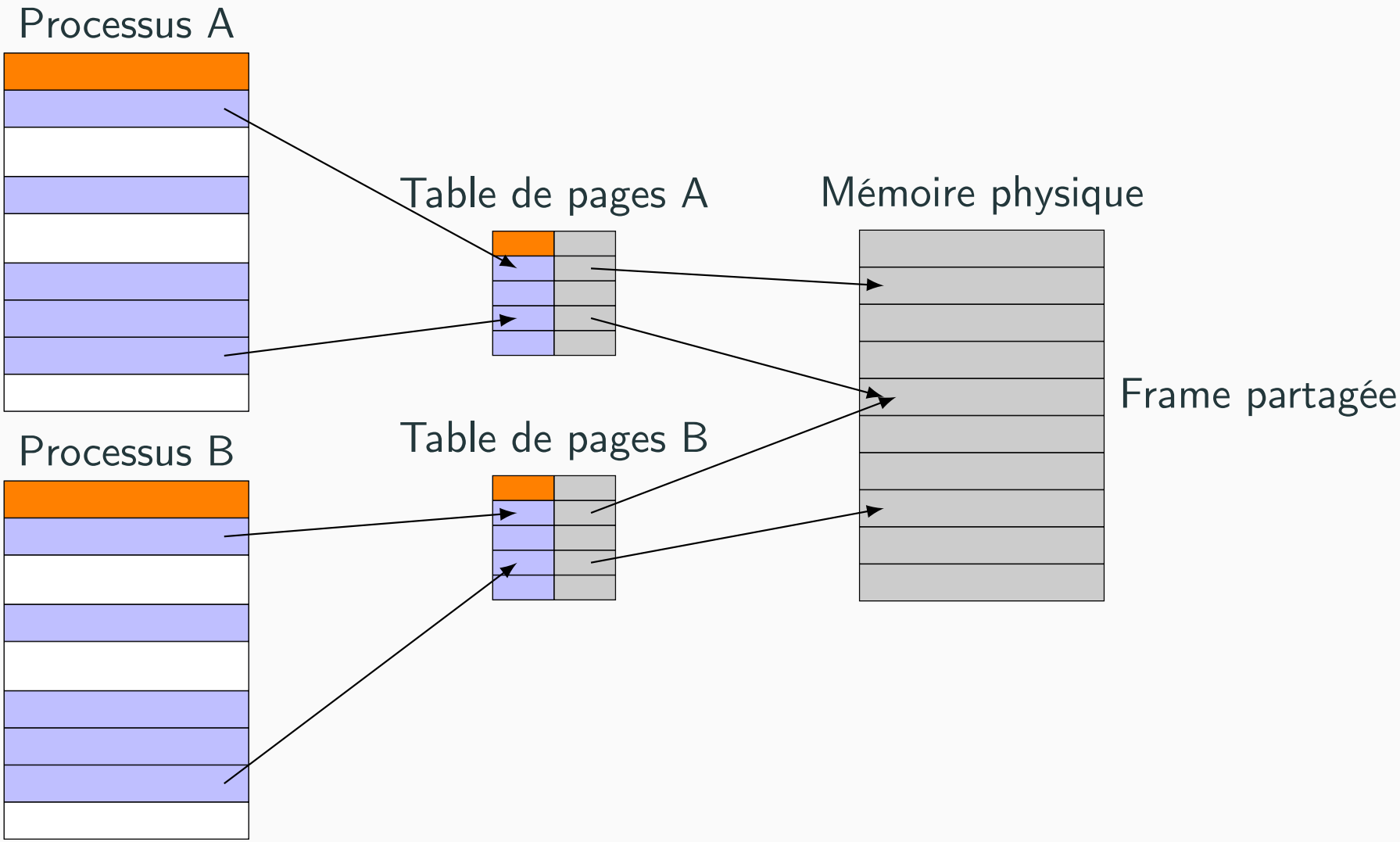
Elle assure la correspondance entre une adresse virtuelle et une adresse physique.

- La taille des pages est une puissance de 2 : 4 ko en général.
- L'adresse virtuelle est découpé en 2 morceaux : numero de page / offset



- Chaque processus à sa propre table.
- La même adresse virtuelle peut être utilisée dans 2 processus différents et mappée avec des adresses physiques différentes. (il y a par exemple sur x86 un registre (CR3) qui pointe sur le repertoire de page actif)

Mémoire virtuelle vs mémoire physique



Le code

- Chargement en mémoire de l'éxecutable (Linux : format ELF)

Les données

- Variables globales
- Variables locales static

```c
#include <stdio.h>

int i=3;
int j;
void count(){
  static int cpt = 0;
  cpt++;
}
```

La pile

- Variables locales, paramètres de fonctions.
- Allouées et desallouées "automatiquement" (pas de `free`)

```
int * f(){
  int i;
  return &i; /* illegal ! */
}
```

- Allocation sur la pile.

```
void stack_alloc(int n){
int * arr = alloca(n*sizeof(int));
return;
}
```

Le tas

- Gestion dynamique de la mémoire.
- `malloc`, `calloc`, `free`, etc.

```
void heap_alloc(int n){

int * arr1,*arr2;
arr1 = (int*) malloc(n*sizeof(int));
arr1 = (int*) calloc(n,sizeof(int));



free(arr1);
free(arr2);
}
```

Durée de vie d'une variable

```
char ch1 = 'A';
const char ch2 = 'B';

void f(void) {
    char ch3 = 'C';
    char* ch4 = (char*)malloc(sizeof(char));
}
```

- *statique* : durée de vie du processus (ch1, ch2).
- *automatique* : le compilateur alloue/désalloue automatiquement en fonction de la portée (ch3,ch4).
- *dynamique* : le programmeur alloue/désalloue lui-même (*ch4).

