Programmation Multi-Tâches

programmation C - POSIX

Matthias BRUN

3 mai 2021

ESEO Apprentissage

Introduction

Définiton :

Un système informatique est qualifié de multitâche quand il permet d'exécuter de façon *apparente* plusieurs programmes informatiques (processus).

→ Alternance rapide d'exécution des processus en mémoire.

→ Le multitâche n'implique pas d'avoir un système multiprocesseur.

Motivation :

- Confort :
  - → Plusieurs utilisateurs sur la même machine.
  - → Exécution de plusieurs programmes en même temps.
- Conception :
  - → Faire coopérer plusieurs programmes simples, plutôt qu'un seul programme capable de tout faire.

Plan

- tâche $\sim$ **unité d'exécution** (séquentielle, avec pile et contexte)
- *n* tâches sur *p* CPU, si $n > p \Rightarrow$ besoin **ordonnanceur**
- politique d'ordonnancement
  - basé sur des quotas de temps (ex. *round robin*)
  - basé sur des priorités (ex. *fixed priority*, temps-réel)
  - basé sur des heuristiques (ex. objectif confort utilisateur)
  - \+ préemptive ou non préemptive
- modèle d'états d'une tâche
  - *ready*, *running*, *waiting* (ex. AUTOSAR, Linux)
  - \+ *suspended* (ex. AUTOSAR), *blocked* (ex. Linux), etc.
- préemption d'une tâche
  - sur interruption matériel ou appel système bloquant
  - $\Rightarrow$ changement de contexte
- isolation des tâches (accès espace mémoire)
  - pas d'isolation (ex : OSEK/VDX)
  - isolation par **processus**
    - un seul fil d'exécution possible par processus (ex : Unix)
    - plusieurs fils d'exécution possibles : ***threads*** (ex : Linux)

- tâche $\sim$ **unité d'exécution** (séquentielle, avec pile et contexte)
- n tâches sur p CPU, si $n > p \Rightarrow$ besoin **ordonnanceur**
- politique d'ordonnancement
  - basé sur des quotas de temps (ex. *round robin*)
  - basé sur des priorités (ex. *fixed priority*, temps-réel)
  - basé sur des heuristiques (ex. objectif confort utilisateur)
  - \+ préemptive ou non préemptive
- modèle d'états d'une tâche
  - *ready*, *running*, *waiting* (ex. AUTOSAR, Linux)
  - \+ *suspended* (ex. AUTOSAR), *blocked* (ex. Linux), etc.
- préemption d'une tâche
  - sur interruption matériel ou appel système bloquant
  - $\Rightarrow$ changement de contexte
- isolation des tâches (accès espace mémoire)
  - pas d'isolation (ex : OSEK/VDX)
  - isolation par **processus**
    - un seul fil d'exécution possible par processus (ex : Unix)
    - plusieurs fils d'exécution possibles : ***threads*** (ex : Linux)

- tâche $\sim$ **unité d'exécution** (séquentielle, avec pile et contexte)
- n tâches sur p CPU, si $n > p \Rightarrow$ besoin **ordonnanceur**
- politique d'ordonnancement
  - basé sur des quotas de temps (ex. *round robin*)
  - basé sur des priorités (ex. *fixed priority*, temps-réel)
  - basé sur des heuristiques (ex. objectif confort utilisateur)
  - \+ préemptive ou non préemptive
- modèle d'états d'une tâche
  - *ready*, *running*, *waiting* (ex. AUTOSAR, Linux)
  - \+ *suspended* (ex. AUTOSAR), *blocked* (ex. Linux), etc.
- préemption d'une tâche
  - sur interruption matériel ou appel système bloquant
  - $\Rightarrow$ changement de contexte
- isolation des tâches (accès espace mémoire)
  - pas d'isolation (ex : OSEK/VDX)
  - isolation par **processus**
    - un seul fil d'exécution possible par processus (ex : Unix)
    - plusieurs fils d'exécution possibles : ***threads*** (ex : Linux)

- tâche $\sim$ **unité d'exécution** (séquentielle, avec pile et contexte)
- n tâches sur p CPU, si $n > p \Rightarrow$ besoin **ordonnanceur**
- politique d'ordonnancement
  - basé sur des quotas de temps (ex. *round robin*)
  - basé sur des priorités (ex. *fixed priority*, temps-réel)
  - basé sur des heuristiques (ex. objectif confort utilisateur)
  - \+ préemptive ou non préemptive
- modèle d'états d'une tâche
  - *ready*, *running*, *waiting* (ex. AUTOSAR, Linux)
  - \+ *suspended* (ex. AUTOSAR), *blocked* (ex. Linux), etc.
- préemption d'une tâche
  - sur interruption matériel ou appel système bloquant
  - $\Rightarrow$ changement de contexte
- isolation des tâches (accès espace mémoire)
  - pas d'isolation (ex : OSEK/VDX)
  - isolation par **processus**
    - un seul fil d'exécution possible par processus (ex : Unix)
    - plusieurs fils d'exécution possibles : ***threads*** (ex : Linux)

