Programmation système II

Threads, synchronisation, IPC, signaux, mémoire partagée

Timothée Ravier

LIFO, INSA-CVL, LIPN

1re année cycle ingénieur STI
2013 – 2014

Plan global

1 Threads

2 Synchronisation

3 Tubes

4 Signaux

5 Files de messages

6 Mémoire partagée

Rappels : pages de manuel

- `man man` ;
- `man -k <fonction>` : cherche un manuel mentionnant la fonction ;
- `man -f <fonction>` : liste toutes les versions d'un manuel associé à une fonction ;
- Notation : `function(2)` $\Leftrightarrow$ `man 2 fonction` ;
- Attention aux différences entre Linux et POSIX !
- Livre de Michael Kerrisk (Linux Programming Interface) et site web (http://man7.org).

Rappels : général

- Appel système ;

Rappels : général

- Appel système ;
- (g)libc ;

Rappels : général

- Appel système ;
- (g)libc ;
- Environnement ;

Rappels : général

- Appel système ;
- (g)libc ;
- Environnement ;
- Descripteur de fichier ;

Rappels : général

- Appel système ;
- (g)libc ;
- Environnement ;
- Descripteur de fichier ;
- Everything is a file ?

Rappels : général

- Appel système ;
- (g)libc ;
- Environnement ;
- Descripteur de fichier ;
- Everything is a file ?
- Everything is a file descriptor ?

Rappels : composition d'un processus

- Zones mémoire (pile d'exécution, tas, copie du code du programme) ;
- Liste des descripteurs de fichiers ouverts ;
- Identifiants (PID, PPID...) et attributs de sécurité (UID, GID...) ;
- État du processus (registres...) ;
- Sous Linux : `include/linux/sched.h` : `struct task_struct`.

Communications inter-processus (IPC)

- Besoins :
  - Partage d'informations ;
  - Modularité ;
  - Séparation des privilèges.
- De nombreux types d'IPC [1] ;
- Les principaux sont :
  - Fichiers ;
  - Sockets ;
  - Verrous, sémaphores ;
  - Signaux ;
  - Mémoire partagée ;
  - Files de message ;
  - Tubes et tubes nommés ;

There's a lot of IPC

- Pipes
- FIFOs
- Pseudoterminals
- Sockets
  - Stream vs Datagram (vs Seq. packet)
  - UNIX vs Internet domain
- POSIX message queues
- POSIX shared memory
- POSIX semaphores
  - Named, Unnamed
- System V message queues
- System V shared memory
- System V semaphores
- Shared memory mappings
  - File vs Anonymous
- Cross-memory attach
  - proc_vm_readv() / proc_vm_writev()
- Signals
  - Standard, Realtime
- Eventfd
- Futexes
- Record locks
- File locks
- Mutexes
- Condition variables
- Barriers
- Read-write locks

It helps to classify

- Pipes
- FIFOs
- Pseudoterminals
- Sockets
  - Stream vs Datagram (vs Seq. packet)
  - UNIX vs Internet domain
- POSIX message queues
- POSIX shared memory
- POSIX semaphores
  - Named, Unnamed
- System V message queues
- System V shared memory
- System V semaphores
- Shared memory mappings
  - File vs Anonymous
- Cross-memory attach
  - proc_vm_readv() / proc_vm_writev()
- Signals
  - Standard, Realtime
- Eventfd
- Futexes
- Record locks
- File locks
- Mutexes
- Condition variables
- Barriers
- Read-write locks

It helps to classify

- Pipes
- FIFOs
- Pseudoterminals
- Sockets
  - Stream vs Datagram (vs Seq. packet)
  - UNIX vs Internet domain
- POSIX message queues
- POSIX shared memory
- POSIX semaphores
  - Named, Unnamed
- System V message queues
- System V shared memory
- System V semaphores

Communication

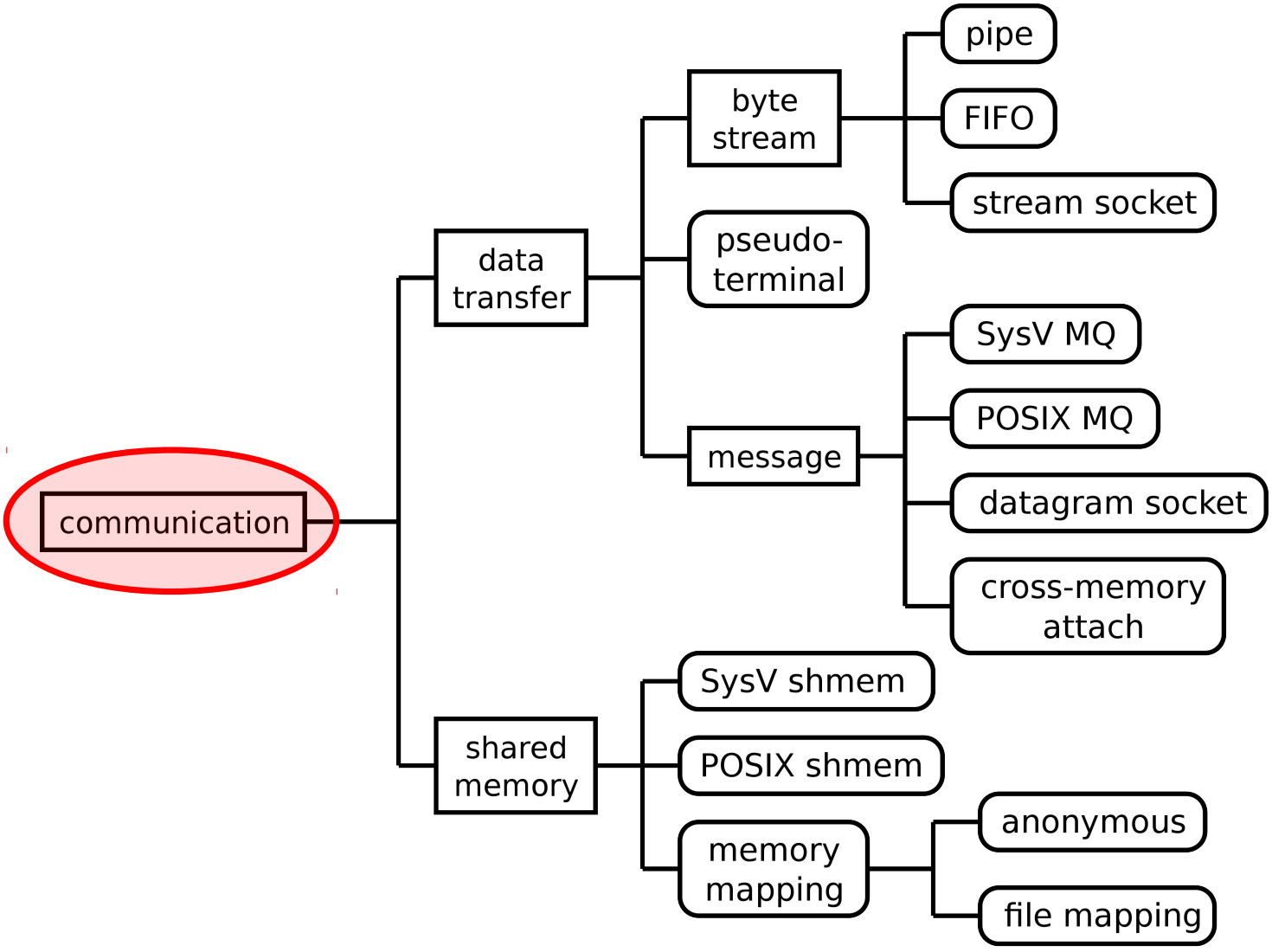
- Shared memory mappings
  - File vs Anonymous
- Cross-memory attach
  - proc_vm_readv() / proc_vm_writev()
- Signals
  - Standard, Realtime
- Eventfd

