EPU - Informatique ROB4
Informatique Système
Pointeurs de fonction et threads

Sovannara Hak, Jean-Baptiste Mouret
hak@isir.upmc.fr

Université Pierre et Marie Curie
Institut des Systèmes Intelligents et de Robotique (CNRS UMR 7222)

2014-2015

Plan de ce cours

Rappel du plan

↦ **Bon à savoir ...**

Règles de fonctionnement

- Les cours et les TPs sont obligatoires.
- Tout(e) étudiant(e) arrivant en retard pourra se voir refuser l'accès à la salle de cours ou à la salle de TP.
- La fonction `man` du shell doit, tant que faire se peut, être utilisée avant de poser une question par email ou en TP.

References

- `https://computing.llnl.gov/` : nice tutorials on PThreads et Parallel Computing
- A. Tannenbaum & A. Woodhull. **Operating Systems : design and implementation**. 3rd ed. Prentice Hall, 2006.

Rappel du plan

Définition

- Un programme en cours d'exécution (processus) occupe de l'espace dans la mémoire centrale.
- Le code de l'exécutable et les variables utilisées sont placés dans la mémoire et constituent une partie de ce que nous avons appelé le processus.
- Ainsi chaque instruction ou fonction du code exécutable est stockée en mémoire et possède donc une adresse.
- Un pointeur de fonction représentent, au même titre qu'un pointeur sur variable, une adresse en mémoire : celle d'une fonction.
- La désignation d'une fonction par son adresse (i.e. par un pointeur) apporte un élément de flexibilité supplémentaire dans la conception du code d'un programme.

Déclaration

- Un pointeur de fonction est une variable et est déclaré en tant que telle.
- La syntaxe de déclaration se rapproche de celle d'un prototype de fonction, le nom de la fonction étant placé entre parenthèse et précédé d'une *.
- Déclaration et initialisation d'un pointeur pour une fonction retournant un entier et prenant comme arguments deux entiers :

```
int (*pf_get_int)(int ,int ) = NULL;
```

Rappel du plan

Affectation, Appel

- Affectation de l'adresse d'une fonction à un pointeur (l'opérateur d'adressage & est optionnel) :

```
/* la fonction doit exister et avoir un prototype identique à celui du pointeur */
pf_get_int = &saisie_entier_borne;
```

- Appel d'une fonction via un pointeur (l'opérateur de déréférencement * est optionnel) :

```
int a = (*pf_get_int) (0,100);
```

Un example simple : **** ex0_pf.c ****

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

void print1(int val){ fprintf(stdout,"Num = %d\n",val);}
void print2(int val){ fprintf(stdout,"N := %d\n",val);}

int main()
{
        void (*pf) (int) = NULL;
        int val = 5;

        fprintf(stdout,"pf points to : %p\n",pf);

        pf = &print1;
        fprintf(stdout,"pf points to : %p\n",pf);
        (*pf) (val);

        pf = &print2;
        fprintf(stdout,"pf points to : %p\n",pf);
        (*pf) (val);

        return 0;
}
```

Output :

```
$ ./ex0
pf points to : (nil)
pf points to : 0x400564
Num = 5
pf points to : 0x400590
N := 5
```

**** ex1_pf.c ****

Rappel du plan

Pointeur de fonction en argument d'une fonction

Un pointeur de fonction peut être passé comme argument d'une fonction `f()` dans deux contextes différents :

- en tant qu'adresse "banalisée" dans la mémoire, le type de l'argument sera alors `void *`.
- en tant qu'adresse d'une fonction à utiliser dans la fonction `f()`. Il est alors nécessaire de passer à la fonction `f()` les arguments de la fonction à appeler.

```
/* Prototype de la fonction f() */
void affiche_adresse(void *p);
int saisie_entier_gen(int min, int max, int (*pf) (int, int));
...
/* Appel */
affiche_adresse((void *)pf_get_int);
int a = saisie_entier_gen(0, 100, pf_get_int);
...
/* Appel, méthode alternative */
int a = saisie_entier_gen(0, 100, &saisie_entier_borne);
```

**** ex2_pf.c ****

Pointeur de fonction en valeur de retour d'une fonction

Un pointeur de fonction peut être passé comme valeur de retour d'une fonction `f()` dans deux contextes différents :

- en tant qu'adresse "banalisée" dans la mémoire, le type de retour de la fonction sera alors `void *`.
- en tant qu'adresse d'une fonction et la syntaxe devient alors un peu obscure.

```
/* Prototype de la fonction */
int (* choix_saisie_entier_gen(char c)) (int,int) ;
...
/* Retour */
return &saisie_entier_borne_abs;
...
/* Appel */
pf_get_int = choix_saisie_entier('a');
```

**** ex3_pf.c ****

Lisibilité : définition de types spécifiques aux pointeurs de fonction

Pour rendre le code plus lisible on peut définir un type spécifique avec `typedef`.

```
typedef int (*pf_saisie) (int, int);
...
/* Prototype de la fonction */
pf_saisie choix_saisie_entier_gen(char c);
...
/* Appel */
pf_get_int = choix_saisie_entier('a');
```

**** ex4_pf.c ****

Rappel du plan

Notion de généricité

- Une fonction est dite générique si son implémentation est indépendante du type et, éventuellement, du nombre de ses arguments.
- Par exemple, l'algorithmique du tri est indépendante du type de choses à trier et une fonction générique de tri doit pouvoir prendre comme argument un tableau à trier indépendamment du type de données qu'il contient. La seule fonction qui soit alors dépendante du type est la fonction de comparaison.
- Dans le même ordre d'idée, les structures de type *arbre, pile, table de hachage ou graphes* et les algorithmes associés (*parcours, recherche, insertion,...*) existent indépendamment du type qu'elles contiennent.

Pointeur de fonctions et généricité

- La fonction `int saisie_entier_gen(int min, int max, int (*pf) (int, int))` peut permettre une certaine souplesse de programmation : la fonction de saisie à effectivement utiliser est une variable et peut donc être défini dynamiquement (par l'utilisateur du programme, via un fichier, en fonction du contexte,...).
- Cependant cette fonction n'est pas générique :
  - elle ne permet pas de saisir autres choses que des entiers ;
  - elle prend forcément deux arguments d'entrées de type entier ;
  - elle ne peut retourner qu'un entier.
- Afin de la rendre générique, son prototype doit être :