- tâche $\sim$ **unité d'exécution** (séquentielle, avec pile et contexte)
- n tâches sur p CPU, si $n > p \Rightarrow$ besoin **ordonnanceur**
- politique d'ordonnancement
  - basé sur des quotas de temps (ex. *round robin*)
  - basé sur des priorités (ex. *fixed priority*, temps-réel)
  - basé sur des heuristiques (ex. objectif confort utilisateur)
  - \+ préemptive ou non préemptive
- modèle d'états d'une tâche
  - *ready*, *running*, *waiting* (ex. AUTOSAR, Linux)
  - \+ *suspended* (ex. AUTOSAR), *blocked* (ex. Linux), etc.
- préemption d'une tâche
  - sur interruption matériel ou appel système bloquant
  - $\Rightarrow$ changement de contexte
- isolation des tâches (accès espace mémoire)
  - pas d'isolation (ex : OSEK/VDX)
  - isolation par **processus**
    - un seul fil d'exécution possible par processus (ex : Unix)
    - plusieurs fils d'exécution possibles : ***threads*** (ex : Linux)

- tâche $\sim$ **unité d'exécution** (séquentielle, avec pile et contexte)
- n tâches sur p CPU, si $n > p \Rightarrow$ besoin **ordonnanceur**
- politique d'ordonnancement
  - basé sur des quotas de temps (ex. *round robin*)
  - basé sur des priorités (ex. *fixed priority*, temps-réel)
  - basé sur des heuristiques (ex. objectif confort utilisateur)
  - \+ préemptive ou non préemptive
- modèle d'états d'une tâche
  - *ready*, *running*, *waiting* (ex. AUTOSAR, Linux)
  - \+ *suspended* (ex. AUTOSAR), *blocked* (ex. Linux), etc.
- préemption d'une tâche
  - sur interruption matériel ou appel système bloquant
  - $\Rightarrow$ changement de contexte
- isolation des tâches (accès espace mémoire)
  - pas d'isolation (ex : OSEK/VDX)
  - isolation par **processus**
    - un seul fil d'exécution possible par processus (ex : Unix)
    - plusieurs fils d'exécution possibles : ***threads*** (ex : Linux)

POSIX (*Portable Operating System Interface - from UNIX*)

Standard (IEEE 1003) d'interface de programmation des logiciels (API, *Application Programming Interface*) pour systèmes d'exploitation (OS, *Operating System*) variantes d'UNIX.

Processus

Espaces mémoires distincts (gérés par MMU, *Memory Managment Unit*) dans lesquels peuvent s'exécuter un ou plusieurs **threads**.

- Historique : Unix (1970) offre notion de processus (avec un seul fil d'exécution), fin des années 1990 possibilité de plusieurs fils d'exécution (les threads) (Exemple : Linux 2.2 1999).
- Régulation de l'accès au processeur par l'ordonnanceur (*scheduler*) du noyau de l'OS.

POSIX (*Portable Operating System Interface - from UNIX*)

Standard (IEEE 1003) d'interface de programmation des logiciels (API, *Application Programming Interface*) pour systèmes d'exploitation (OS, *Operating System*) variantes d'UNIX.

Processus

Espaces mémoires distincts (gérés par MMU, *Memory Managment Unit*) dans lesquels peuvent s'exécuter un ou plusieurs **threads**.

- Historique : Unix (1970) offre notion de processus (avec un seul fil d'exécution), fin des années 1990 possibilité de plusieurs fils d'exécution (les threads) (Exemple : Linux 2.2 1999).
- Régulation de l'accès au processeur par l'ordonnanceur (*scheduler*) du noyau de l'OS.

POSIX (*Portable Operating System Interface - from UNIX*)

Standard (IEEE 1003) d'interface de programmation des logiciels (API, *Application Programming Interface*) pour systèmes d'exploitation (OS, *Operating System*) variantes d'UNIX.

Processus

Espaces mémoires distincts (gérés par MMU, *Memory Managment Unit*) dans lesquels peuvent s'exécuter un ou plusieurs **threads**.

- Historique : Unix (1970) offre notion de processus (avec un seul fil d'exécution), fin des années 1990 possibilité de plusieurs fils d'exécution (les threads) (Exemple : Linux 2.2 1999).
- Régulation de l'accès au processeur par l'ordonnanceur (*scheduler*) du noyau de l'OS.

Processus

Système :

- liste des processus : `ps ax`
- PID : identifiant d'un processus (rq : `init`, PID 1)
- liste hiérarchique : `ps [f]axj`
- PPID : identifiant d'un processus père (rq : `init`, PPID 0)

Programmation :

- création d'un processus : `pid_t fork(void)`

→ duplication du processus appelant (≪ père ≫), en un processus ≪ fils ≫

→ exécution à partir du retour de `fork` :
return = *child PID* $> 0 \Rightarrow$ processus père, *return* $= 0 \Rightarrow$ porcessus fils

→ appel système économe (*copy-on-write*)

- PID : `pid_t getpid(void)`
- PPID : `pid_t getppid(void)`

Processus

Système :

- liste des processus : `ps ax`
- PID : identifiant d'un processus (rq : `init`, PID 1)
- liste hiérarchique : `ps [f]axj`
- PPID : identifiant d'un processus père (rq : `init`, PPID 0)

Programmation :

- création d'un processus : `pid_t fork(void)`

→ duplication du processus appelant (≪ père ≫), en un processus ≪ fils ≫

→ exécution à partir du retour de `fork` :
return = *child PID* $> 0 \Rightarrow$ processus père, *return* $= 0 \Rightarrow$ porcessus fils