Gestion du tas

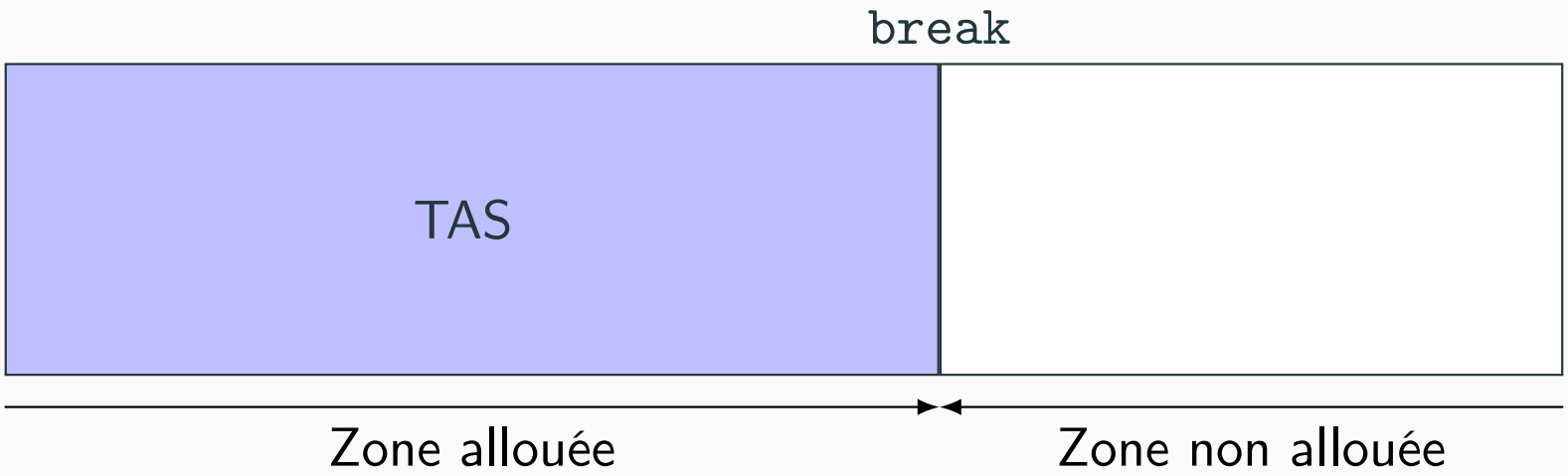
Changement de sa taille

- Déplacement absolu

```
int brk(void *addr);
```

- Déplacement relatif

```
void *sbrk(intptr_t increment);
```



mmap

```
void *mmap(void *addr, size_t length, int prot, int flags,
                    int fd, off_t offset);
```

`mmap()` est le moyen le plus standard d'allouer de grande quantité de mémoire en espace user.

- Bien que souvent utilisé pour des fichiers, le flag `MAP_ANONYMOUS` permet d'allouer de la mémoire au processus.
- `MAP_SHARED` permet de partager des pages avec d'autres processus.

La taille demandée est alignée sur la taille des pages.

L'implantation des fonctions `malloc/calloc` peuvent utiliser soit `brk` ou `mmap`

`mallopt` et le paramètre `M_MMAP_THRESHOLD` permet de contrôler le comportement.

Interêt : rendre la partie martérielle plus simple et rapide.

Exemple :

Imaginons un cache (cpu) avec des blocks de 128 octets. Ces lignes auront des adresses systématiquement alignés.

Les adresses `127,128,129,130` vivent dans 2 blocs différentes :

- `[0,127]`
- `[128,255]`

Un entier (4 octets) avec une adresse alignée sur 4
`[4n,4n+1,4n+2,4n+3]` est toujours dans le même bloc !

Alignement

x86-64 Linux. `T` type primitif

Type	Taille	Adresse	(alignof(Type))
`char` (signed , unsigned)	1	/	1
`short` (unsigned short)	2	Multiple de 2	2
`int` (unsigned int)	4	Multiple de 4	4
`long` (unsigned long)	8	Multiple de 8	8
`float`	4	Multiple de 4	4
`double`	8	Multiple de 8	8
`long double`	16	Multiple de 16	16
`T*`	8	Multiple de 8	8

alignof(T) == sizeof(T)

`malloc()`

- renvoie un pointeur générique, qui doit pouvoir être casté vers `T*` pour n'importe quel type `T`.
- dans la pratique, `malloc` renvoie une adresse aligné sur 16.

Cas des structures

- la taille est un multiple de l'alignement.
- l'adresse du premier membre est l'adresse de la structure.
- les membres sont rangés dans l'ordre avec les contraintes d'alignement propre à leur type (padding souvent nécessaire).
- l'alignement de la structure est égale à l'alignement maximal de ses membres (le ppcm, mais il s'agit toute de puissance de 2).

On peut allouer en alignant (sur la taille du cache par exemple) avec la fonction

```
int posix_memalign(void **memptr,
                   size_t alignment, size_t size);
```

Rappels de C

Rappels de C

Pointeur de fonctions, type `void *`

En C, l'identitiant d'une fonction est son adresse (son point d'entrée) en mémoire.

- Il est possible de manipuler les adresses des fonctions en C, et d'avoir des pointeurs sur des fonctions.
- Il est obligatoire de savoir manipuler les pointeurs sur les fontions pour gérer les signaux et les threads.

Le type d'un pointeur sur fonction doit contenir les types des paramètres de la fonction, et le type de retour.