```
void* saisie_gen(void* (*f) (void* arg), void* arg);
```

- Cela signifie :
  - que `saisie_gen()` retourne forcément un pointeur (ou, en trichant un peu, un entier puisqu'un pointeur a une valeur entiere) ;
  - que la fonction `f()` doit elle même retourner un pointeur (idem) ;
  - que les arguments à passer à la fonction `f()` doivent être passés sous la forme d'un seul pointeur (sur une structure si les arguments sont de types hétérogènes) ;
  - que la fonction `f()` devra commencer par une récupération des arguments via un cast de `void*` vers autre chose et se terminer par un cast pour retourner qqch de type `void *`

Remarques

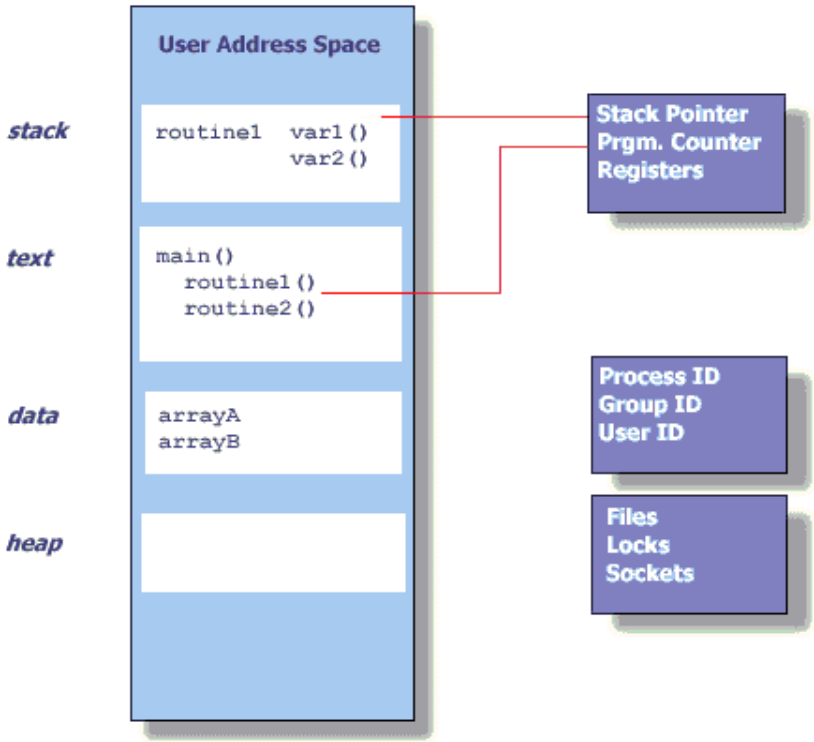
Les pointeurs de fonction apporte une grande souplesse au code...mais sont aussi une grande source d'erreurs de syntaxe ou de bugs pas évidents à trouver.

**** ex5_pf.c ****

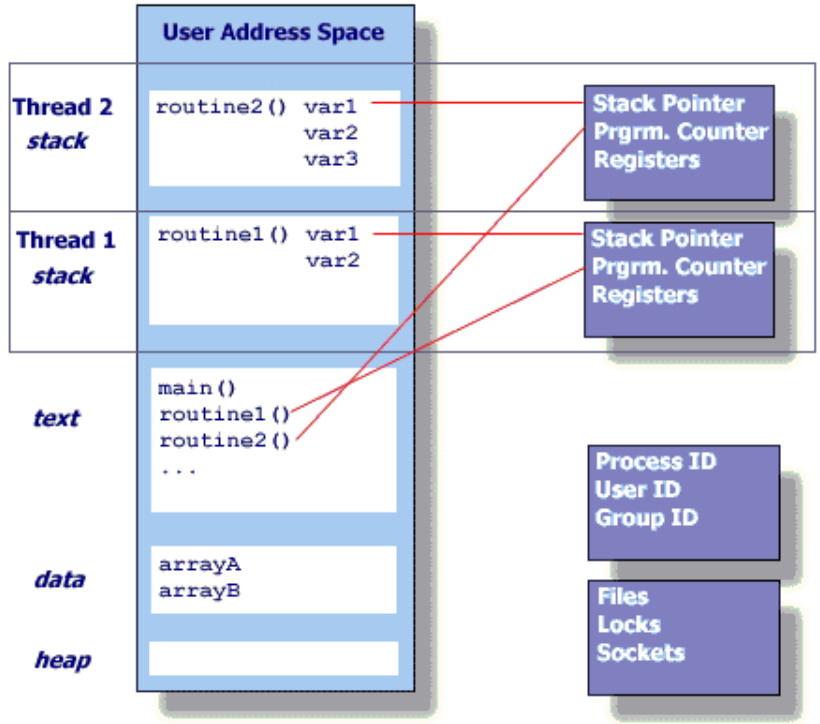
Threads

Rappel du plan

Un thread est un processus **léger**, ils partagent la mémoire virtuelle de leurs processus père.



PROCESS



THREADS WITHIN A PROCESS

Notion de processus léger

- Un processus classique (lourd) est constitué d'une zone mémoire qui lui est réservée (**espace d'adressage**) et d'un **fil d'exécution** représenté par la valeur du pointeur de programme (pointeur sur la prochaine instruction à exécuter) et la pile d'exécution (zone mémoire réservées associée aux appels de fonction).
- Un processus peut cependant être composé d'un seule espace d'adressage mais de plusieurs fils d'exécution (avec chacun leur pointeur de programme et leur pile). Ces fils d'exécution sont alors appelés **processus légers** ou encore **threads**.

Avantages et inconvénients des threads

+ Un des avantages des processus légers est un allègement des opérations de commutation de contexte : lorsque le processeur commute d'un thread à un autre appartenant au même processus, le contexte mémoire reste le même et seul la pile et le compteur de programme sont à changer.
+ Contrairement à un fork, **l'opération de création d'un thread ne nécessite pas la copie complète de l'espace d'adressage du père**.
± Le même espace d'adressage étant partagé par les threads au sein d'un même processus, l'accès aux ressources se fait de manière concourrante et peut donc poser problème (par exemple, les variables globables sont partagées). Ceci étant dit ce partage peut permettre d'alléger la communication inter-processus.

- Threads sont des "**processus léger**"
  - La création d'un processus nécessite une certaine charge supplémentaire ("overhead") pour l'OS, qui va devoir enregistrer les informations à propos des ressources, de l'état du processus
  - Process ID, group ID, user ID ; environement ; répertoire courant ; instructions programme ; registres, pile, tas, descripteurs de fichiers ; donnée de communication inter-processus (pipes, memoire partagée, etc.) ; ...
- Threads **existent dans le processus et utilisent les resources de ce dernier**..
  - .. mais ils peuvent être ordonnancé et exécuté par l'OS de manière indépendante ..
  - .. car ils dupliquent uniquement les ressources dont ils ont besoin pour exister en tant que code exécutable et être ordonnançable indépendamment du processus.
- Threads ont leurs **propres déroulement**
  - un thread maintient son propre pointeur de pile, registres, etc.
  - tant que son parent existe et que l'OS le supporte, il meurt quand son père meurt
- Threads **partagent ses ressources avec leurs processus père**
  - changement effectué par un thread sur une ressource partagée (fermer un fichier par exemple) est vu par les autres threads
  - comme ils peuvent lire/écrire aux mêmes endroits de la mémoire, il faut une **synchronisation** explicite

Rappel du plan

Fork vs Thread

- **Fork**
  - Créer un nouveau processus (enfant), qui est une copie du processus courant (père)
  - Le même code exécutable
  - Chaque processus a son propre PID
  - Même image mémoire à la création (mais ça peut changer lors de l'exécution)
  - Pendant l'exécution, l'enfant ne partage pas de mémoire avec le père (ils peuvent communiquer en utilisant desprimitives Inter Process Communication (IPC))
  - Overhead à la création et la terminaison
  - Facile à debugger
- **Thread**
  - Un autre fil d'exécution pour le même processus, une entité dans le processus qui peut être ordonancé individuellement
  - Chaque thread a son propre Thread ID
  - Chaque thread a sa propre pile (variables locales...)
  - Les threads d'un même processus partagent espace d'adressage, descripteurs de fichiers (fichiers ouvert), signaux, répertoire courant (au lieu de tout recréer)
  - Les données partagées doivent être accédé via des primitives de synchronisation, pour éviter des corruptions ou comportements étranges.
    - Plus rapides IPC
    - Accès concurrent : il faut utiliser des mutex et des joins pour obtenir un ordre et un résultat prévisible et sûr
  - Rapide à démarrer et terminer
  - Plus rapide sur un CPU multi-threadé car il n'y a pas de changement de contexte au niveau du processus

Fork vs Thread

Une comparaison intéressante : création de 50 000 processus/threads, unité en secondes, pas de flags d'optimisation. (source : `https://computing.llnl.gov/tutorials/pthreads/`)

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

**** performanceFork.c ****