→ appel système économe (*copy-on-write*)

- PID : `pid_t getpid(void)`
- PPID : `pid_t getppid(void)`

Processus

Système :

- liste des processus : `ps ax`
- PID : identifiant d'un processus (rq : `init`, PID 1)
- liste hiérarchique : `ps [f]axj`
- PPID : identifiant d'un processus père (rq : `init`, PPID 0)

Programmation :

- création d'un processus : `pid_t fork(void)`

→ duplication du processus appelant (≪ père ≫), en un processus ≪ fils ≫

→ exécution à partir du retour de `fork` :
return = *child PID* $> 0 \Rightarrow$ processus père, *return* $= 0 \Rightarrow$ porcessus fils

→ appel système économe (*copy-on-write*)

- PID : `pid_t getpid(void)`
- PPID : `pid_t getppid(void)`

Processus

Exécution d'un programme dans un processus :

- appels systèmes : cf. `man 3 exec`

→ remplacer l'espace mémoire d'un processus appelant par le code et les données d'une nouvelle application

- `execl`, `execle`, `execlp`, `execv`, `execve`, `execvp`

→ basées sur l'appel système `execve` (cf. `man execve`)

→ '`l`' : liste d'arguments, '`v`' : vecteur d'arguments (tableau)

→ '`e`' : transmission de l'environnement dans un tableau (`envp[]`)
(par défaut utilisation implicite de la variable globale `environ`)

→ '`p`' : utilise `PATH` (var. env.) pour chercher exécutables
(par défaut nécessite un chemin complet, absolu ou relatif)

Processus

Exécution d'un programme dans un processus :

- appels systèmes : cf. `man 3 exec`

→ remplacer l'espace mémoire d'un processus appelant par le code et les données d'une nouvelle application

- `execl`, `execle`, `execlp`, `execv`, `execve`, `execvp`

→ basées sur l'appel système `execve` (cf. `man execve`)

→ '`l`' : liste d'arguments, '`v`' : vecteur d'arguments (tableau)

→ '`e`' : transmission de l'environnement dans un tableau (`envp[]`)
(par défault utilisation implicite de la variable globale `environ`)

→ '`p`' : utilise `PATH` (var. env.) pour chercher exécutables
(par défault nécessite un chemin complet, absolu ou relatif)

Processus

Exécution d'un sous-programme (programme dans processus fils) :

- `fork` + `exec`
- `int system (cons char * cmd)`
  - → ∼ `fork()` + `execl("/bin/sh", "sh", "-c", cmd, (char *)NULL)`
  - → `system` retourne quand la commande lancée se termine
  - → Attention : Faille de sécurité si application avec Set-UID root
- `FILE * popen(const char * cmd, const char * mode)`
  - → ∼ `system(cmd)` avec flux d'entrée ou sortie standard (`mode` = `"w"` ou `"r"`)
  - → retourne le flux de communication
  - → Fermeture nécessaire du flux avec `pclose`
  - → Attention : Faille de sécurité si application avec Set-UID root

Processus

Exécution d'un sous-programme (programme dans processus fils) :

- `fork` + `exec`
- `int system (cons char * cmd)`

→ ∼ `fork()` + `execl("/bin/sh", "sh", "-c", cmd, (char *)NULL)`

→ `system` retourne quand la commande lancée se termine

→ Attention : Faille de sécurité si application avec Set-UID root

- `FILE * popen(const char * cmd, const char * mode)`

→ ∼ `system(cmd)` avec flux d'entrée ou sortie standard (`mode` = `"w"` ou `"r"`)

→ retourne le flux de communication

→ Fermeture nécessaire du flux avec `pclose`

→ Attention : Faille de sécurité si application avec Set-UID root

Processus

Exécution d'un sous-programme (programme dans processus fils) :

- `fork` + `exec`
- `int system (cons char * cmd)`

→ ∼ `fork()` + `execl("/bin/sh", "sh", "-c", cmd, (char *)NULL)`

→ `system` retourne quand la commande lancée se termine

→ Attention : Faille de sécurité si application avec Set-UID root

- `FILE * popen(const char * cmd, const char * mode)`

→ ∼ `system(cmd)` avec flux d'entrée ou sortie standard (`mode` = `"w"` ou `"r"`)

→ retourne le flux de communication

→ Fermeture nécessaire du flux avec `pclose`

→ Attention : Faille de sécurité si application avec Set-UID root

Processus

Terminaison d'un programme

- `return` sur thread principal (*main*)
- `void exit(int code)`

→ code retour (cf. `EXIT_SUCCESS`, `EXIT_FAILURE`, shell `$?`)

1. appel des fonctions enregistrées avec `atexit` (ordre inverse enregistrements)
2. *flush* et fermeture des flux *IO*
3. appel système `_exit()` pour terminer le processus :

→ fermeture des descripteurs de fichiers (transfert des données aux périphériques)
→ libération des ressources verrouillées
→ processus fils adoptés par `init` (ou `systemd --user`)
→ processus père reçoit le signal `SIGCHLD`
→ processus devient zombie (`Z`) en attente que son père lise son code retour