- les paramètres n'ont pas besoin d'avoir de nom,
- le compilateur doit juste savoir quel type empiler sur la pile

Pour déclarer un pointeur sur une fonction :

```
type_retour (*nompointeur) (liste_arguments...);
```

Exemple :

```
int (*fct)(int);
```

déclare `fct` comme un pointeur sur une fonction prenant en argument un entier, et renvoyant un entier. Appeler une fonction par un pointeur se fait de la même manière que pour une fonction normale.

```c
#include <stdio.h>
int carre (int x) {
  return x*x;
}
int cube (int x) {
  return x*x*x;
}
void boucle ( int (*f) (int)) {
  int i ;
  for (i=1; i <=10; i++)
    printf ("%d : %d\n",i, f(i)); // equivalent à (*f)(i)
}
int main(){
  boucle(&carre);  // equivalent à boucle(carre)
  boucle(&cube);
}
```

Passage de paramètre générique à une fonction

Beaucoup de fonctions de librairies standards du C utilise le type `void *`.

```
void *malloc(size_t size);
void free(void *ptr);
void *calloc(size_t nmemb, size_t size);
void *realloc(void *ptr, size_t size);
void bzero(void *s, size_t n);
void qsort(void *base, size_t nmemb, size_t size,
    int (*compar)(const void *, const void *));
```

- Il s'agit d'un pointeur (adresse) "générique" (sans type), qui permet de transmettre à une fonction une adresse de n'importe quelle type : `char *`,`int *`, `int **`, `etc.`
- La manière d'interpréter cette adresse est laissée "libre".
- Normalement, il est interdit de faire de l'arithmétique sur un pointeur `void *`. `gcc` le permet (`-Wpointer-arith` donnera une erreur).

Exemple 1 : memcpy()

```
void *
memcpy (void *dest, const void *src, size_t len)
{
  char *d = dest;
  const char *s = src;
  while (len--)
    *d++ = *s++;
  return dest;
}
```

Exemple 2 : `atexit()`

```c
#include <stdlib.h>
int atexit(void (*function)(void));
```

`atexit()` enregistre une fonction qui sera appelée à la fin (normale) du processus.

Exemple 3 : qort()

```c
#include <stdlib.h>

void qsort(void *base, size_t nmemb, size_t size,
    int (*compar)(const void *, const void *));
```

Le troisième argument représente la fonction de comparaison (ordre) que qsort() doit utiliser. pour trier le tableau dans l'ordre croissant.