```
/*
from: https://computing.llnl.gov/tutorials/pthreads/
================================================================================
C Code for fork() creation test
================================================================================
*/
#include <stdio.h>
#include <stdlib.h>
#define NFORKS 50000

void do_nothing() {int i;i= 0;}

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

for (j=0; j<NFORKS; j++) {

  /*** error handling ***/
  if ((pid = fork()) < 0 ) {
    printf ("fork failed with error code= %d\n", pid);
    exit(0);
    }

  /*** this is the child of the fork ***/
  else if (pid ==0) {
    do_nothing();
    exit(0);
    }

  /*** this is the parent of the fork ***/
  else {
    waitpid(pid, status, 0);
    }
  }
}
```

**** performanceThread.c ****

```c
/*
from: https://computing.llnl.gov/tutorials/pthreads/
================================================================================
C Code for pthread_create() test
================================================================================
*/
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define NTHREADS 50000

void *do_nothing(void *null) {int i;i=0; pthread_exit(NULL);}

int main(int argc, char *argv[]) {
int rc, i, j, detachstate;
pthread_t tid;
pthread_attr_t attr;

pthread_attr_init(&attr);
pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);

for (j=0; j<NTHREADS; j++) {
  rc = pthread_create(&tid, &attr, do_nothing, NULL);
  if (rc) {
    printf("ERROR; return code from pthread_create() is %d\n", rc);
    exit(-1);
    }

  /* Wait for the thread */
  rc = pthread_join(tid, NULL);
  if (rc) {
    printf("ERROR; return code from pthread_join() is %d\n", rc);
    exit(-1);
    }
  }

pthread_attr_destroy(&attr);
pthread_exit(NULL);
}
```

Rappel du plan

Utilisation des threads

Les threads sont souvent utilisés dans les contextes suivants :

- Séparation du processus de calcul et du processus lié à l'interface graphique d'un même programme : on peut continuer à interagir avec l'interface graphique même dans le cas d'un calcul long.
- Utilisation de la puissance de calcul d'ordinateurs multi-coeurs. Pour des calculs longs, il peut être intéressant de paralléliser les sous-calculs au moyen de threads, chacun de ces processus légers pouvant être alors traité par un processeur différent et en parallèle d'autres calculs.

Remarques

- Le temps de création d'un thread n'étant pas nul, la parallélisation n'est viable que pour des calculs "assez" longs.
- La parallélisation ne présente aucun intérêt dans le cas d'ordinateurs à processeur unique.

Rappel du plan

Posix Threads - IEEE POSIX 1003.1 (1995 - .)

- Pourquoi une API standard ?
  - Les constructeurs implémentaient leurs propres version de threads
  - Difficile d'avoir un code portable
- http://posixcertified.ieee.org/
- IEEE and The Open Group
- POSIX = Portable Operating System Interface
- Sur les système Linux il y a la librairie Native Posix Threads Library (NPTL)

Généralités

- La librairie de gestion des threads NPTL.
- Elle nécessite l'inclusion de `<pthread.h>` dans les fichiers sources et le lien avec `libpthread.a` lors de l'édition de liens.

```
$ gcc -o mon_prog mon_prog.o -lpthread
```

- Chaque thread est identifié de manière unique par une identifiant de type `pthread_t` (soit un entier long non signé soit un type structuré).
- Cet identifiant joue un rôle analogue au PID d'un processus ?
- La fonction `pthread_self()` renvoie une valeur de type `pthread_t` et permet à chaque thread de connaitre son propre identifiant.

POSIX threads

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

Creation / termination

```
int   pthread_create
   (pthread_t *, const pthread_attr_t *, void *(*)(void *), void *);
void  pthread_exit(void *);
```

Detach (storage for the thread can be reclaimed when that thread terminates) / Cancel execution / Wait for thread termination