- `void abort(void)`

→ Terminaison anormale (signal `SIGABRT` au processus, génération fichier core)

Processus

Terminaison d'un programme

- `return` sur thread principal (*main*)
- `void exit(int code)`

→ code retour (cf. `EXIT_SUCCESS`, `EXIT_FAILURE`, shell `$?`)

1. appel des fonctions enregistrées avec `atexit` (ordre inverse enregistrements)
2. *flush* et fermeture des flux *IO*
3. appel système `_exit()` pour terminer le processus :

→ fermeture des descripteurs de fichiers (transfert des données aux périphériques)
→ libération des ressources verrouillées
→ processus fils adoptés par `init` (ou `systemd --user`)
→ processus père reçoit le signal `SIGCHLD`
→ processus devient zombie (`Z`) en attente que son père lise son code retour

- `void abort(void)`

→ Terminaison anormale (signal `SIGABRT` au processus, génération fichier core)

Processus

Terminaison d'un programme

- `return` sur thread principal (*main*)
- `void exit(int code)`

→ code retour (cf. `EXIT_SUCCESS`, `EXIT_FAILURE`, shell `$?`)

1. appel des fonctions enregistrées avec `atexit` (ordre inverse enregistrements)
2. *flush* et fermeture des flux *IO*
3. appel système `_exit()` pour terminer le processus :

→ fermeture des descripteurs de fichiers (transfert des données aux périphériques)

→ libération des ressources verrouillées

→ processus fils adoptés par `init` (ou `systemd --user`)

→ processus père reçoit le signal `SIGCHLD`

→ processus devient zombie (`Z`) en attente que son père lise son code retour

- `void abort(void)`

→ Terminaison anormale (signal `SIGABRT` au processus, génération fichier core)

Processus

Terminaison d'un programme

- `return` sur thread principal (*main*)
- `void exit(int code)`

→ code retour (cf. `EXIT_SUCCESS`, `EXIT_FAILURE`, shell `$?`)

1. appel des fonctions enregistrées avec `atexit` (ordre inverse enregistrements)
2. *flush* et fermeture des flux *IO*
3. appel système `_exit()` pour terminer le processus :

→ fermeture des descripteurs de fichiers (transfert des données aux périphériques)

→ libération des ressources verrouillées

→ processus fils adoptés par `init` (ou `systemd --user`)

→ processus père reçoit le signal `SIGCHLD`

→ processus devient zombie (`Z`) en attente que son père lise son code retour

- `void abort(void)`

→ Terminaison anormale (signal `SIGABRT` au processus, génération fichier core)

Processus

Fin d'un processus

- état zombie (`Z`) (en attente lecture code retour par processus père)
- lecture code retour d'un fils : `pid_t wait(int * status)`

→ appel bloquant jusqu'à terminaison d'un fils

→ retourne `PID` du fils terminé

→ retourn -1 si pas de fils à attendre (`errno == ECHILD`)

→ `status` = `NULL` : on ne s'intéresse pas à la circonstance de terminaison du fils

→ `status` ≠ `NULL` : ensemble de macros pour évaluer `status` (cf. `man 3 wait`)

- `pid_t waitpid(pid_t pid, int * status, int options)`

→ `pid` : fils dont on attend la fin (-1 : attente de n'importe quel fils)

→ `options` : `WNOHANG` non bloquant, `WUNTRACED` accès également fils stoppés

À suivre : `wait` + gestionnaire de signaux + `SIGCHLD`

Processus

Fin d'un processus

- état zombie (`Z`) (en attente lecture code retour par processus père)
- lecture code retour d'un fils : `pid_t wait(int * status)`

→ appel bloquant jusqu'à terminaison d'un fils

→ retourne PID du fils terminé

→ retourn -1 si pas de fils à attendre (`errno == ECHILD`)

→ `status` = `NULL` : on ne s'intéresse pas à la circonstance de terminaison du fils

→ `status` ≠ `NULL` : ensemble de macros pour évaluer `status` (cf. `man 3 wait`)

- `pid_t waitpid(pid_t pid, int * status, int options)`

→ `pid` : fils dont on attend la fin (-1 : attente de n'importe quel fils)

→ `options` : `WNOHANG` non bloquant, `WUNTRACED` accès également fils stoppés

À suivre : `wait` + gestionnaire de signaux + `SIGCHLD`

Processus

Fin d'un processus

- état zombie (`Z`) (en attente lecture code retour par processus père)
- lecture code retour d'un fils : `pid_t wait(int * status)`

→ appel bloquant jusqu'à terminaison d'un fils

→ retourne PID du fils terminé

→ retourn -1 si pas de fils à attendre (`errno == ECHILD`)

→ `status` = `NULL` : on ne s'intéresse pas à la circonstance de terminaison du fils

→ `status` ≠ `NULL` : ensemble de macros pour évaluer `status` (cf. `man 3 wait`)

- `pid_t waitpid(pid_t pid, int * status, int options)`

→ `pid` : fils dont on attend la fin (-1 : attente de n'importe quel fils)

→ `options` : `WNOHANG` non bloquant, `WUNTRACED` accès également fils stoppés

À suivre : `wait` + gestionnaire de signaux + `SIGCHLD`

Processus

Fin d'un processus

- état zombie (`Z`) (en attente lecture code retour par processus père)
- lecture code retour d'un fils : `pid_t wait(int * status)`