Signals

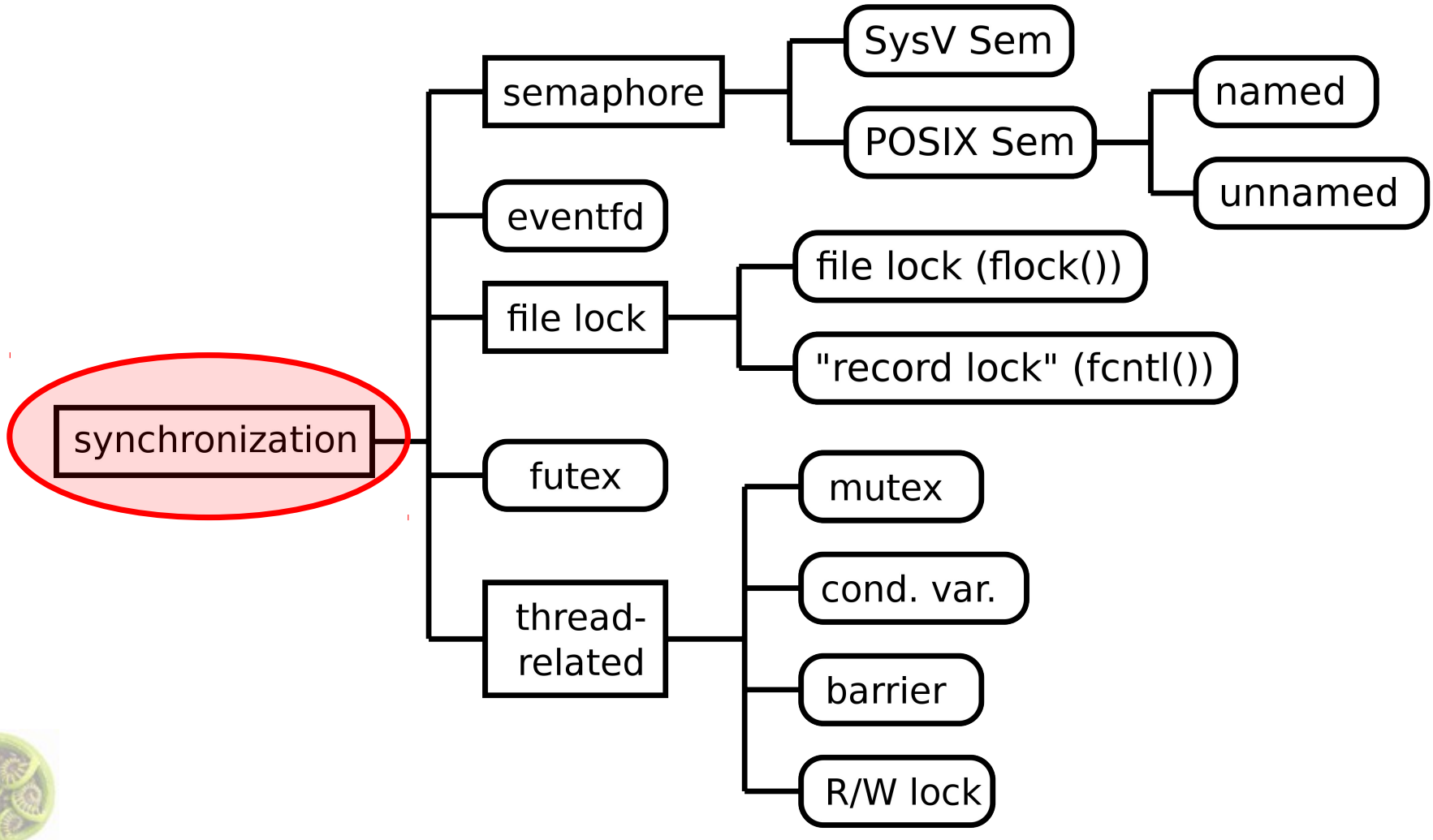
- Futexes
- Record locks
- File locks
- Mutexes
- Condition variables
- Barriers
- Read-write locks

Synchronization

Communication



Synchronizatoin



System V vs POSIX

- Deux implémentations :
  - System V ;
  - POSIX.
- Si possible, privilégier l'implémentation POSIX (meilleure interface) ;
- Ironiquement, l'interface System V est plus « portable » car implémentée sur plus de systèmes ;

Plan

1 Threads
- Comparatif processus vs thread
- Utilisation des threads POSIX

2 Synchronisation

3 Tubes

4 Signaux

5 Files de messages

6 Mémoire partagée

Pourquoi les threads ?

- Communication entre processus limitée aux IPC ;
- Partage limité des ressources et des données de travail ;
- Changements de contexte entre deux processus coûteux ;
- Déduplication de processus effectuant la même tâche sur les même données.

Concept du thread

- Fil d'exécution parallèle dans un processus ;
- Sorte de processus plus « léger » ;
- Plusieurs threads associé à un processus ;
- Partage d'un certains nombre d'informations entre les différents threads d'un même processus ;

Processus vs thread

- Processus indépendants vs threads liés à un processus ;
- Les processus contiennent plus d'information sur leur d'état qu'un thread. Les threads partagent ces informations dans un même processus.
- Les processus ont chacun leur espace d'adressage mémoire. Les threads le partagent avec le processus initial ;
- Le changement de contexte entre deux threads sera plus rapide que le changement de contexte entre deux processus.