```
int   pthread_detach(pthread_t);
int   pthread_cancel(pthread_t);
int   pthread_join(pthread_t thread, void **);
```

Lock/unlock mutex

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

Création d'un thread utilisateur

- La création d'un nouveau thread au sein d'un processus se fait via la fonction :
  `int pthread_create(pthread_t* thread,pthread_attr_t* attr,void* (*pf)(void*),void* arg);`
- Cette fonction retourne un entier : 0 en cas de succès et sinon une valeur négative correspondant à l'erreur survenue.
- Le premier argument permet au père de récupérer l'identifiant `*thread` du thread créé.
- Le second argument correspond aux attributs associés au thread au moment de sa création.
- Pour hériter des attributs standards, le second argument est souvent mis à `NULL`.
- Le troisième argument indique un pointeur de fonction `pf` sur la fonction à exécuter dans le thread créé.
- Le quatrième argument est un pointeur banalisé sur les arguments passé à la fonction pointée par `pf`.

Terminaison d'un thread

- La fonction `pthread_exit(void* retour)` met fin au thread qui l'exécute (analogue à `exit()`)
- Le thread peut retourner une valeur au travers de l'argument `retour`.
- Cette valeur de retour peut-être récupérée par un autre thread (le père par exemple) au moyen de la fonction `pthread_join(pthread_t thread, void** ret)` (analogue à `waitpid()`).
- Le premier argument de `pthread_join()` correspond à l'identifiant du thread attendu et `ret` permet de récupérer la valeur `retour` passée à `pthread_exit()`.
- L'appel à `pthread_join()` est bloquant pour le processus qui l'exécute et suspend son exécution jusqu'à ce que le thread attendu soit achevé.

Remarque à propos de la valeur de retour d'un thread

- Les fonctions `pthread_exit()` et `pthread_join()` permettent le passage d'une valeur de retour.
- Ceci étant dit, ce résultat peut aussi être passé à un des arguments de la fonction exécuté par le thread ce qui est une méthode tout aussi générique.

Attributs d'un thread

- L'adresse de départ et la taille de la pile associée ;
- La politique d'ordonancement associée (FIFO, tourniquet (*round robin* en anglais), par priorité) ;
- La priorité qui lui est associée ;
- son attachement ou son détachement : un thread détaché se termine immédiatement sans pouvoir être pris en compte par `pthread_join()` (pas de sauvegarde du retour).

Modification de la valeur des attributs

- Si les attributs par défaut ne sont pas choisis au moment de la création du thread, il faut initialiser une variable de type `pthread_attr_t` en utilisant la fonction `pthread_attr_init()` puis modifier la valeur des attributes en ayant recours aux fonctions `pthread_attr_getXXX()` et `pthread_attr_setXXX()` où `XXX` est l'attribut à modifier.
- Un thread peut être détaché après sa création en utilisant la fonction `int pthread_detach(pthread_t th);`.
- Dans le cadre de ce cours, les attributs par défaut seront utilisés.

Rappel du plan

**** ex0_t.c ****

```
/* Creating threads and printing their info */

#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define NUM_THREADS     10

void *InfoThread(void *threadid)
{
   pthread_t tid = (pthread_t)threadid; //cast
   printf("Thread n= %ld\n", tid);
   pthread_exit(NULL);
}

int main(int argc, char *argv[])
{
   pthread_t threads[NUM_THREADS];
   int err; long t;

   for(t=0;t<NUM_THREADS;t++)
   {
        printf("Main creates thread %ld\n", t);
        err = pthread_create(&threads[t], NULL, InfoThread, (void *)t); //cast
        if(err) { printf("ERROR; return code :%d\n", err);       exit(-1); }
        }

   // main must do this at the end
   pthread_exit(NULL);
}
```

****** ex0_t.c ******

- Le processus père va créer plusieurs enfants threads
- Les threads impriment leurs identifiants

Comment compiler et générer l'exécutable ?

****** ex0_t.c ******

- Le processus père va créer plusieurs enfants threads
- Les threads impriment leurs identifiants

Comment compiler et générer l'exécutable ?

```
$ gcc -g -Wall ex0_t.c -lpthread -o ex0_t
```

Sortie ?

****** ex0_t.c ******

- Le processus père va créer plusieurs enfants threads
- Les threads impriment leurs identifiants

Comment compiler et générer l'exécutable ?

```
$ gcc -g -Wall ex0_t.c -lpthread -o ex0_t
```

Sortie ?

```
$ ./ex0_t
Main creates thread 0
Main creates thread 1
Main creates thread 2
Thread n= 2
Thread n= 0
Main creates thread 3
Main creates thread 4
Thread n= 3
Thread n= 4
Main creates thread 5
Main creates thread 6
Thread n= 5
Main creates thread 7
Thread n= 1
Thread n= 6
Main creates thread 8
Thread n= 7
Main creates thread 9
Thread n= 8
Thread n= 9
```

Est-ce la seule sortie possible ?

Est-ce la seule sortie possible ?

Non. Une autre sortie possible :

```
$ ./ex0_t
Main creates thread 0
Main creates thread 1
Thread n= 0
Main creates thread 2
Thread n= 1
Main creates thread 3
Thread n= 2
Main creates thread 4
Thread n= 3
Main creates thread 5
Thread n= 4
Main creates thread 6
Thread n= 5
Main creates thread 7
Thread n= 6
Main creates thread 8
Thread n= 7
Main creates thread 9
Thread n= 8
Thread n= 9
```

Pourquoi ? ?

Attention !

- Après la création d'un thread, par défaut, c'est l'OS qui l'ordonance
- À moins d'utiliser explicitement des mécanismes d'ordonancement explicites de Pthreads, c'est l'OS qui décide du moment de l'exécution d'un thread

- **Des programmes robuste ne doivent pas se reposer sur l'exécution de thread dans un ordre particulier ou un processeur/coeur particulier !**

Rappel du plan

****** ex1_t.c ******

- Un processus père crée un thread enfant
- Une variable globale de type entier est une variable partagée : `int i;`
- Père (main) fait : `i += 1000; ... i += 2000;`
- Enfant (thread) fait : `i += 10; ... i += 20;`

Output ?

****** ex1_t.c ******

- Un processus père crée un thread enfant
- Une variable globale de type entier est une variable partagée : `int i;`
- Père (main) fait : `i += 1000; ... i += 2000;`
- Enfant (thread) fait : `i += 10; ... i += 20;`

Output ?