→ appel bloquant jusqu'à terminaison d'un fils

→ retourne PID du fils terminé

→ retourn -1 si pas de fils à attendre (`errno == ECHILD`)

→ `status` = `NULL` : on ne s'intéresse pas à la circonstance de terminaison du fils

→ `status` ≠ `NULL` : ensemble de macros pour évaluer `status` (cf. `man 3 wait`)

- `pid_t waitpid(pid_t pid, int * status, int options)`

→ `pid` : fils dont on attend la fin (-1 : attente de n'importe quel fils)

→ `options` : `WNOHANG` non bloquant, `WUNTRACED` accès également fils stoppés

À suivre : `wait` + gestionnaire de signaux + `SIGCHLD`

Processus

Fin d'un processus

- état zombie (`Z`) (en attente lecture code retour par processus père)
- lecture code retour d'un fils : `pid_t wait(int * status)`

→ appel bloquant jusqu'à terminaison d'un fils

→ retourne PID du fils terminé

→ retourn -1 si pas de fils à attendre (`errno == ECHILD`)

→ `status` = `NULL` : on ne s'intéresse pas à la circonstance de terminaison du fils

→ `status` ≠ `NULL` : ensemble de macros pour évaluer `status` (cf. `man 3 wait`)

- `pid_t waitpid(pid_t pid, int * status, int options)`

→ `pid` : fils dont on attend la fin (-1 : attente de n'importe quel fils)

→ `options` : `WNOHANG` non bloquant, `WUNTRACED` accès également fils stoppés

À suivre : `wait` + gestionnaire de signaux + `SIGCHLD`

Processus

Pour aller plus loin :

- processus et UID (identifiant utilisateur)
- processus et GID (identifiant de groupe)
- groupe de processus
- session de groupes de processus
- capacité ($\sim$ privilèges) d'un processus

Threads

Système :

- fils d'exécution qui partagent le même espace mémoire
- thread : pile et contexte d'exécution (registres CPU et cmptr. inst.)
- liste processus avec *threads* : `ps m[aux],ps -eLf`

Programmation :

- **pthread** : thread compatible Posix (1c) (années 90)
- bibliothèque NPTL (*Native Posix Thread Library*)

→ `<pthread.h>`

→ option gcc : `-pthread` (∼ `-D_REENTRANT` et `-lpthread`)

→ type d'un thread : `pthread_t` (∼ `unsigned long`, mais structure interne)

→ comparaison de threads : `pthread_equal(thread1, thread2)`

→ fonctions NPTL renvoient directement codes erreurs (définis dans `<errno.h>`)

→ `man pthreads`

Threads

Système :

- fils d'exécution qui partagent le même espace mémoire
- thread : pile et contexte d'exécution (registres CPU et cmptr. inst.)
- liste processus avec *threads* : `ps m[aux],ps -eLf`

Programmation :

- **pthread** : thread compatible Posix (1c) (années 90)
- bibliothèque NPTL (*Native Posix Thread Library*)

→ `<pthread.h>`

→ option gcc : `-pthread` (∼ `-D_REENTRANT` et `-lpthread`)

→ type d'un thread : `pthread_t` (∼ `unsigned long`, mais structure interne)

→ comparaison de threads : `pthread_equal(thread1, thread2)`

→ fonctions NPTL renvoient directement codes erreurs (définis dans `<errno.h>`)

→ `man pthreads`

Threads

Système :

- fils d'exécution qui partagent le même espace mémoire
- thread : pile et contexte d'exécution (registres CPU et cmptr. inst.)
- liste processus avec *threads* : `ps m[aux],ps -eLf`

Programmation :

- **pthread** : thread compatible Posix (1c) (années 90)
- bibliothèque NPTL (*Native Posix Thread Library*)

→ `<pthread.h>`

→ option gcc : `-pthread` (∼ `-D_REENTRANT` et `-lpthread`)

→ type d'un thread : `pthread_t` (∼ `unsigned long`, mais structure interne)

→ comparaison de threads : `pthread_equal(thread1, thread2)`

→ fonctions NPTL renvoient directement codes erreurs (définis dans `<errno.h>`)

→ `man pthreads`

Threads

Système :

- fils d'exécution qui partagent le même espace mémoire
- thread : pile et contexte d'exécution (registres CPU et cmptr. inst.)
- liste processus avec *threads* : `ps m[aux],ps -eLf`

Programmation :

- **pthread** : thread compatible Posix (1c) (années 90)
- bibliothèque NPTL (*Native Posix Thread Library*)

→ `<pthread.h>`

→ option gcc : `-pthread` (∼ `-D_REENTRANT` et `-lpthread`)

→ type d'un thread : `pthread_t` (∼ `unsigned long`, mais structure interne)

→ comparaison de threads : `pthread_equal(thread1, thread2)`

→ fonctions NPTL renvoient directement codes erreurs (définis dans `<errno.h>`)

→ `man pthreads`

Threads

Création d'un thread :

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

→ `thread` : pointeur initialisé par identifiant du thread créé

→ `attr` : attributs du thread (`NULL` pour attributs standards par défaut)

→ `start_routine` : point d'entrée du nouveau fil d'exécution

→ `arg` : pointeur passé en argument de la routine de point d'entrée

Threads

Terminaison d'un thread :