Rappels de C

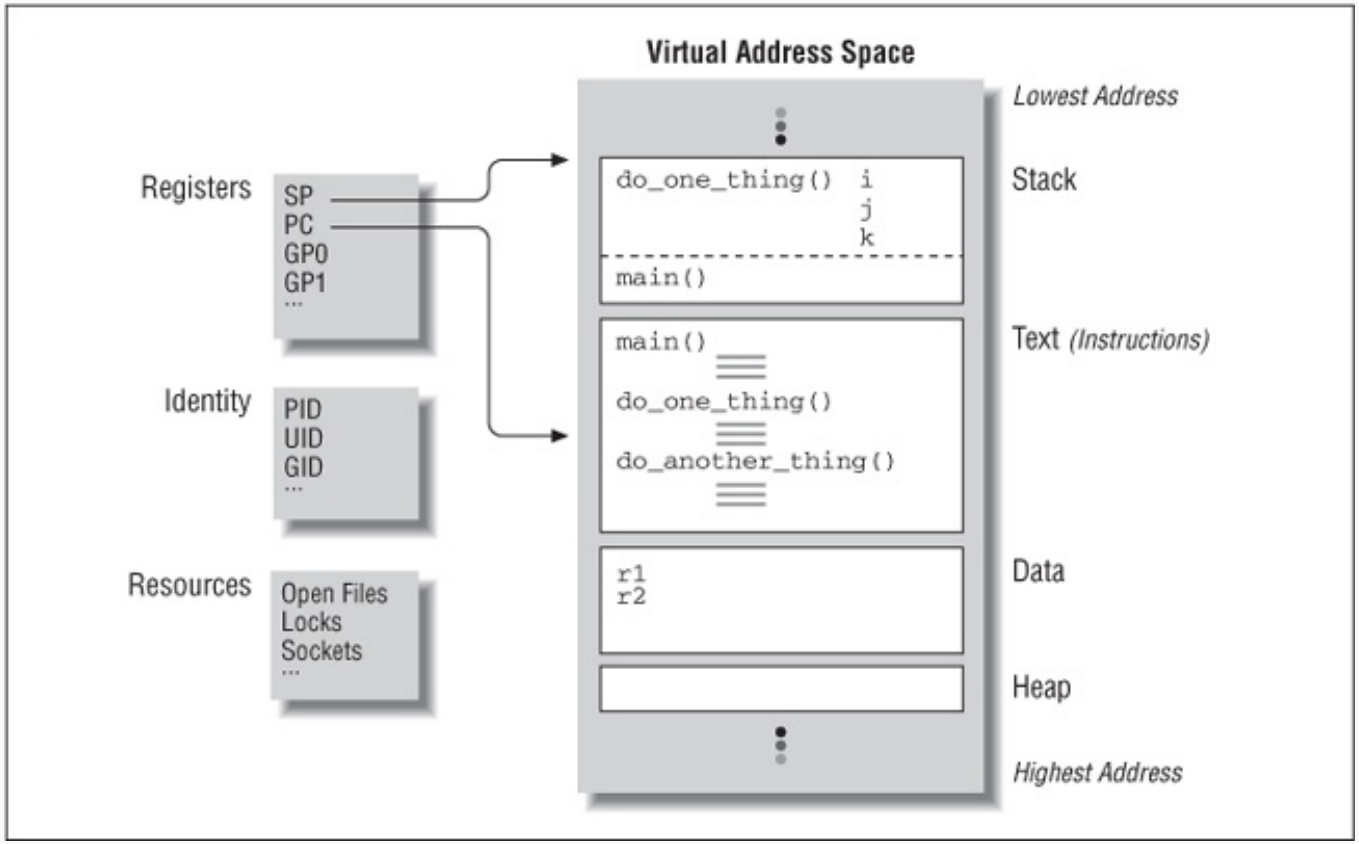
Rappels sur le (pseudo-)parallélisme

Soit le programme suivant

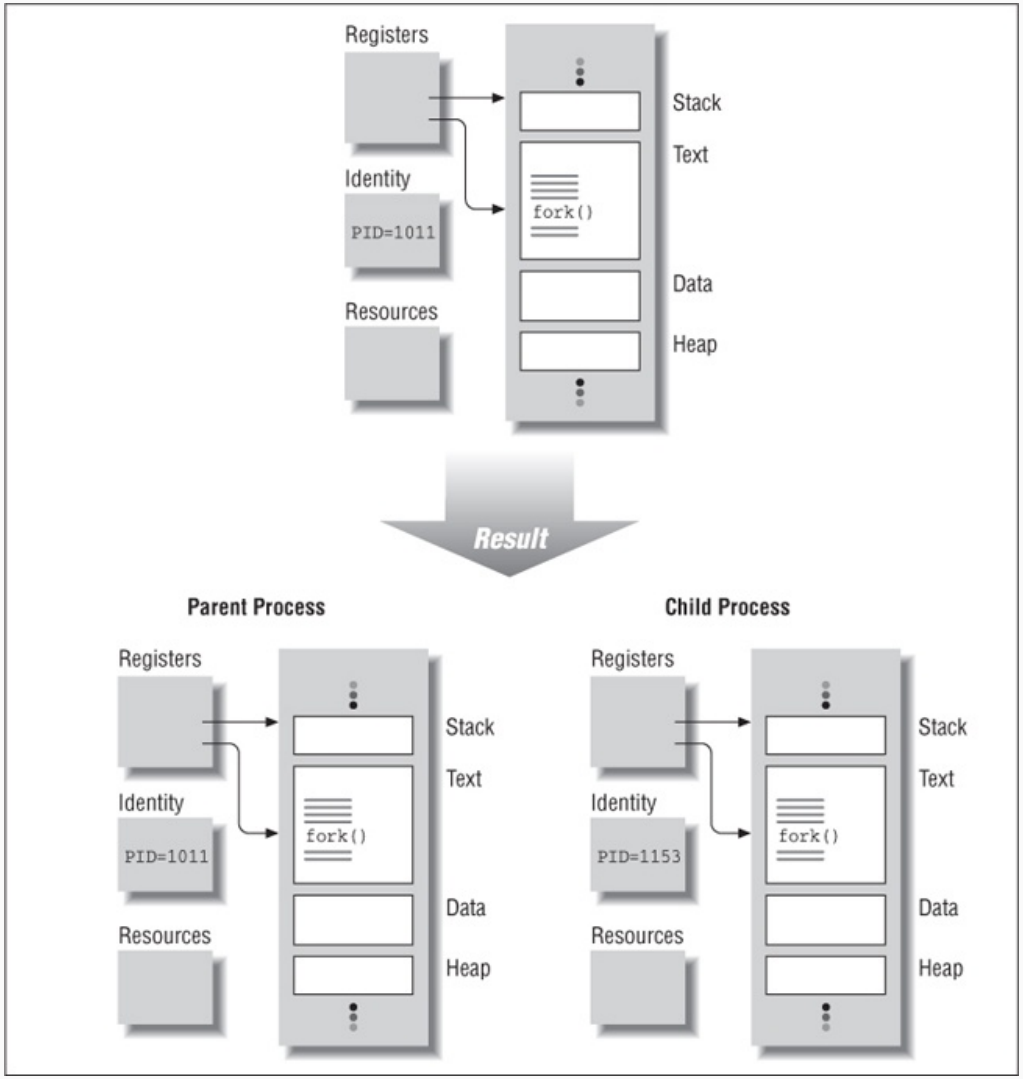
```c
int r1 = 0, r2 = 0;

int main(void)
{
  do_one_thing(&r1);
  do_another_thing(&r2);
  do_wrap_up(r1, r2);
  return 0;
}

void do_one_thing(int *pnum_times)
{
  int i, j, x;

  for (i = 0;    i  <  4;  i++) {
    printf("doing one thing\n");
    for (j = 0; j < 10000; j++) x = x + i;
    (*pnum_times)++;
  }
}
```

```c
void do_another_thing(int *pnum_times)
{
  int i, j, x;

  for (i = 0;    i  <  4;  i++) {
    printf("doing another \n");
    for (j = 0; j < 10000; j++) x = x + i;
    (*pnum_times)++;
  }
}

void do_wrap_up(
    int one_times,
    int another_times
    )
{
  int total;

  total = one_times + another_times;
  printf("wrap up: ");
  printf("one thing %d, another %d, total %d\
      ,one_times, another_times, total);
}
```