```
$ ./ex1_t
Hello, ici thread père PID 1973. Pour info, i =  1000
Hello, ici thread père PID 1973. Pour info, i =  3000
Hello, ici thread fils PID 1973. Pour info, i =  3010
Hello, ici thread fils PID 1973. Pour info, i =  3030
Ca tombe bien !
```

Seule sortie possible ?

Seule sortie possible ?

Non. Autre sortie possible :

```
$ ./ex1_t
Hello, ici thread père PID 1988. Pour info, i =  1000
Hello, ici thread fils PID 1988. Pour info, i =  1010
Hello, ici thread père PID 1988. Pour info, i =  3010
Hello, ici thread fils PID 1988. Pour info, i =  3030
Ca tombe bien !
```

And this one too :

```
icub@eva:~/Desktop/InfoSys_2012/trunk/cours/c3/code/thread$ ./ex1_t
Hello, ici thread père PID 2050. Pour info, i =  1000
Hello, ici thread fils PID 2050. Pour info, i =  1010
Hello, ici thread fils PID 2050. Pour info, i =  3030
Hello, ici thread père PID 2050. Pour info, i =  3010
Ca tombe bien !
```

Pourquoi ? ? Qu'est-ce qui ne va pas dans le code ?

Seule sortie possible ?

Non. Autre sortie possible :

```
$ ./ex1_t
Hello, ici thread père PID 1988. Pour info, i =  1000
Hello, ici thread fils PID 1988. Pour info, i =  1010
Hello, ici thread père PID 1988. Pour info, i =  3010
Hello, ici thread fils PID 1988. Pour info, i =  3030
Ca tombe bien !
```

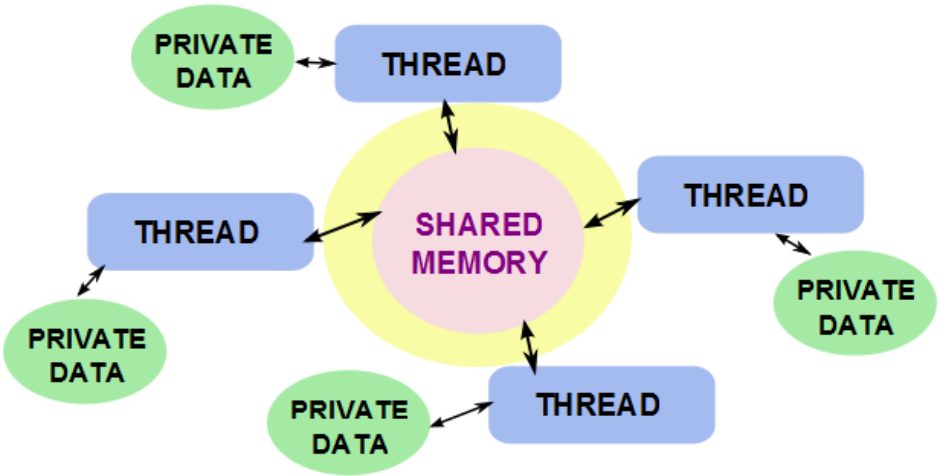
And this one too :

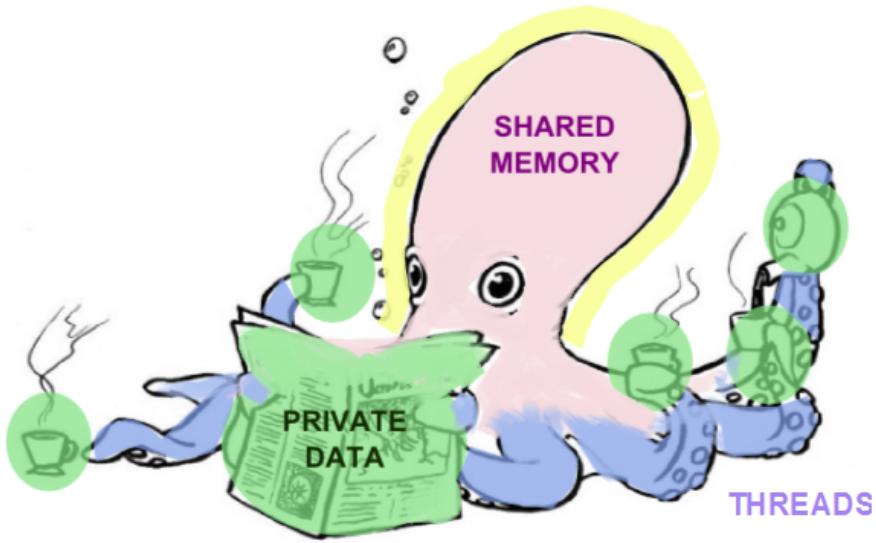
```
icub@eva:~/Desktop/InfoSys_2012/trunk/cours/c3/code/thread$ ./ex1_t
Hello, ici thread père PID 2050. Pour info, i =  1000
Hello, ici thread fils PID 2050. Pour info, i =  1010
Hello, ici thread fils PID 2050. Pour info, i =  3030
Hello, ici thread père PID 2050. Pour info, i =  3010
Ca tombe bien !
```

Pourquoi ? ? Qu'est-ce qui ne va pas dans le code ?

- L'accès à la variable partagée `i`
- La synchronisation père/fils

Synchronisation de threads





Rappel du plan

Threads sont idéaux pour de la programmation parallèle

Il faut considérer plusieurs choses quand on fait une telle programmation par exemple :

- Problème de partitionnement : peut-on identifier des sous-routines ou tâche qui peuvent être exécuté en même temps ?
- Équilibre des charges
- Communication : pas que de la mémoire partagée !
- Synchronisation et accès concurent

Quand utiliser les threads ?

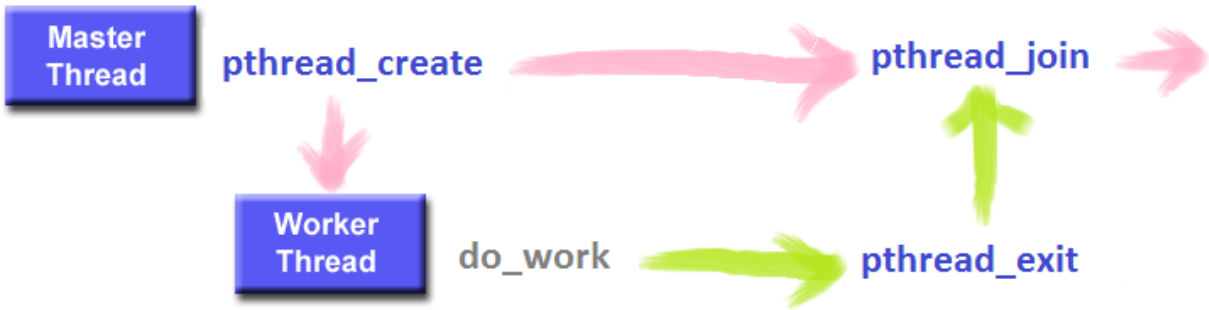
Exemple typique :

- Programmes où le travail peut être décomposé en plusieurs tâches simultané : N données traités par N tâches
- Programmes où les I/O peuvent être potentiellement bloquante (long delais, ..)

Synchronisation basique

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

- `pthread_join` suspend l'exécution du thread appelant jusqu'à ce que le thread cible termine
- renvoie immédiatement si la cible est déjà terminée
- renvoie zero lorsque la cible est terminée
- sinon, un numéro erreur



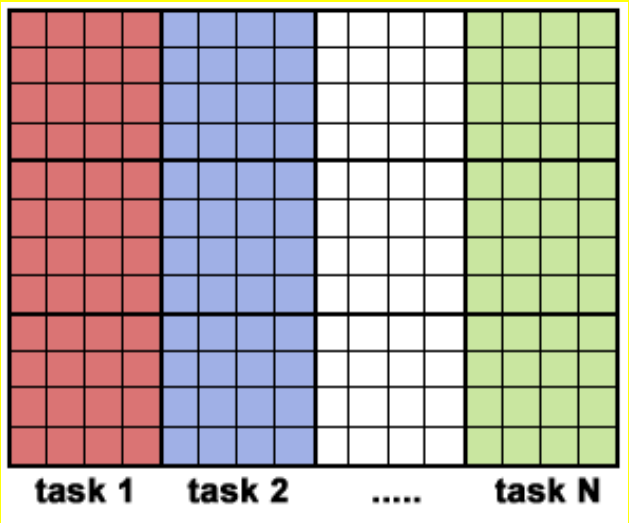
Rappel du plan

**** ex2_t.c ****

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

typedef struct
{
        int *ar;
        long n;
        int key;
} message;

// the task executed by the threads
void * cypher(void *arg)
{
   long i;
   for (i = 0; i < ((message *)arg)->n; i++)
       ((message *)arg)->ar[i]+=((message *)arg)->key;
}

int main(void)
{
   int ar[20];
   int key = 3;
   pthread_t  th1, th2;
   message m1, m2;

   m1.ar = &ar[0]; m1.n  = 10; m1.key = key;
   (void) pthread_create(&th1, NULL, cypher, &m1);

   m2.ar = &ar[10]; m2.n  = 10; m2.key = key;
   (void) pthread_create(&th2, NULL, cypher, &m2);

        // wait for the threads to be done
   (void) pthread_join(th1, NULL);
   (void) pthread_join(th2, NULL);

   return 0;
}
```