Pour deux threads dans un même processus

- Partagé :
  - Espace d'adressage mémoire ;
  - Code du programme ;
  - Tas (heap) ;
  - Variables d'environnement ;
  - Descripteurs de fichiers ;
  - Gestionnaire de signaux.
- Distinct :
  - Pile d'exécution (stack) ;
  - Gestionnaire d'exceptions ;
  - Priorité d'ordonnancement ;
  - Stockage local (!= stack) ;
  - Identifiants uniques (PID, UID, GID...).

Thread userspace vs kernelspace

- Plusieurs implémentations possibles des threads :
  - 1 : 1 (Kernel-level threading) : chaque thread correspond à un entité distincte dans le noyau. L'ordonnancement est géré par thread ;
  - N : 1 (User-level threading) : les threads sont créés et gérés en espace utilisateur (pas d'interaction avec le noyau). Changements de contexte très rapide entre les threads. Pas d'exécution concourante si un appel est bloquant ;
  - M : N (Hybrid threading) : compromis entre les cas précédents. Complexe et potentiellement moins efficace.

Thread userspace vs kernelspace

- En pratique : utilisation des threads noyaux ;
- Win32 Threads sous Windows [2] ;
- POSIX Threads ou NSThread sous Mac OS X.
- Native Posix Thread Library sous Linux (NTPL) :
  - Appel système `clone` ;
  - Utilisation des fonctions de l'API POSIX.
  - `#include <pthread.h>`
  - Compilation avec l'option `-lpthread` pour l'éditeur de liens ;

Création d'un thread

- `int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*fonction) (void *), void *argument);`
- `pthread_t * thread` : identifiant similaire au PID d'un processus pour un thread :
- `thread_attr_t * attributs` : attributs donnés au thread à sa création ;
- `void *(*fonction) (void *)` : pointeur vers la fonction exécutée dans le thread ;
- `void * argument` ; pointeur passé comme argument à la fonction.

Attributs d'un thread

- Type opaque : `pthread_attr_t` ;
- Initialisé avec `pthread_attr_init`, détruit avec `pthread_attr_destroy` ;
- Fonctions `pthread_attr_set*()` et `pthread_attr_get*()` ;
- Il n'est plus possible de modifier les attributs d'un thread un fois qu'il est démarré ;
- Exception : `int pthread_detach(pthread_t thread)`.

Attributs d'un thread

- pthread_attr_{get,set}affinity_np(3)
- pthread_attr_{get,set}detachstate(3)
- pthread_attr_{get,set}guardsize(3)
- pthread_attr_{get,set}inheritsched(3)
- pthread_attr_{get,set}schedparam(3)
- pthread_attr_{get,set}schedpolicy(3)
- pthread_attr_{get,set}scope(3)
- pthread_attr_{get,set}stack(3)
- pthread_attr_{get,set}stackaddr(3)
- pthread_attr_{get,set}stacksize(3)

Terminaison d'un thread

- Un thread se termine quand la fonction qui lui est associée se termine : `void pthread_exit(void *retval);`
- Ne pas utiliser `exit`, qui mettrai fin à l'exécution du programme entier ;
- Attribut `detashstate` :
  - `JOIGNABLE` : le thread attends qu'un autre thread du même processus appelle `pthread_join` avant de se terminer. Possibilité de récupérer la valeur de retour ;
  - `DETACHED` : le thread se termine sans retourner de valeur de retour ni attendre un autre thread.
- On peut forcer l'état `DETACHED` avec `pthread_detach()`.

Terminaison d'un thread : cleanup

- Il est possible de définir des fonctions qui seront appelées lors de la terminaison d'un thread : `void pthread_cleanup_push(void (*routine)(void *), void *arg)` :
  - Ces fonctions seront exécutées dans l'ordre inverse de leur ajout ;
  - Permets de libérer les ressources qui ont été utilisée par un thread de façon inconditionnelle lors de sa terminaison.
- On retire les fonctions précédemment ajoutées avec `void pthread_cleanup_pop(int execute)`.

Terminaison d'un thread : cancel

- Il est possible de terminer un thread prématurément :
  `int pthread_cancel(pthread_t thread)` :
  - Attribut `cancelstate` du thread à `enabled` ;
  - Attribut `canceltype` :
    - `asynchronous` : terminaison « immédiate » ;
    - `deferred` : terminaison au prochain appel de `void pthread_testcancel(void)` ou d'une fonction répertoriée comme point d'annulation (`pthread(7)`).

Note générale : cas non définis (undefined behavior)

- Beaucoup de cas non spécifiés par la norme POSIX ;
- Lire attentivement les pages de manuel !
- `pthreads(7)`, `pthread_*(3)` et `pthread_*(3p)`.

Plan

safety »

1 Threads

2 Synchronisation
- Section critique
- Opération atomique
- Mutex et Sémaphore
- Concept de « thread

3 Tubes

4 Signaux

5 Files de messages

6 Mémoire partagée

Section critique

- Portion de code qui ne doit pas être exécutée par plus d'un thread à la fois ;
- Exemples : accès aux ressources partagées entre les threads :
  - Thread 1 lit la variable v = 1 ;
  - Thread 2 lit la variable v ;
  - Thread 1 ajoute 1 à v et stocke le résultat dans v ;
  - Thread 2 ajoute 1 à v et stocke le résultat dans v ;
  - v ne vaut alors pas 3 mais 2.
- Nécessité d'un mécanisme pour s'assurer qu'un seul thread effectue un ensemble d'action sur une ressource.

Opération atomique

- Opération dont l'exécution est considérée comme « instanée » ;
- S'appuie sur des instructions du processeur :
  - Test-and-set : défini une variable à une valeur et retourne la valeur précédente ;
  - Fetch-and-add : lit une valeur, lui ajoute un, stocke le résultat ;
  - Compare-and-swap : compare deux valeurs, si elles sont égales alors la première est remplacée par une nouvelle valeur.
- Fondamentalement limité par les instruction implémentées dans le processeur.

Mutex

- Primitive de synchronisation pouvant prendre deux états : disponible ou verrouillé ;
- Permets de définir des sections critiques arbitrairement ;
- Initialisation avec `pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER` (attributs par défaut) ou `int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *attr);` (attributs particuliers) ;
- Destruction avec `pthread_mutex_destroy(pthread_mutex_t *mutex)`.