Exécution dans un processus

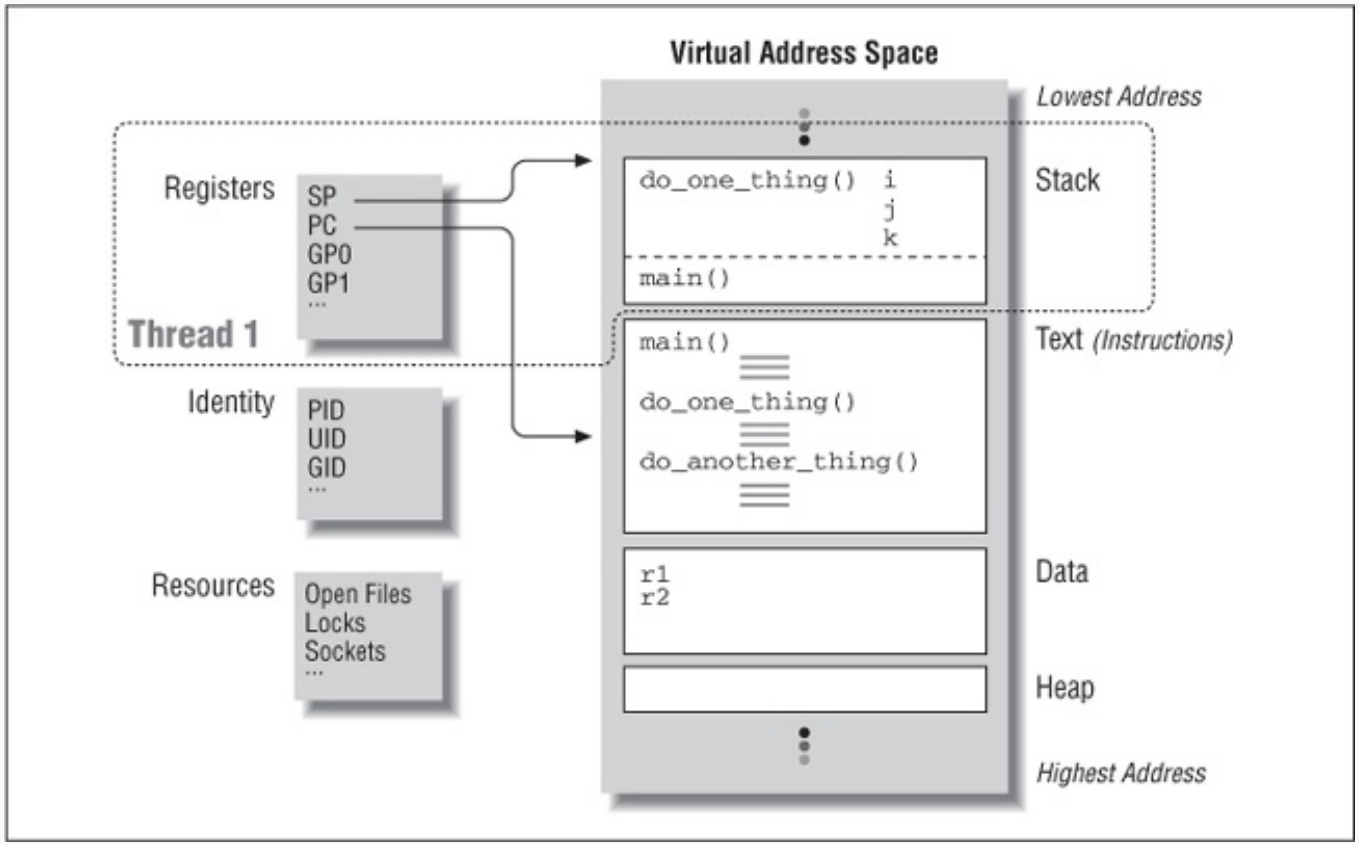


Exécution dans deux processus

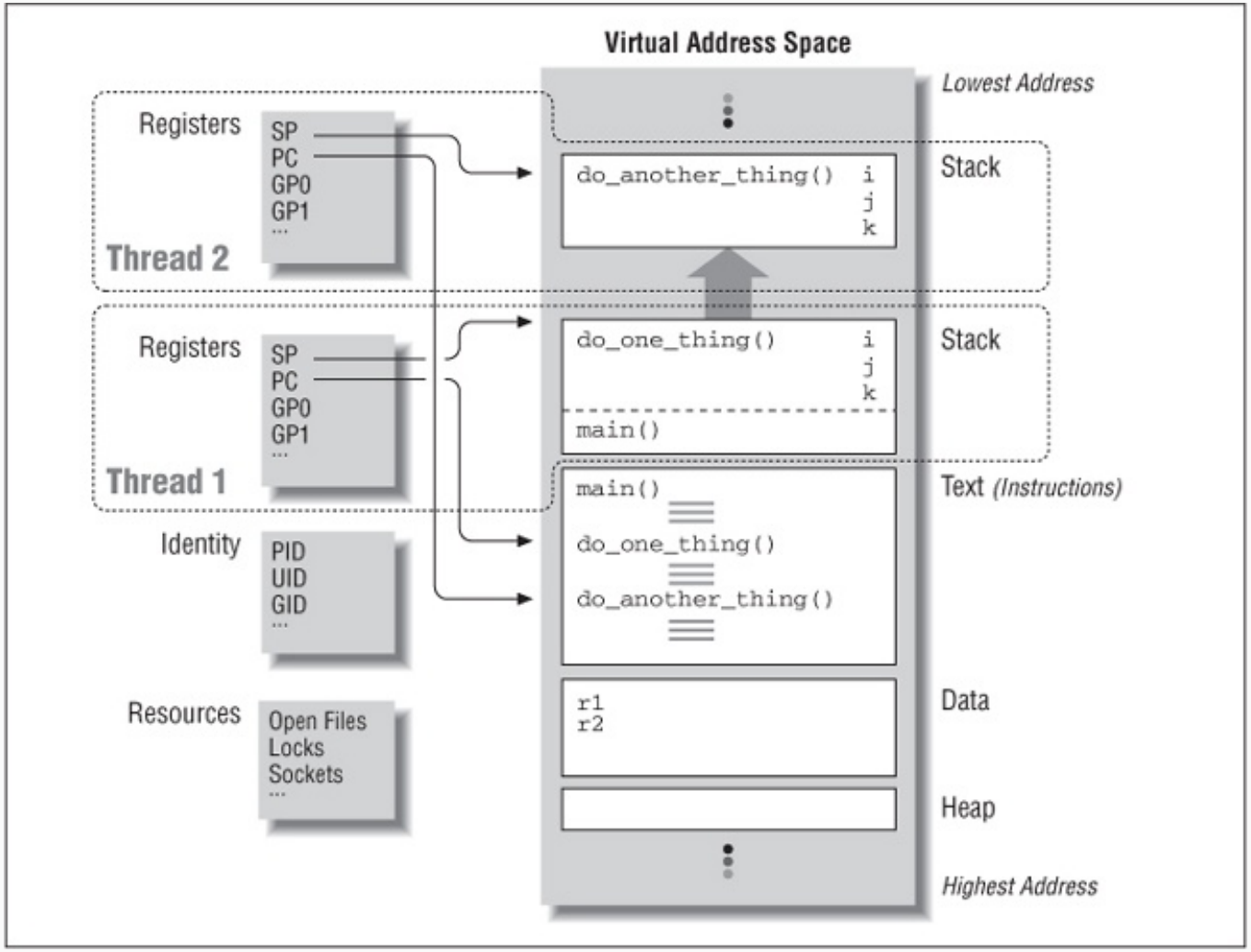


Nécessité de mécanismes de communication.