**** ex2_t.c ****

- C'est un exemple très simple d'un traitement parallèle d'un tableau
- Les portions du tableau sont traité independamment → pas besoin de gérer la communication entre les tâches
- Chaque tâche a sa propre partion de tableau



Rappel du plan

TEST

How do we compile **** ex2_t.c **** ?

Modifier le programme **** ex2_t.c **** pour synchroniser les threads tel que :

- Thread 2 démarre uniquement lorsque Thread 1 a terminé
- Voici une sortie possible :

```
$ ./ex3_t
Original message:
0-1-2-3-4-5-6-7-8-9-10-11-12-13-14-15-16-17-18-19-
After thread 1:
3-4-5-6-7-8-9-10-11-12-10-11-12-13-14-15-16-17-18-19-
After thread 2:
3-4-5-6-7-8-9-10-11-12-13-14-15-16-17-18-19-20-21-22-
```

Rappel du plan

La programmation parallèle peut être dangereuse !

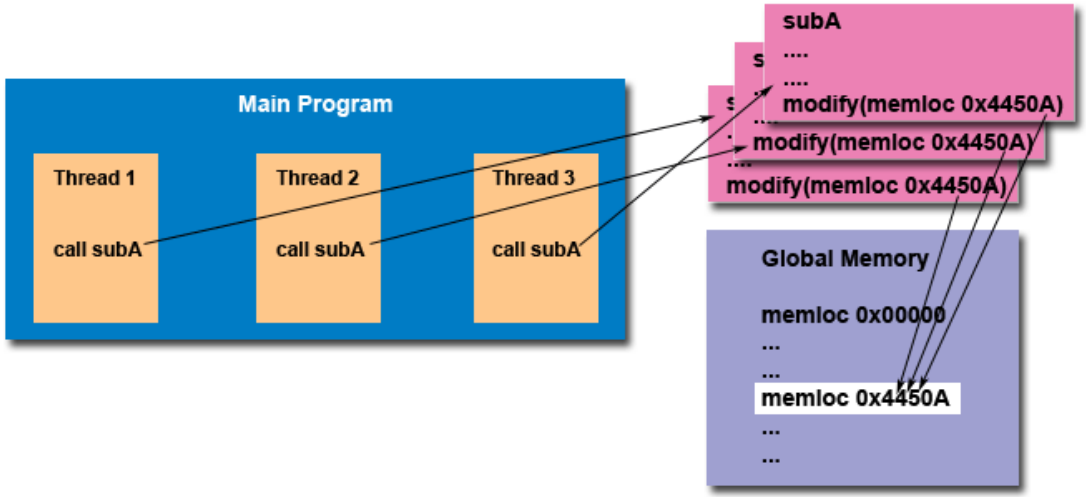
- Tout les threads accèdent aux même données globales
- Chaque thread a ses propres données privée

⇒ **programmeurs sont responsable de l'écriture de code thread-safe !**

Sécurité d'un thread

Un programme est thread-safe lorsqu'il peut exécuter plusieurs threads en même temps sans créer "d'accès concurent" ou créer des menaces potentielles à une donnée partagée

- exemple : si tout les threads doivent accéder et modifier le même endroit de la mémoire partagée il est possible que plusieurs thread le fasse en même temps
- synchronisation requise pour éviter les corruption de données



Origine du problème

- Deux threads appartenant au même processus partagent le même espace d'adressage et donc leurs variables globales.
- Cela peut poser des problèmes si ces deux threads tentent d'accéder à une mme variable simultanément. Par exemple :
  - les deux threads tentent de **modifier la variable au même moment**. Le résultat est indéterminé.
  - un des deux threads est train de lire un variable de type structuré pendant que l'autre est entrain de la modifier. La lecture peut être incohérente, la moitié de la variable seulement ayant été écrite au moment de la lecture. La valeur lue est donc incohérente.

Solution

Afin d'empêcher de tels incohérences, il est possible de protéger certaines opérations via un mécanisme d'exclusion mutuelle.

- Ce méchanisme est basé sur des variables appelés *mutex* de type `pthread_mutex_t`.
- Ces variables doivent être globales afin d'être partagés par l'ensemble des threads d'un même processus.