Mutex

- Verrouillage avec `int pthread_mutex_lock(pthread_mutex_t *mutex)` :
  - Si le mutex est libre : verrouillage, et les appels à `pthread_mutex_lock()` seront alors bloquant pour les autres threads ;
  - Si le mutex est déjà verrouillé : le thread sera lui aussi bloqué ;
- Déverrouillage avec `int pthread_mutex_unlock(pthread_mutex_t *mutex)` ;
- `int pthread_mutex_trylock(pthread_mutex_t *mutex)` : ne bloque pas le thread si le mutex est verrouillé.

Mutex : attributs

- Définitions d'attributs avec `int pthread_mutexattr_set*(...)` :
  - `prioceiling` : priorité maximale à laquelle sera exécutée une section critique ;
  - `protocol` : changement de priorité suivant différents cas ;
  - `pshared` : pour autoriser le partage d'un mutex entre plusieurs processus (cf section sur la mémoire partagée) ;
  - `robust` : comportement en cas de terminaison du thread ayant verrouillé le mutex ;
  - `type` : comportement du mutex lors du (dé)verrouillage.

Mutex : attributs

Mutex type	**Robustness**	**Relock**	**Unlock when not owner**
NORMAL	non-robust	deadlock	undefined behavior
NORMAL	robust	deadlock	error returned
ERRORCHECK	either	error returned	error returned
RECURSIVE	either	recursive	error returned
DEFAULT	non-robust	undefined behavior	undefined behavior
DEFAULT	robust	undefined behavior	error returned

Sémaphores

- Mutex étendu pour permettre plusieurs accès simultanés : le nombre d'accès autorisé est stocké dans un compteur ;
- Deux implémentations :
  - SystemV : sem* ;
  - POSIX : sem_*.
- Si possible, privilégier l'implémentation POSIX.

Sémaphores anonymes

- Initialisation avec `int sem_init(sem_t *sem, int pshared, unsigned int value)` : peut être local à une application ou partagé entre plusieurs processus (cf section sur la mémoire partagée) ;
- `int sem_wait(sem_t *sem)` : prendre une « place » (diminue le compteur de 1) ou attendre qu'une place se libère (si le compteur est à 0) ;
- `int sem_post(sem_t *sem)` : libérer le sémaphore (augmente le compteur de 1) ;
- `int sem_getvalue(sem_t *sem, int *sval)` : obtenir la valeur actuelle du compteur.

Sémaphores nommés

- Ouverture avec `sem_t *sem_open(const char *name, int oflag...)` : crée un nouveau sémaphore nommé ou en ouvre un existant ;
- Le nom doit être de la forme `/nomdusemaphore` ;
- Ne pas confondre l'attribut partagé d'un sémaphore anonyme et les sémaphores nommés (cf section sur la mémoire partagée) ;
- Le fonctionnement est similaire à un sémaphore anonyme ;
- Liste des semaphores POSIX : `ls /dev/shm/sem.*` ;
- Liste des semaphores System V : `ipcs -s`.

Verrous en lecture et écriture

- `pthread_rwlock_t` : autorise plusieurs lecteurs simultanément, mais un seul écrivain ;
- Fonctionnement très similaire aux verrous (mutex) ;
- `int pthread_rwlock_wrlock(pthread_rwlock_t *rwlock);`
- `int pthread_rwlock_rdlock(pthread_rwlock_t *rwlock);;`
- `int pthread_rwlock_unlock(pthread_rwlock_t *rwlock).`

pthread_cond

- Attendre le déverrouillage d'un mutex pour effectuer une action ;
- Possibilité de définir plusieurs conditions ;
- Lorsqu'elle est atteinte, on peut « réveiller » uniquement les threads qui attendent cette condition.

pthread_cond

- Thread 1 :
  - Initialise la condition et le mutex ;
  - Obtient le mutex ;
  - `pthread_cond_wait()` : rends le mutex et bloque en attendant la condition ; Lorsque la condition est atteinte, le mutex est à nouveau obtenu ;
  - Le thread continue son exécution.
- Thread 2 :
  - Réalise la condition ;
  - Bloque le mutex lié à la condition ;
  - `pthread_cond_{broadcast,signal}()` : informe les autres threads que la condition est remplie ;
  - Rends le mutex.

Interblocage (deadlock)

- Blocage qui se produit lorsque deux threads ou processus essaient d'accéder à des ressources et qu'ils se bloquent l'un l'autre ;

Interblocage (deadlock) : exemple

- Thread 1 :
  - Obtenir verrous 1 ;
  - Obtenir verrous 2 ;
  - Opération critique ;
  - Rendre verrous 2 ;
  - Rendre verrous 1 ;
- Thread 2 :
  - Obtenir verrous 2 ;
  - Obtenir verrous 1 ;
  - Opération critique ;
  - Rendre verrous 1 ;
  - Rendre verrous 2 ;

- Thread 1 obtient le verrous 1 ;
- Thread 2 obtient le verrous 2 ;
- Thread 1 bloque sur le verrous 2 ;
- Thread 2 bloque sur le verrous 1 ;

Concept de « thread safety »

- Une portion de code est considérée « thread safe » si elle manipule des données partagées entre plusieurs d'une façon qui garantie une exécution sure en cas d'exécution concurrente par plusieurs threads ;
- Exemples :
  - Variables statiques : elles sont partagées entre les threads ;
  - Protection de l'accès à des structures en utilisant des verrous ;
- Liste des fonctions « thread-safe » : `pthreads(7)`.

Exemple : le cas d'errno

- Stocke le code d'erreur retournée par un appel système ;
- Défini dans le header `errno.h` ;
- Stockage local à un thread : chaque thread à sa propre variable `errno`.
- Attention : il ne faut pas redéfinir `errno` !

Exemple : gethostbyname, strerror...

- Fonction dépréciées de la `libc` qui utilisent une variable statique comme stockage pour la valeur de retour ;
- L'utilisation simultanée de ces fonctions dans des threads conduira à des bugs difficiles à comprendre ;
- Il faut toujours faire très attention en lisant les pages de manuel et ne pas utiliser de fonction considérées comme dépréciées.

Plan

1 Threads

2 Synchronisation

3 Tubes
- Tubes anonymes
- Tubes nommés

4 Signaux

5 Files de messages

6 Mémoire partagée

Tubes anonymes (pipes)

- `$ ls | wc -l.`

Tubes anonymes (pipes)

- Flux d'octets avec un buffer au niveau du noyau ;
- Séquentiel : contrairement aux fichiers, on ne peut pas utiliser `off_t lseek(int fildes, off_t offset, int whence)` ;
- Unidirectionnel ;
- Pratique uniquement s'il n'y a qu'un seul processus en lecture et un seul en écriture.