Exécution dans un processus avec un seul thread

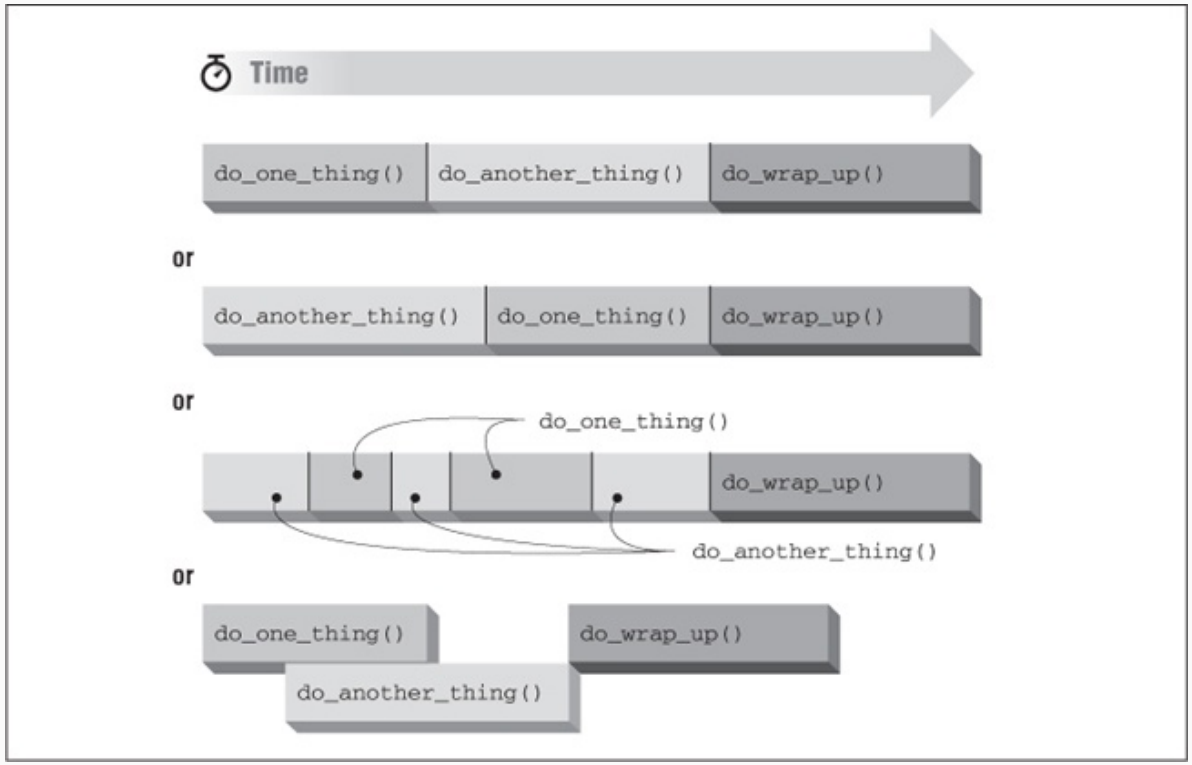


Exécution dans un processus avec deux threads



Décompostion d'un processus en sous-tâches dont les exécutions peuvent être entrelacées et/ou parallèles.

Différentes traces d'exécutions suivant le modèle d'exécution



- 1 et 2 sont séquentielles.
- 3 est entrelacée (pseudo-parallèlisme).
- 4 est parallèle.

L'ordre de `one_thing` et `another_thing` n'a pas d'importance.
Parallélisme possible.

Interêts du pseudo-parallèlisme (concurrence), un seul processeur.

- Recouvrir les temps de latences (E/S). Si une tâche est en attente d'un ressource, inutile qu'elle ait accès au processeur.
- Plusieurs tâches progressent en même temps (pas simultanèment).
- Propriété de vivacité.

Interêts du parallèlisme, multiprocesseurs

- Accélerer les traitements : exécution de tâches simultanément.
- Certains problèmes s'y prêtent, d'autres beaucoup moins.

Évidemment, on peut faire les deux !

Threads Posix

Les threads

Threads (processus légers, fils d'exécution) est une unité d'exécution dans un processus

- même processus
- piles/registres indépendants - plusieurs fils d'exécution
- partagent la mémoire du processus - code, données, tas, etc.
- partagent les fichiers ouverts, les gestionnaires de signaux, etc.

La mémoire est partagée

- création plus rapide
- commutation de contexte plus rapide
- communication plus rapide
- programmation plus délicate

Sous linux, bibliothéque pthread. Compilez avec `-lpthread`.

Création processus vs thread : comparaison

```c
void do_nothing(void){
  exit(0);

}
int main(int argc, char * argv[]){
  int n = strtol(argv[1],NULL,0);
  while(n--){
    if (fork()==0) donothing();
    wait(NULL);
  }
}
```

```c
void * do_nothing(void *x){
  pthread_exit(NULL);
}
int main(int argc, char * argv[]){
  int n = strtol(argv[1],NULL,0);
  pthread_t th;
  while(n--){
    pthread_create(&th,NULL,do_nothing,NULL);
    pthread_join(th,NULL);
  }
}
```

n	fork()			pthread_create()		
	real	user	sys	real	user	sys
1000	0.252s	0.153s	0.105s	0.048s	0.08s	0.040s
10000	2.389s	1.438s	1.018s	0.384s	0.096s	0.335s
100000	24.1s	15.1s	9.882s	3.542s	0.666s	3.437s

Table 1: avec la commande `time`

Threads Posix

Api

Api de base de la gestion de threads en C via la librairie POSIX, `pthreads`.