- `return` depuis fonction principale (type retour `void *`)
- `void pthread_exit(void * ret)`
- peut retourner `NULL`

Remarques :

- `exit()` : tout le processus se termine
- `return` dans thread principal (*main*) : tout le processus se termine
- `pthread_exit()` dans thread principal : seul le thread principal se termine
- erreur fatale dans un thread : tout le processus est tué

Threads

Terminaison d'un thread :

- `return` depuis fonction principale (type retour `void *`)
- `void pthread_exit(void * ret)`
- peut retourner `NULL`

Remarques :

- `exit()` : tout le processus se termine
- `return` dans thread principal (*main*) : tout le processus se termine
- `pthread_exit()` dans thread principal : seul le thread principal se termine
- erreur fatale dans un thread : tout le processus est tué

Threads

Valeur de retour d'un thread :

- `int pthread_join(pthread_t thread, void ** ret)`

→ suspension thread appelant jusqu'à terminaison thread en argument

→ valeur de retour via le pointeur `ret` (si $\neq$ `NULL`)

Remarques :

- `pthread_join` bloquant et explicite sur le thread à attendre
- pile du thread (avec code retour) conservée jusqu'à lecture de son code retour

Détachement d'un thread :

- `int pthread_detach(pthread_t thread)`

→ libération de la pile du thread dès sa terminaison

→ auto-détachement : `pthread_detach(pthread_self())`

→ `pthread_join` impossible sur thread détaché

Threads

Valeur de retour d'un thread :

- `int pthread_join(pthread_t thread, void ** ret)`

→ suspension thread appelant jusqu'à terminaison thread en argument

→ valeur de retour via le pointeur `ret` (si $\neq$ `NULL`)

Remarques :

- `pthread_join` bloquant et explicite sur le thread à attendre
- pile du thread (avec code retour) conservée jusqu'à lecture de son code retour

Détachement d'un thread :

- `int pthread_detach(pthread_t thread)`

→ libération de la pile du thread dès sa terminaison

→ auto-détachement : `pthread_detach(pthread_self())`

→ `pthread_join` impossible sur thread détaché

Threads

Valeur de retour d'un thread :

- `int pthread_join(pthread_t thread, void ** ret)`

→ suspension thread appelant jusqu'à terminaison thread en argument

→ valeur de retour via le pointeur `ret` (si $\neq$ `NULL`)

Remarques :

- `pthread_join` bloquant et explicite sur le thread à attendre
- pile du thread (avec code retour) conservée jusqu'à lecture de son code retour

Détachement d'un thread :

- `int pthread_detach(pthread_t thread)`

→ libération de la pile du thread dès sa terminaison

→ auto-détachement : `pthread_detach(pthread_self())`

→ `pthread_join` impossible sur thread détaché

Threads

Attributs d'un thread :

- type `pthread_attr_t` opaque : utilisation de fonctions d'accès

→ `int pthread_attr_init(pthread_attr_t * attr)`

→ attributs utilisés à la création d'un thread (mais non attachés)

→ attributs utilisables pour plusieurs créations de threads

→ `int pthread_attr_destroy(pthread_attr_t * attr)`

→ accesseurs : `pthread_attr_getXXX()`, `pthread_attr_setXXX()`

→ cf. `man pthread_attr_init`

Exemples :

```
pthread_attr_setdetachstate(&attr, state);
pthread_attr_getdetachstate(&attr, &state);
```

`state` : `PTHREAD_CREATE_DETACHED` ou `PTHREAD_CREATE_JOINABLE` (par défaut)

Threads

Attributs d'un thread :

- type `pthread_attr_t` opaque : utilisation de fonctions d'accès

→ `int pthread_attr_init(pthread_attr_t * attr)`

→ attributs utilisés à la création d'un thread (mais non attachés)

→ attributs utilisables pour plusieurs créations de threads

→ `int pthread_attr_destroy(pthread_attr_t * attr)`

→ accesseurs : `pthread_attr_getXXX()`, `pthread_attr_setXXX()`

→ cf. `man pthread_attr_init`

Exemples :

```
pthread_attr_setdetachstate(&attr, state);
pthread_attr_getdetachstate(&attr, &state);
```

`state` : `PTHREAD_CREATE_DETACHED` ou `PTHREAD_CREATE_JOINABLE` (par défaut)

Threads

Pour aller plus loin :

- Annulation d'un thread : `pthread_cancel`

→ nécessite une maîtrise des points d'annulation

- Nettoyage quand un thread se termine

→ mémoire allouée dynamiquement non libérée automatiquement

→ fichiers, tubes, sockets ouverts non fermés automatiquement

→ resources verrouillées non déverrouillées automatiquement

→ enregistrement de routines de libération dans une pile spéciale : `pthread_cleanup_push`, `pthread_cleanup_pop`