Rappel du plan

mutex == mutual exclusion

- Un mutex agit comme un **verrou**, protégeant l'accès aux données partagées
  - il est utilisé pour prévenir les "**accès concurent**" entre threads
  - il est utilisé généralement pour autoriser la mise à jour d'une donnée globale et **prévenir la corruption de donnée**
  - Variables mises à jour entre le vérouillage/dévérouillage appartiennent à la "**section critique**"
- seul un thread peut verouiller (ou posséder) une variable mutex à un instant donné
- les autres threads qui tentent de vérouiller lock the mutex seront bloqué jusqu'à ce que le thread maître la dévérouille
  - un thread peut faire des appels de dévérouillage non bloquant en utilisant `trylock` au lieu de `lock`

Usage typique

- Création et initialisation du mutex
- Plusieurs threads tentent de vérouiller le mutex : un seul y parvient
- le thread maître effectue ses actions sur la variable partagée protégée
- le thread maître dévérouille le mutex
- repétition : un autre thread acquiert le mutex
- le mutex est détruit

- **Le programmeur doit s'assurer que chaque thread utilise un mutex pour accéder à uen variable partagée !**

Création

Un mutex est créé comme suit :

```
pthread_mutex_t jeton = PTHREAD_MUTEX_INITIALIZER;
```

Initialisation

Un mutex est initialisé comme suit :

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

Si `attr=NULL`, par default le mutex est libre (unlocked).

Verrouillage (lock)

Un mutex est verrouillé comme suit :

```
pthread_mutex_lock(&jeton);
```

Si le mutex est déjà verrouillé, cette fonction est bloquante (jusqu'au déverrouillage du mutex en question par le thread l'ayant verrouillé).

Déverrouillage (unlock)

Un mutex est déverrouillé comme suit :

```
pthread_mutex_unlock(&jeton);
```

Rappel du plan

****** test_thread.c, basic_func.c, basic_func.h ******

- variables globales : `int glob; pthread_mutex_t my_mutex`
- struct avec les paramètres : `incStruct`
- la routine : `void *somme_inc_Ntimes(void *arg);`

Remarque :

```
//begin critical section
pthread_mutex_lock(&my_mutex);
for(i = 0; i < var->N; i++)
{
   sleep(0.5);
   *(var->res) += var->inc;
}
pthread_mutex_unlock(&my_mutex);
//end critical section
```

Dangers

1. Le verrouillage multiple (sans déverrouillage intermédiaire) dans un même thread d'un même mutex est à prohiber.
2. Le principe d'exclusion mutuelle peut être la source d'un interblocage.
   Le thread A vérouille la ressource 1 avec le mutex M1
   Le thread B lock la ressource 2 avec le mutex M2
   Le thread A demande l'accès à la ressource 2 bloquée par M2 $\rightarrow$ le thread A attend.
   Le thread B demande l'accès à la ressource 1 bloquée par M1 $\rightarrow$ le thread B attend.
   $\hookrightarrow$ Situation d'interblocage.

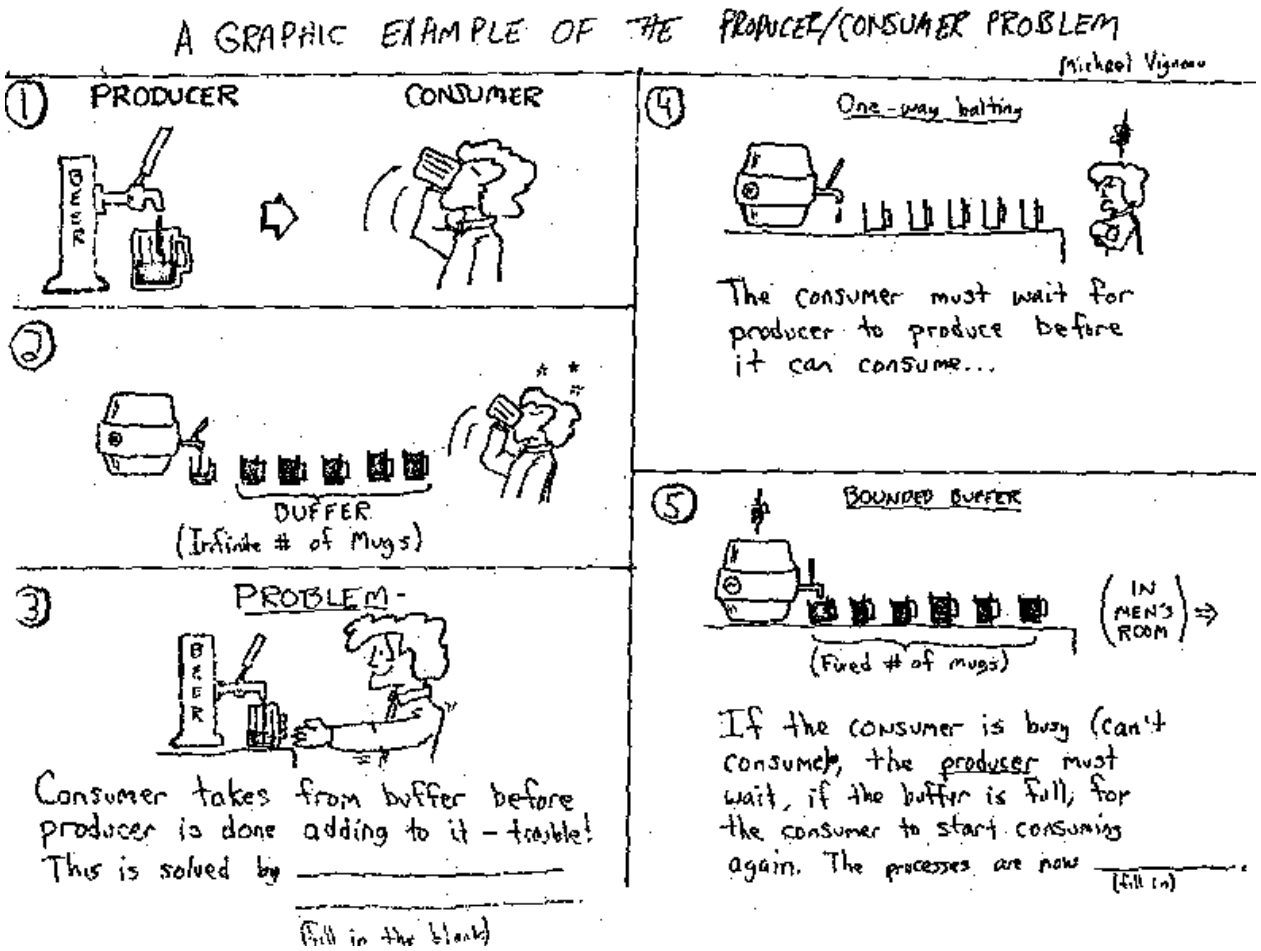
Remarque

- Les *sémaphores* généralisent la notion de mutex.
- Ils constituent un des moyens de protection de l'accès à des ressources partagées les plus utilisés.
- Ils ne sont pas traités dans le cadre de ce cours.