Création d'un tube anonyme

- `int pipe(int filedesc[2])` : crée un tube.
- Les données sont alors écrites dans `filedesc[1]` et lues depuis `filedesc[0]` ;
- FIFO : first-in-first-out ;
- `int pipe2(int pipefd[2], int flags);` : spécifique Linux ; ajoute d'autres options ;

Partage d'un tube anonyme

- Tubes anonymes : pas de nom dans le système de fichier ;
- Les processus créés après la création du pipe en héritent et peuvent donc s'en servir.
- Ne pas oublier de fermer le descripteur dont on ne se sert pas.

Entrées / sorties avec les tubes anonyme

- `read()` : bloquant si le tube est vide ;
- `write()` : bloquant si le tube est plein ;
- Taille du tube :
  - Avant le noyau 2.6.11 : 4096 octets ;
  - Après le 2.6.11 : 65 536 octets.
- L'écriture de données de taille inférieure à `PIPE_BUF` est atomique (restreint à 512 octets par POSIX, 4096 octets sous Linux) ;
- Permets d'utiliser les tubes anonymes comme moyen de communication synchronisé.

Création d'un tube nommé

- Tube avec un nom dans le système de fichier ;
- Shell : `mkfifo a=rw mon_tube` ; suppression avec `rm` ;
- C : `int mkfifo(const char *pathname, mode_t mode)` :
  - `mode` : Attention à l'`umask` ;
  - Le tube n'est pas détruit lorsqu'un processus arrête de s'en servir ;
  - Suppression explicite avec `unlink`.

Exemple d'utilisation de tube nommé

- Shell :

```
$ mkfifo mon_tube
$ gzip -9 -c < mon_tube > compressé.gz &
$ cat un_fichier un_autre_fichier > mon_tube
$ file mon_tube
```

- C :
  - `open(pathname, O_RDONLY)` : ouvre l'extrémité en lecture ;
  - `open(pathname, O_WRONLY)` : ouvre l'extrémité en écriture ;
  - L'appel bloque tant que l'autre extrémité n'est pas ouverte.

Plan

1 Threads

2 Synchronisation

3 Tubes

4 Signaux
- Les différents types de signaux
- Gestion des signaux
- Signaux temps réel
- Terminaux
- Fonctions POSIX sûres
- Quand utiliser les signaux ?

5 Files de messages

6 Mémoire partagée

Signaux : concept

- Notification asynchrone indiquant qu'un évènement particulier s'est produit ;
- Interrompt « immédiatement » un processus ;
- Celui-ci peut alors :
  - ignorer le signal ;
  - le capturer pour exécuter certaines instructions (gestionnaire de signal) puis reprendre l'exécution du processus ;
  - laisser le noyau traiter le signal avec le comportement par défaut (Term, Ign, Core, Stop, Cont).
- Interruption logicielle (par rapport aux interruptions matérielles).

Signaux : concept

- Un signal peut être émis pour l'une des raisons suivantes :
  - une exception matérielle c'est produite, le noyau en est informé et envoie le signal correspondant au processus en cours d'exécution. Exemple : mauvaise instruction, division par zéro, accès à une zone mémoire non référencée ;
  - un utilisateur a entrée un caractère spécial dans un terminal ce qui a généré un signal (`interrupt` : ^C, `suspend` : ^Z).
  - un évènement logiciel s'est produit : des données peuvent être lues à partir d'un descripteur de fichiers, la fenêtre du terminal a été redimensionnée, un fils du processus s'est terminé...

Signaux : types

- Linux supporte les 32 signaux standard ainsi que les 32 signaux temps réel (normes POSIX) ;
- Chaque signal est défini par un numéro unique auquel est associé un nom symbolique (SIG*) ;
- Éviter d'utiliser les numéros, et préférer les noms explicites (compatibilité entre les implémentations et sûreté) ;
- Liste complète : signal(7).

Signaux classiques

- L'ensemble des signaux (non temps réel) en attente pour un processus est stocké sous forme de masque ;
- Les signaux classiques ne s'empilent pas contrairement aux signaux temps réel ;
- La réception d'un signal `SIGCHLD` par exemple peut signifier que **plusieurs** fils du processus ont fini de s'exécuter ;

Signaux classiques et appels systèmes

- Les signaux reçus par un processus peuvent restent temporairement bloqués si celui-ci est en train d'effectuer un appel système (sauf SIGKILL et SIGSTOP) ;
- Certains appels systèmes sont non bloquants (getpid, gettimeofday, getuid, setuid...). Les signaux sont bloqués puisque l'appel système n'est pas sensé durer longtemps.
- Certains appels systèmes sont bloquants (sleep, read...). Le noyau peut décider d'interrompre l'appel système pour envoyer le signal.

Signaux classiques et appels systèmes

- Par défaut, les appels systèmes interrompus sont relancés sous Linux ;
- `int siginterrupt(int sig, int flag)` : définit le comportement pour le processus ;
- Il faut alors faire particulièrement attention au code de retour des fonctions bloquantes ;

Signaux : comportements classiques

- `SIGKILL` : termine de façon inconditionnelle un processus. Ne peut pas être capturé ;
- `SIGSTOP` : mets un processus en pause. Ne peut pas être capturé ;
- `SIGTERM` : indique (gentiment) à un processus qu'il doit se terminer. Attention : aucune garantie !
- `SIGINT` : idem, mais résulte généralement d'une commande clavier (Control-C) ;
- `SIGABRT` : indique la fin anormale d'un processus (et la création d'un core dump) ;
- `SIGILL` : instruction invalide ;
- `SIGCONT` : reprends l'exécution d'un programme.

Signaux : comportements classiques

- `SIGCHLD` : indique que l'un des fils de ce processus s'est terminé ou que son exécution a été stoppée (`SIGSTOP`) ou reprise
- `SIGSEGV` : accès à une zone mémoire non référencée (segmentation fault) ;
- `SIGFPE` : erreur lors d'un calcul arithmétique (division par zéro) ;
- `SIGPIPE` : reçu si un processus essaie d'écrire dans un tube, une FIFO ou une socket pour lesquels il n'y a plus de processus en lecture ;
- `SIGQUIT` : tue le processus en produisant un core dump.