Accès concurrents

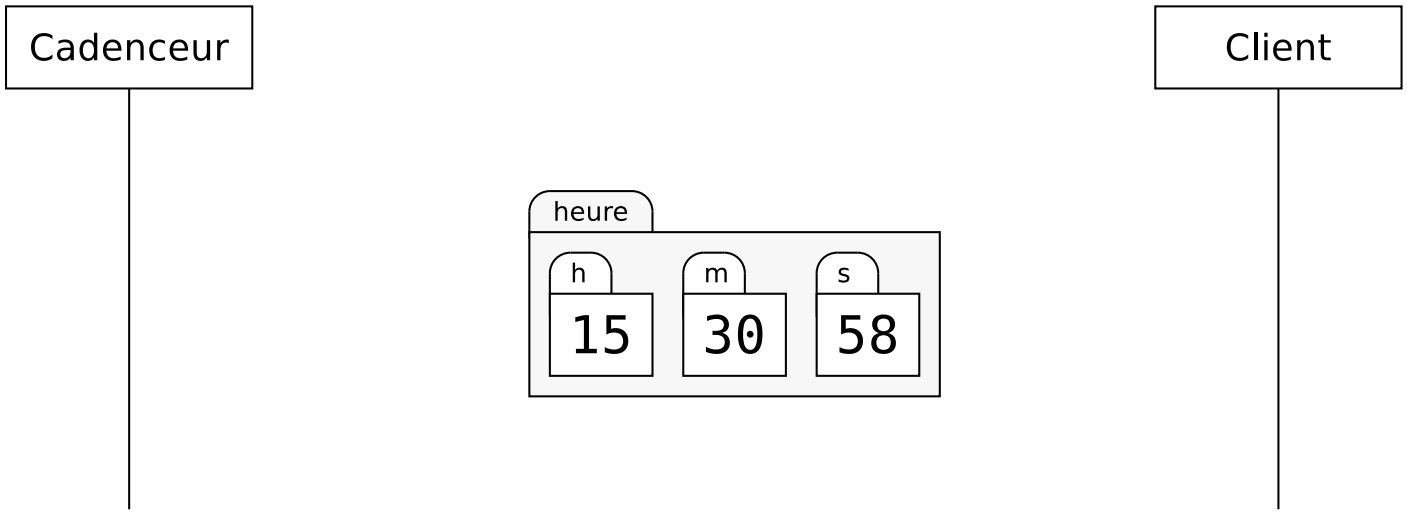
Exécutions concurrentes dans un espace mémoire partagé

$\Rightarrow$ risques de corruption de données.

Accès concurrents

Exécutions concurrentes dans un espace mémoire partagé

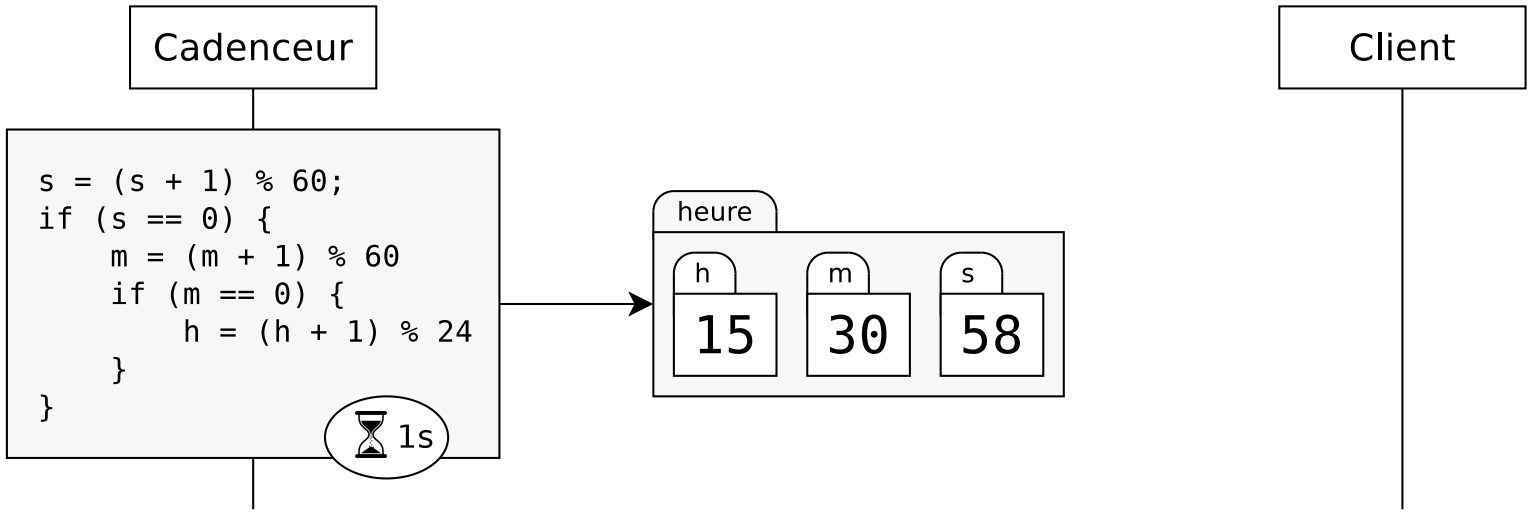
$\Rightarrow$ risques de corruption de données.