Toutes les fonctions sont définies dans l'entête `pthread.h`

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

Il faut également faire l'édition des liens avec la librairie `pthread`, option `-lpthread`.

```
gcc -o main main.c -lpthread
gcc -g -fsanitize=address -fsanitize=leak -fsanitize=thread
    -o main main.c -lpthread
```

(On peut activer `AddressSanitizer`, `LeakSanitizer`, `ThreadSanitizer`)

Identitification

- Interne : type `pthread_t`

```
pthread_t pthread_self(void);
int pthread_equal(pthread_t t1, pthread_t t2);
```

- Externe : TID/LWP

```
$ ps -Lf -C a.out
UID          PID    PPID     LWP  C NLWP STIME TTY          TIME CMD
denis      16091   15695   16091  0    2 10:57 pts/2    00:00:00 ./a.out
denis      16091   15695   16092  0    2 10:57 pts/2    00:00:00 ./a.out
```

- L'appel système

  `pid_t gettid(void);`

  permet de récupérer le tid.

Création

```c
#include <pthread.h>
int pthread_create(
    pthread_t *thread,
    const pthread_attr_t *attr,
    void *(*start_routine) (void *),
    void *arg
    );
```

Crée un nouveau thread qui exécute `start_routine(arg)`

```c
void * start_routine(void *arg){...}
```

- `thread` reçoit l'identifiant du thread crée.
- `attr` précise les attributs du thread.

Retourne 0 (succès) ou -1 (echec).

Terminaison

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

Termine le thread appelant, équivalent à un return dans le thread. (pas vrai pour le thread main).

La valeur de retour `retval` est disponible dans un thread (du même processus) avec `pthread_join` :

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

Cette fonction est bloquante. Les ressources (la pile en autres) du thread qui s'est terminé sont libérées.

Remarque : le paramètre `void**retval` est un pointeur vers le type de retour de la fonction `start_routine`.

Il est important que `retval` soit un pointeur de pointeur, car on change la valeur passée en argument. Ainsi, si cette valeur est un pointeur, la seule façon de la modifier est d'avoir une indirection supplémentaire (double pointeur).

On peut mettre ce paramètre à `NULL`.

Détachement

Un thread détaché est un thread dont les ressources seront libérées dès sa terminaison.

```
int pthread_detach(pthread_t thread);
```

- Avantage : libération automatique.
- Inconvénient : synchronisation impossible sur la fin du thread avec `pthread_join`.

Par défaut, (paramètre `attr` à NULL dans `pthread_create`) un thread est "joinable" (`PTHREAD_CREATE_JOINABLE`)

Attributs de thread

À la création, il faut passer un `pthread_attr *` à la fonction `phtread_create()`. Cette structure est initialisée/détruite avec

```
int pthread_attr_init(pthread_attr_t *attr);
int pthread_attr_destroy(pthread_attr_t *attr);
```

Il existe plusieurs attributs configurables (RTFM). À titre d'exemple, on peut configurer l'attribut `PTHREAD_CREATE_DETACHED` avec

```
int pthread_attr_setdetachstate(pthread_attr_t *attr,
    int detachstate);
```

Terminaison alternative

- L'autoterminaison : l'appel à `pthread_exit()` termine le thread et retourne `retval` au thread effectuant la jointure.
  Un cas particulier est lorsque le thread principal appelle `pthread_exit()`. Le processus attend la fin de tous les threads, alors qu'un `return` dans le thread principal provoque la mort de tout le processus
- L'annulation

```c
int pthread_cancel(pthread_t thread);
```

  Attention, un thread n'est pas toujours "annulable" (c'est la cas par défaut).

Threads Posix

Exclusion mutuelle : Verrous

Mutex

Peut être dans deux états :

- pris par un (et un seul) thread (verrouillé).
- pris par aucun thread (déverrouillé).

Tout thread qui souhaite prendre un mutex verrouillé doit attendre.

Il n'y a pas de file d'attente dédiée.

Création/destruction

```
int pthread_mutex_init(pthread_mutex_t *restrict mutex,
    const pthread_mutexattr_t *restrict attr);

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

Verrouillage/Déverrouillage

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
```

Verrouille le mutex :

- Si mutex est déjà verrouillé par un autre thread ⇒ suspend le thread appelant jusqu'au déverrouillage du mutex
- Si mutex est déverrouillé ⇒ retourne immédiatement

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

Déverrouille le mutex

Attention : le mutex doit être verrouillé par le thread appelant.

Renvoient 0 en cas de succès, sinon un code d'erreur non-nul

```c
pthead_mutex_t m=PTHREAD_MUTEX_INITIALIZER;
void * f(void * arg){
  int i, *n=(int*)arg;
  for(i=0;i<10000;i++){
    pthread_mutex_lock(&m); //<-
    (*n)++;
    pthread_mutex_unlock(&m);//<-
  }
}
int main(){
  int n=0;
  pthread_t t1,t2;
  pthread_create(&t1,NULL,f,&n);
  pthread_create(&t2,NULL,f,&n);
  pthread_join(t1,NULL);
  pthread_join(t2,NULL);
  printf("n=%d\n",n);
}
```

Sans le mutex

```
$ ./a.out
n=15684
```

L'incrémentation n'est pas atomique !

Avec le mutex

```
$ ./a.out
n=20000
```

Évidemment, cela a un coût (reduit le parallèlisme)

gcc accepte l'option `-fsanitize=thread` :

```
WARNING: ThreadSanitizer: data race (pid=18400)
  Read of size 4 at 0x7ffec19a9e14 by thread T2:
    #0 f <null> (a.out+0x120f)

  Previous write of size 4 at 0x7ffec19a9e14 by thread T1:
    #0 f <null> (a.out+0x1224)

  Location is stack of main thread.

  Location is global '<null>' at 0x000000000000 ([stack]+0x00000001fe14)

  Thread T2 (tid=18403, running) created by main thread at:
    #0 pthread_create /build/gcc/src/gcc/libsanitizer/tsan/tsan_interceptors.cc:964 (lib
    #1 main <null> (a.out+0x12ca)

  Thread T1 (tid=18402, finished) created by main thread at:
    #0 pthread_create /build/gcc/src/gcc/libsanitizer/tsan/tsan_interceptors.cc:964 (lib
    #1 main <null> (a.out+0x12ab)

SUMMARY: ThreadSanitizer: data race (/home/denis/Enseignements/IUT/FA2/2019-2020/AS3/thr
==================
```

Sémaphores Posix

Les sémaphores POSIX peuvent être nommés ou non nommés.