Signaux : comportements classiques

- `SIGSYS` : le processus a demandé au noyau d'exécuter un appel système qui n'existe pas ;
- `SIGUSR1`/`SIGUSR2` : aucun sens défini. Ils ne seront jamais directement envoyés par le noyau. Les processus peuvent utiliser ces signaux pour notifier d'autres processus d'un événement ou s'en servir comme moyen de synchronisation ;
- `SIGWINCH` : envoyé lorsque que la taille d'un terminal change, généralement lors du redimensionnement de la fenêtre qui le contenait.

Émettre un signal

- `int kill(pid_t pid, int sig);`
- `pid` : processus cible ;
- Si le pid est -1, alors le signal est envoyé à tous les processus auxquels le processus a le droit d'envoyer le signal ;
- `sig` : numéro du signal ;
- On peut se servir de 0 comme numéro de signal pour tester l'existence d'un processus sans lui envoyer de signaux ;

Gestion des signaux : signal()

- `void (*signal(int sig, void (*handler)(int)))(int);)`
- Interface héritée d'UNIX ;
- `sig` : le signal dont on veut changer la gestion ;
- `func` : la fonction qui sera appelée lors de la réception du signal ;
- Renvoie un pointeur sur la fonction qui gérait le signal précédemment.
- Simple, mais généralement insuffisant et non portable. A éviter !

Gestion des signaux : sigprocmask()

- `int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);`
- Bloque l'arrivée des signaux présent dans le `sigset` ;
- Les signaux reçus sont tout de mêmes conservés et seront activés si le masque change ;
- Ce masque est spécifique à un thread et peut être modifié indépendamment avec `pthread_sigmask()` ;
- `int sigpending(sigset_t *set);` pour récupérer la liste des signaux en attente.

Gestion des signaux : sigaction()

- `int sigaction(int sig, const struct sigaction *act, struct sigaction *oldact);`
- `sig` : le signal dont on veut changer le comportement ;
- `act` : structure décrivant :
  - la fonction chargée de la gestion du signal (handler) ;
  - les signaux bloqués pendant l'exécution du handler ;
  - d'autres options.
- `act` : la version précédente de la structure sigaction.

Gestion des signaux : attendre un signal

- `int pause(void);`

Signaux temps réel

- Numéros de `SIGRTMIN` à `SIGRTMAX` ;
- Comportement par défaut : terminaison du processus ;
- Stockées dans une file d'attente (plusieurs instance d'un même signal peuvent être reçues à la suite) ;
- Plus le signal a un numéro petit, plus il est traité en priorité ;
- Pour une même priorité, l'ordre d'arrivée est garanti d'être celui dans lequel les signaux ont été envoyés ;
- Il est possible d'envoyer des informations supplémentaires avec le signal.

Signaux temps réel

- Très peu utilisé en pratique :
  - notification pour les opérations asynchrones sur les fichiers ;
  - notifications pour les files de messages ;
  - signaux définis spécifiquement pour une application.
- Il faut préférer les autres IPCs aux signaux temps réel.

Signaux et terminaux

- ^C : SIGINT ;
- ^Z : SIGTSTP ;
- kill -<nomdusignal> <pid> ... :
  - $ kill -term `pidof firefox`
  - $ kill -kill `pidof firefox`
  - $ killall firefox
  - $ killall -s STOP firefox
  - $ pkill -cont firefox

« Async-signal-safe functions »

- Ensemble de fonction que l'on peut utiliser sans risque dans un gestionnaire de signaux ;
- Elles ont été pensée pour que leur comportement reste valide quel que soit l'instant où elles sont interrompues.
- Liste dans la page de manuel : signal(7).

Faut-il utiliser les signaux ?

- Interface peu pratique ;
- Contraintes très fortes sur ce qu'il est possible de faire dans les fonctions de traitement des signaux ;
- Il faut éviter au maximum d'utiliser les signaux ;
- Voir la section `signalfd` dans le cours sur les « eventloop ».

Plan

1 Threads

2 Synchronisation

3 Tubes

4 Signaux

5 Files de messages
- Concept
- Usage

6 Mémoire partagée

File de messages (message queue ou MQ)

- Communication à l'aide de message :
  - Les messages sont lus un à un, en intégralité ;
  - Contrairement aux tubes, il est possible et pratique d'avoir plusieurs écrivains / lecteurs ;
- Les messages ont des priorités : ils sont « livrés » par ordre de priorité ;
- Un processus peut demander à être prévenu de l'arrivée d'un nouveau message.

Usage

- `mq_open()` : ouvre ou crée une MQ ;
- `mq_close()` : ferme une MQ ;
- `mq_unlink()` : supprime une MQ du système de fichier ;
- `mq_send()`, `mq_receive()` : envoyer et recevoir des messages ;
- `mq_setattr()`, `mq_getattr()` : définir ou récupérer les attributs d'une MQ ;
- `mq_notify()` : demander à être prévenu de l'arrivée d'un message.

Ouvrir une POSIX MQ

- `mqd = mq_open(name, flags, [mode, attr]);`
- Ouvre ou crée une nouvelle MQ ;
- Nom sous la forme : `/filedemessage` ;
- Pour lister les MQ POSIX : `ls /dev/mqueue` ;
- Pour lister les MQ System V : `ipcs -q`.

Ouvrir une POSIX MQ

- `mqd = mq_open(name, flags, [mode, attr]);`
- `flags` :
  - `O_CREAT` : créer la MQ si elle n'existe pas ;
  - `O_EXCL` : (avec `O_CREAT`) ne crée pas la MQ si elle existe déjà et retourne un code d'erreur ;
  - `O_RDONLY`, `O_WRONLY`, `O_RDWR` : comportement classique.
- `mode` : permissions ;
- `attr` : attributs (`NULL` = par défaut).

Supprimer une MQ

- `mq_unlink` : supprime une MQ.
- Seul le nom est effectivement supprimé du système de fichier ;
- La MQ n'est détruite qu'une fois que tous les processus ont appelé `mq_close` ;
- Principe du « reference-counting ».

Comportement et taille

- Comme les tubes, les MQ ont une capacité limitée (`mq_maxmsg` et `mq_msgsize`) ;
- `mq_receive()` : bloque s'il n'y a pas de messages dans la MQ ;
- `mq_send()` : bloque s'il n'y a plus de place dans la MQ ;

Envoie d'un message

- `mq_send(mqd, msg_ptr, msg_len, msgprio);` :
  - `mqd` : pointeur sur le type opaque `mqd_t` ;
  - `msg_ptr` : pointeur sur le message ;
  - `msg_len` : taille du message ;
  - `msgprio` : priorité du message :
    - priorité >= 0 ;
    - 0 == priorité minimale.