Problème de synchronisation classique

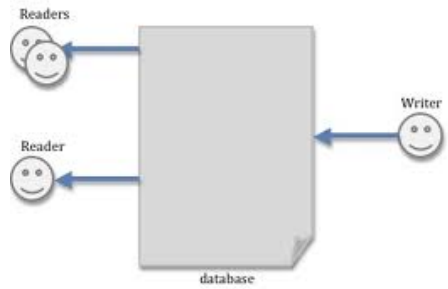
- Producteur-Consommateur (ou Bounded Buffer)
- Lecteur-Écrivain
- Dîner des philosophes

Producteur-Consommateur



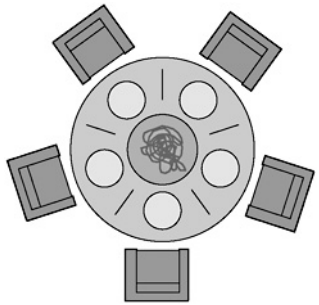
- Producteur fournit des données, consommateur la prend
  - implémentations possible : une seule donnée (cas le plus simple), buffer circulaire, ...
- Problèmes : consommateur doit attendre que le producteur fournisse de nouvelles données, impossible de prendre une donnée partielle (donnée corrompue), ...

Lecteurs-Écrivains



- Plusieurs lecteurs accèdent à une donnée (lecture seule)
  - comme ils ne la modifient pas, ils peuvent y accéder en même temps
- Un ou plusieurs écrivains modifient la donnée
- Quand un écrivain la modifie, personne d'autre ne peut y accéder
  - si un autre écrivain la modifie, la donnée peut être corrompue
  - un lecteur va pouvoir lire une donnée partiellement modifiée (valeur inconsistante)
- Différentes solutions
  - Donner une plus grande priorité au lecteurs/écrivains peut causer une famine des écrivains/lecteurs
  - Pas de priorité : tout le monde doit attendre
  - Résolu habituellement avec des sémaphores

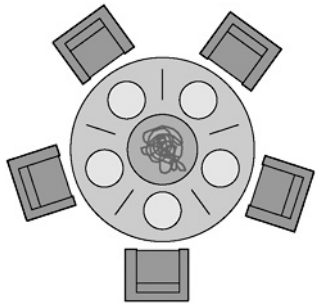
Le dîner des philosophes (Dijkstra, 1965)



- Les philosophes peuvent : manger, penser un moment, prendre la baguette de gauche/droite
- pour manger, un philosophe doit avoir la baguette de gauche et de droite
  - un philosophe ne peut pas prendre une baguette qui a déjà été prise par son voisin
  - les baguettes sont des données partagées et doivent être protégé par des mutex
- En fonction de la solution adoptée, plusieurs problèmes peuvent être rencontrées

- si tout les philosophes prennent la baguette droite au même moment ?

Le dîner des philosophes (Dijkstra, 1965)

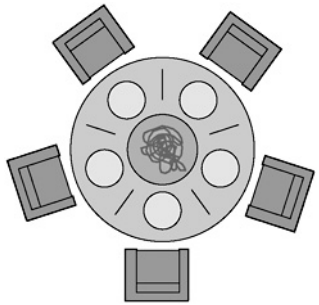


- Les philosophes peuvent : manger, penser un moment, prendre la baguette de gauche/droite
- pour manger, un philosophe doit avoir la baguette de gauche et de droite
  - un philosophe ne peut pas prendre une baguette qui a déjà été prise par son voisin
  - les baguettes sont des données partagées et doivent être protégé par des mutex
- En fonction de la solution adoptée, plusieurs problèmes peuvent être rencontrées

- si tout les philosophes prennent la baguette droite au même moment ?
  attente circulaire $\rightarrow$ **deadlock**

- si deux philosophes pensent rapidement (et donc monopolisent les baguettes et mangent tout le temps) ?

Le dîner des philosophes (Dijkstra, 1965)



- Les philosophes peuvent : manger, penser un moment, prendre la baguette de gauche/droite
- pour manger, un philosophe doit avoir la baguette de gauche et de droite
  - un philosophe ne peut pas prendre une baguette qui a déjà été prise par son voisin
  - les baguettes sont des données partagées et doivent être protégé par des mutex
- En fonction de la solution adoptée, plusieurs problèmes peuvent être rencontrées

- si tout les philosophes prennent la baguette droite au même moment ?
  attente circulaire $\rightarrow$ **deadlock**

- si deux philosophes pensent rapidement (et donc monopolisent les baguettes et mangent tout le temps) ?
  les autres philosophes n'ont pas l'occasion de manger car la ressource partagée est toujours occupée $\rightarrow$ **famine**

Rappel du plan

TEST : Producteur & Consommateur de messages

- Considérons les deux processus légers suivants :
  - P1 : producteur de messages
  - P2 : consommateur de messages

  Les deux threads peuvent communiquer par un tampon de capacité égale à un message, qui en fait est un objet partagé (une variable globale). P1 construit un message, le dépose et attend que P2 ait consommé le message. Chaque message produit doit être consommé et il est prélevé une et une seule fois.
- Le but de cet exercice est de contrôler l'accès au tampon partagé en utilisant des primitives de synchronisation entre les deux processus, qui utilisent des mutex.
- Il faut contrôler les deux cas suivants :
  - si le consommateur exécute sa séquence avant le producteur, il va trouver le tampon libre (x vide) : il faut éviter qu'il prélève un message qui n'a jamais été déposé ;
  - si le consommateur est lent, il faut éviter que le producteur écrase plusieurs messages avant que le consommateur ait consommé (il écrit plusieurs fois sur x)

Usage du programme :

```
$ ./mutex a b N t1 t2
```

But du programme :

- créer les deux threads P1 et P2
  - P1 : producteur de messages
  - P2 : consommateur de messages
- les threads exécutent N fois les même opérations
  - P1 calcule $x = r + a$, où r est un nombre aléatoire, appelle `sleep(t1)` et affiche x ;
  - P2 calcule $x = x + b$, appelle `sleep(t2)` et affiche x.

Par exemple, l'utilisation et la sortie du programme doivent correspondre à :

```
$ ./mutex 1 2 2 3 5
P1: r=3 x=3+1=4
P2: x=4+2=6
P1: r=65 x=65+1=66
P2: x=66+2=68
```

Hint : votre programme peut notamment utiliser les fonctions suivantes : `atoi`,`atof`,`sleep`, `pthread_create`,`pthread_join`, `pthread_mutex_lock`,`pthread_mutex_unlock`.

Code :

Questions ?