Un sémaphore non nommé n'est accessible que par sa position en mémoire. Il permet de synchroniser des threads, qui partagent par définition le même espace.

Initialisation

```c
#include <semaphore.h>

int sem_init(sem_t *sem, int pshared, unsigned int value);

sem_t s;
sem_init(&s,0,3);
```

Opération `P()`

```
int sem_wait(sem_t *sem);

sem_wait(&s);
```

Opération `V()`

```
int sem_post(sem_t *sem);

sem_post(&s);
```

Destruction

```
int sem_destroy(sem_t *sem);

sem_destroy(&s);
```

Moniteurs : Conditions

Condition

Possibilité d'attente passive qu'une condition soit réalisé pour continuer.

- Un thread prend le mutex pour vérifier une condition `C`.
- La condition n'est pas réalisée $\Rightarrow$ Le mutex est libéré et le thread passe en attente.
- La condition est satisfaite $\Rightarrow$ on notifie les (un) threads en attente que la condition est satisfaite. Les (le) threads notifiés reprennent le mutex (ils sont en compétition) et réessayent.

Initilisation/destruction

```
int pthread_cond_init(pthread_cond_t *restrict cond,
    const pthread_condattr_t *restrict attr);
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

Le deuxième argument peut-être `NULL` par défaut.

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

Attention : aucun thread ne doit être en attende sur `cond`.

En cas de succès, renvoient 0, sinon un code d'erreur non-nul.

Attente

```
int pthread_cond_wait(pthread_cond_t *restrict cond,
    pthread_mutex_t *restrict mutex);
```

Attend sur la variable de condition.

Attention : le mutex doit être vérouillé par le thread appelant.

- déverrouille `mutex`.
- passe en attente jusqu'à une notification sur `cond`.
- une fois la notification reçu, essaye de reprendre le mutex.
- une fois `mutex` repris, retourne 0.

Notification

```
int pthread_cond_signal(pthread_cond_t *cond);
```

Notifie un thread en attente sur `cond`

```
int pthread_cond_broadcast(pthread_cond_t *cond);
```

Notifie tous les threads en attente sur `cond`.

Renvoient 0 si succès, sinon un code d'erreur non-nul.

Exemple : un rendez-vous

```
pthread_mutex_t m;
pthread_cond_t c;
int nThreads=0;

void rendezvous(){
  pthread_mutex_lock(&m);
  nThreads ++;
  while (nThreads != N) phread_cond_wait(&c,&m);
  pthread_cond_broadcast(&c); // reveil tout le monde
  pthread_mutex_unlock(&m);
}
```

Allocateur

```
pthread_cond_t c=PTHREAD_COND_INITIALIZER;
pthread_cond_m m=PTHREAD_MUTEX_INITIALIZER;
int rc=10;
void allouer(int n){
  pthread_mutex_lock(&m);
  while (rc<n) pthread_cond_wait(&c,&m);
  rc-=n;
  pthread_mutex_unlock(&m);
}
void liberer(int n){
  pthread_mutex_lock(&m);
  rc+=n;
  pthread_cond_broadcast(&c); // pas de strategie !
  pthread_mutex_unlock(&m);
}
```

Producteurs/Consommateurs

Des producteurs et des consommateurs partagent l'accès à un buffer.

```
message buffer[N];

pthread_mutex_t m;
pthread_cond_t c_empty,  c_full;

void function deposer(message mes){
  pthread_mutex_lock(&m);
  while (is_full(buffer))
    pthread_cond_wait(&c_empty,&m);
  put(mes);
  pthread_cond_signal(&c_full);
  pthread_mutex_unlock(&m);
}
```

```
message function prendre(){
  message mes;
  pthread_mutex_lock(&m);
  while (is_empty(buffer))
    pthread_cond_wait(&c_full,&m);
  mess = get();
  pthread_cond_signal(&c_empty);
  pthread_mutex_unlock(&m);
  return mes;
}
```

Lecteurs/Rédacteurs

Rappels :

- plusieurs lecteurs peuvent lire la ressource,
- les rédacteurs s'excluent mutuellement,
- les lecteurs et rédacteurs s'excluent mutuellement.

Algorithme :

Lecture
s'il y a un rédacteur alors attendre
sinon passer
Fin lecture
réveiller les rédacteurs en attente
que s'il est le dernier à sortir.

Ecriture
s'il y a un lecteur alors attendre.
s'il y a un rédacteur alors attendre.
sinon passer.
Fin écriture
réveil soit des lecteurs en attente,
soit d'un autre rédacteur.

Variables entières :

- `nb_lect` entier qui compte les lecteurs,
- `nb_rect` entier qui compte les redacteurs.

Mutex et conditions :

- un mutex `m`,
- une condition `c`.

```c
void demander_lecture(){

  pthread_mutex_lock(&m);

  while (nb_red != 0)
    pthread_cond_wait(&c,&m);

  nb_lect ++ ;

  pthread_mutex_unlock(&m);
}

void fin_lecture(){

  pthread_mutex_lock(&m);

  nb_lect -- ;
  if (nb_lect == 0)
    pthread_cond_signal(&c);

  pthread_mutex_unlock(&m);
}
```

```c
void demander_ecriture(){

  pthread_mutex_lock(&m);

  while (nb_red != 0 || nb_lect != 0)
    pthread_cond_wait(&c,&m);

  nb_red ++ ;

  pthread_mutex_unlock(&m);
}

void fin_ecriture(){

  pthread_mutex_lock(&m);

  nb_red -- ;
  pthread_cond_broadcast(&c);

  pthread_mutex_unlock(&m);
}
```

Aucune priorité !