Réception d'un message

- Retourne le message le plus vieux parmi les messages avec la plus haute priorité ;
- `taille = mq_receive(mqd, msg_ptr, msg_len, prio);` :
  - `mqd` : pointeur sur le type opaque `mqd_t` ;
  - `msg_ptr` : pointeur sur un buffer pour recevoir le message ;
  - `msg_len` : taille du buffer ;
  - `msgprio` : priorité du message reçu ;
  - `taille` : retourne la taille du message reçu.

Notifications d'arrivée de messages

- Un processus peut demander à recevoir une notification lorsque qu'un message arrive dans une MQ si :
  - la MQ était **vide** ;
  - aucun autre processus n'est en train de lire de message.
- `mq_notify(mqd, notification)` : indique comment le processus veut être notifié :
  - Démarre un nouveau thread ;
  - Reçois un signal (cf partie sur les signaux).
- Ne se déclenche qu'une seule fois. Il faut le réactiver systématiquement et ce avant de vider la MQ (sinon on risque de louper des notifications).

Plan

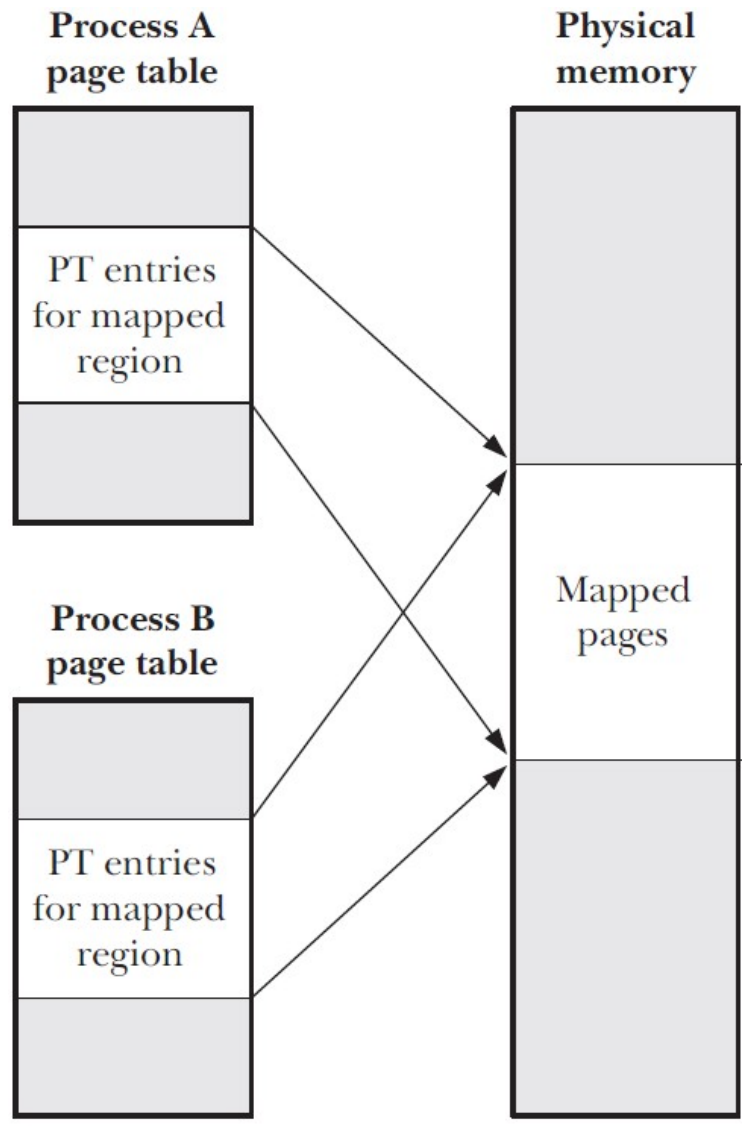
1 Threads
2 Synchronisation
3 Tubes
4 Signaux
5 Files de messages
6 Mémoire partagée

Partage de mémoire

- Partage d'une page de mémoire entre deux processus ;
- Il suffit alors d'écrire des données dans cette zone pour les partager ;
- Méthode de communication plus efficace que les tubes, MQ...
- Mais il faut synchroniser les accès à cette zone mémoire (mutex, semaphores, read/write lock).

Shared memory

- Processes share physical pages of memory



Différents types de partage mémoire

- Partage anonyme :
  - entre un processus et ses fils uniquement.
- Partage lié à un fichier :
  - partage entre plusieurs processus ;
  - contenu stocké sur le disque dans un fichier ;
  - plus sûr.
- Partage nommé (POSIX) :
  - partage entre plusieurs processus ;
  - pas de stockage dans un fichier ;
  - plus rapide.

Appel système mmap

- `void *mmap(void *daddr, size_t len, int prot, int flags, int fd, off_t offset);`
  - `daddr` : où placer la zone de mémoire. Le mieux est de laisser le noyau choisir en mettant `NULL` ;
  - `len` : taille de la zone mémoire ;
  - `prot` : protection mémoire (lecteur, écriture, exécution) ;
  - `flags` : contrôle du comportement (partagé ou anonyme) ;
  - `fd` : le descripteur de fichier correspondant pour les partage utilisant un fichier ;
  - `offset` : le décalage par rapport au début du fichier ;
  - `addr` : l'adresse où est placée la zone de mémoire.

Appel système mmap

- `addr` : fonctionne de la même façon que n'importe quel pointeur en C ;
- Mais cette adresse n'est pas nécessairement similaire pour tous les processus qui partage de la mémoire.

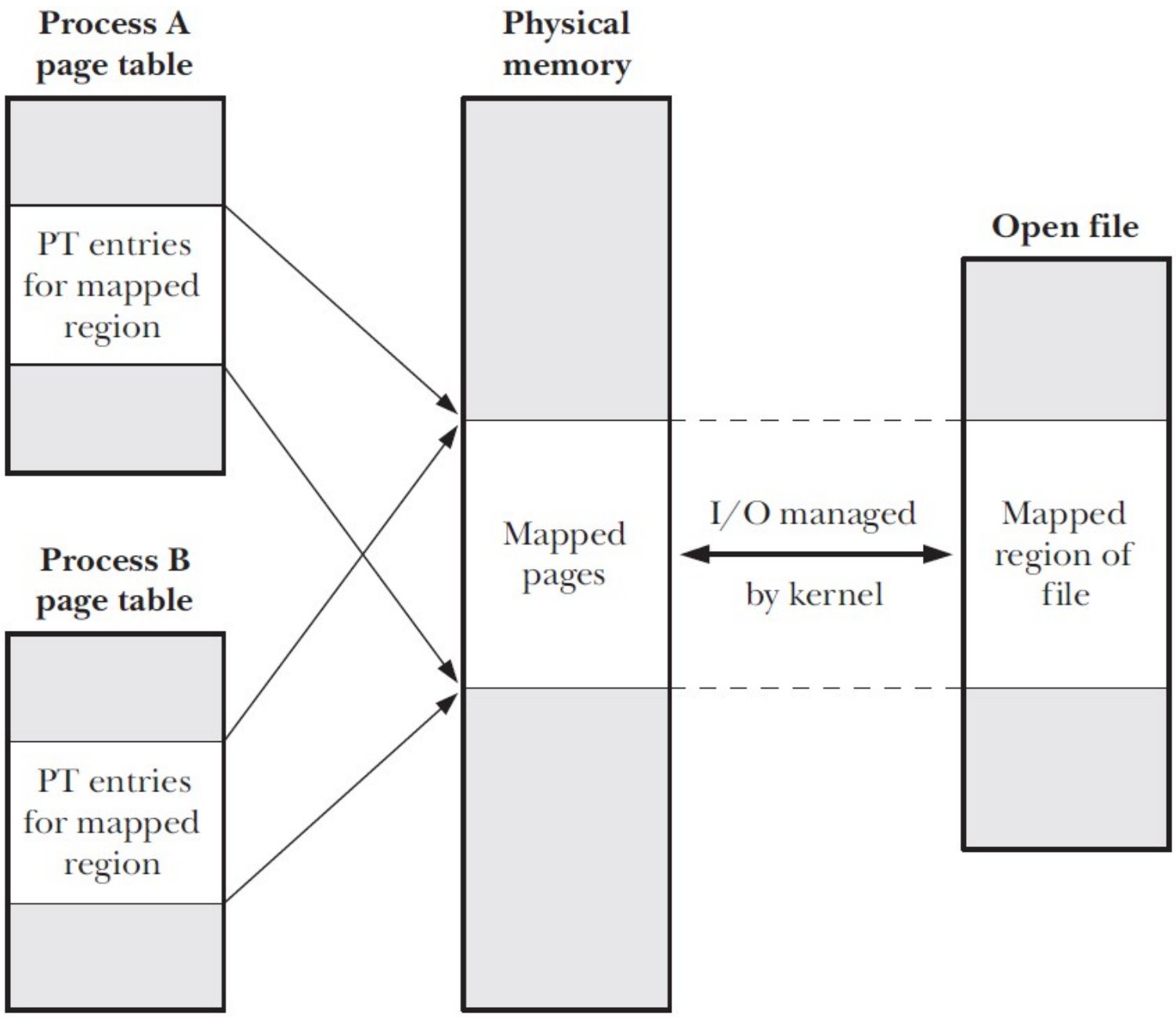
Partage anonyme

- Partage de mémoire entre un père et ses fils ;
- `addr = mmap(NULL, length, PROT_READ | PROT_WRITE, MAP_SHARED | MAP_ANONYMOUS, -1, 0);`
  `pid = fork();;`
- Alloue un block d'au moins `length` octets ;
- Père et fils partagent la zone mémoire à l'adresse addr.

Partage par l'intermédiaire d'un fichier

- Partage d'une portion de fichiers entre plusieurs processus ;
- `fd = open(...); addr = mmap(..., fd, offset);`
- Le contenu de cette zone mémoire est celui du fichier ;
- Les changements impactent automatiquement le fichier ;
- Tous les processus qui `mmap` la même portion de fichier partagent aussi la même région en mémoire.

Shared file mapping



Partage nommé (POSIX shared memory)

- Partage d'une zone mémoire entre plusieurs processus sans créer de fichier traditionnel ;
- Pas besoin de stocker le contenu sur le disque donc plus performant.

Partage nommé (POSIX) : usage

- `shm_open()` : ouverture ou création d'un partage nommé ;
- `mmap()` : mappage en mémoire de ce partage ;
- `shm_unlink()` : suppression d'un partage nommé ;
- `fstat()` : informations liés au partage (taille, propriétaire, permissions) ;
- `ftruncate()` : changement de taille ;
- `fchown()`, `fchmod()` : changement de propriétaire ou de permissions.

Partage nommé (POSIX) : ouverture

- `fd = shm_open(nom, flags, mode);`
- Ouvre ou crée un partage nommé et retourne un desripteur de fichier ;
- Comme pour les MQ, le `nom` est sous la forme : `/partagenommé` ;
- Pour lister les partage nommés POSIX : `ls /dev/shm` ;
- Pour lister les partage nommés System V : `ipcs -m`.

Partage nommé (POSIX) : ouverture

- `fd = shm_open(nom, flags, mode);`
- `flags` : similaire à ce qui est utilisé pour `open()` :
  - `O_CREAT` : crée le partage si il n'existe pas ;
  - `O_EXCL` : le crée uniquement si il n'existe pas ;
  - `O_RDONLY`, `O_RDWR`... : indique le type d'accès ;
  - `O_TRUNC` : supprime le contenu actuel d'un partage nommé en changeant sa taille à 0 ;
- `mode` : définis les permissions d'accès.

Partage nommé (POSIX) : accès

- Création et mappage d'un nouveau partage :

```
fd = shm_open("/mon_shm", O_CREAT | O_EXCL |
                          O_RDWR, 0600);
// Définir la taille du partage
ftruncate(fd, size);
addr = mmap(NULL, size, PROT_READ | PROT_WRITE,
                        MAP_SHARED, fd, 0);
```

- Ouverture et mappage d'un partage existant :

```
// Pas d'option O_CREAT
fd = shm_open("/myshm", O_RDWR, 0);
// Récupération de la taille du partage
struct stat sb; fstat(fd, &sb);
addr = mmap(NULL, sb.st_size, PROT_READ | PROT_WRITE,
                              MAP_SHARED, fd, 0);
```

Partage nommé (POSIX) : synchronisartion

- Il est possible de partager des mutex, des sémaphores ou d'autres structures par l'intermédiaire de partages nommés ;
- Le développeur est responsable de la synchronisation des accès par plusieurs processus aux données stockées dans un partage mémoire.

Références I

1. An introduction to Linux IPC, Michael Kerrisk : `http://man7.org/conf/lca2013/IPC_Overview-LCA-2013-printable.pdf`
2. About Processes and Threads (on Windows) : `http://msdn.microsoft.com/en-us/library/windows/desktop/ms681917%28v=vs.85%29.aspx`
3. The Linux Programming Interface by Michael Kerrisk : `http://man7.org/tlpi/index.html`