Accès concurrents

Exécutions concurrentes dans un espace mémoire partagé

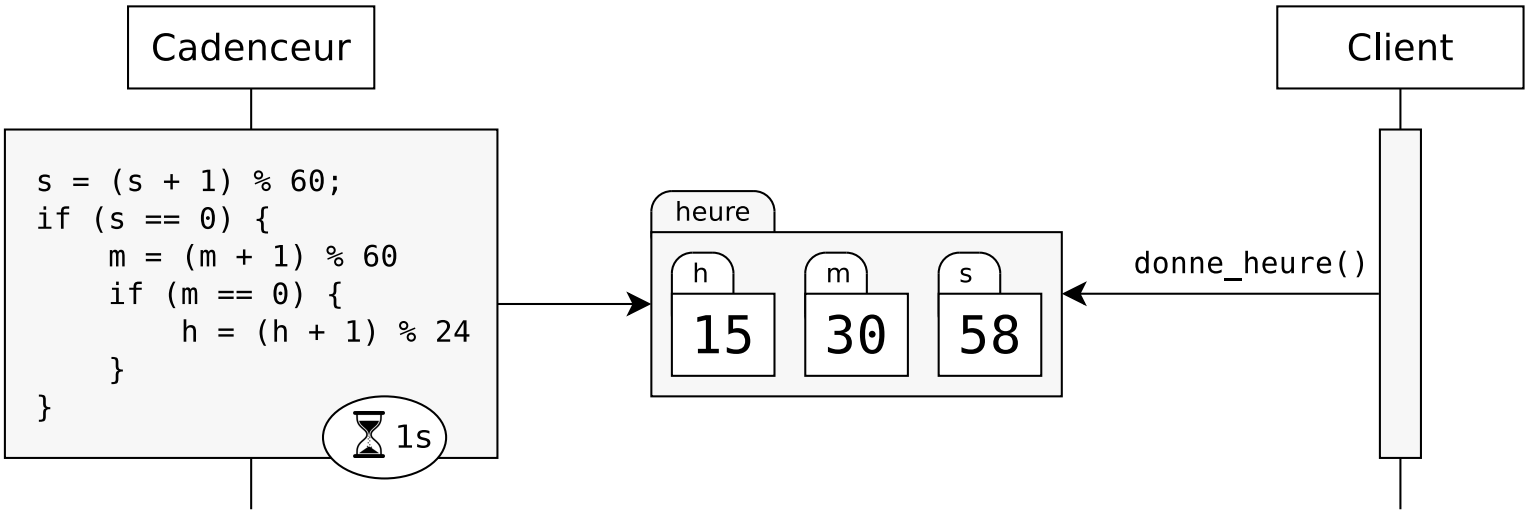
$\Rightarrow$ risques de corruption de données.



Accès concurrents

Exécutions concurrentes dans un espace mémoire partagé

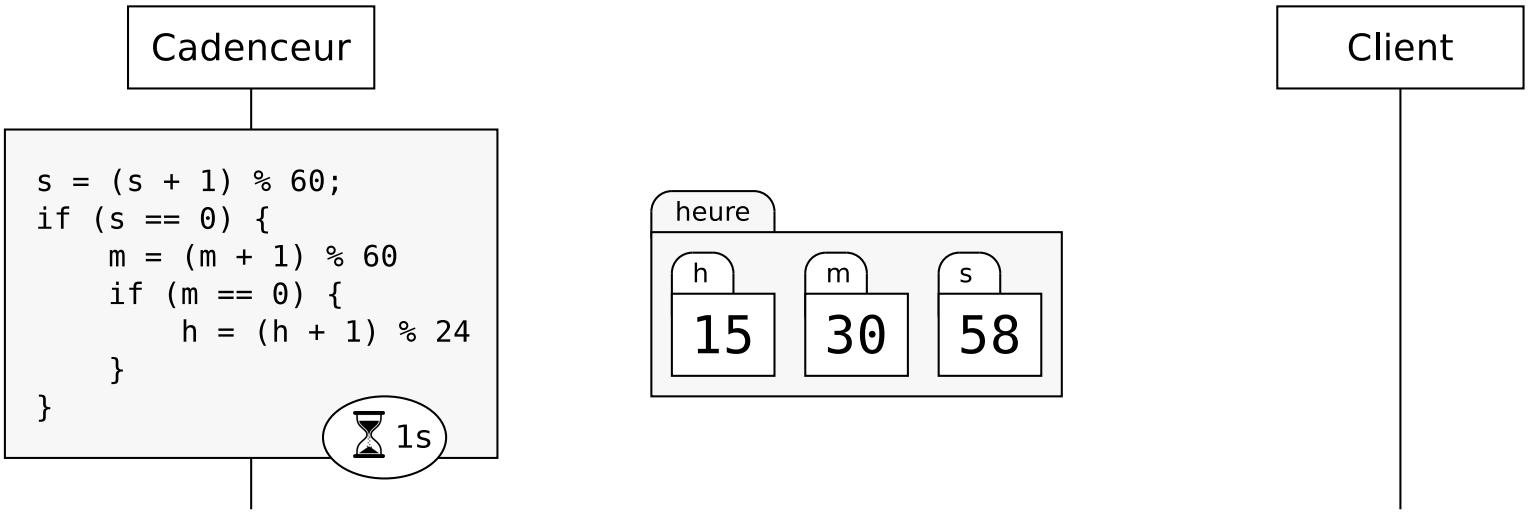
$\Rightarrow$ risques de corruption de données.



Accès concurrents

Exécutions concurrentes dans un espace mémoire partagé

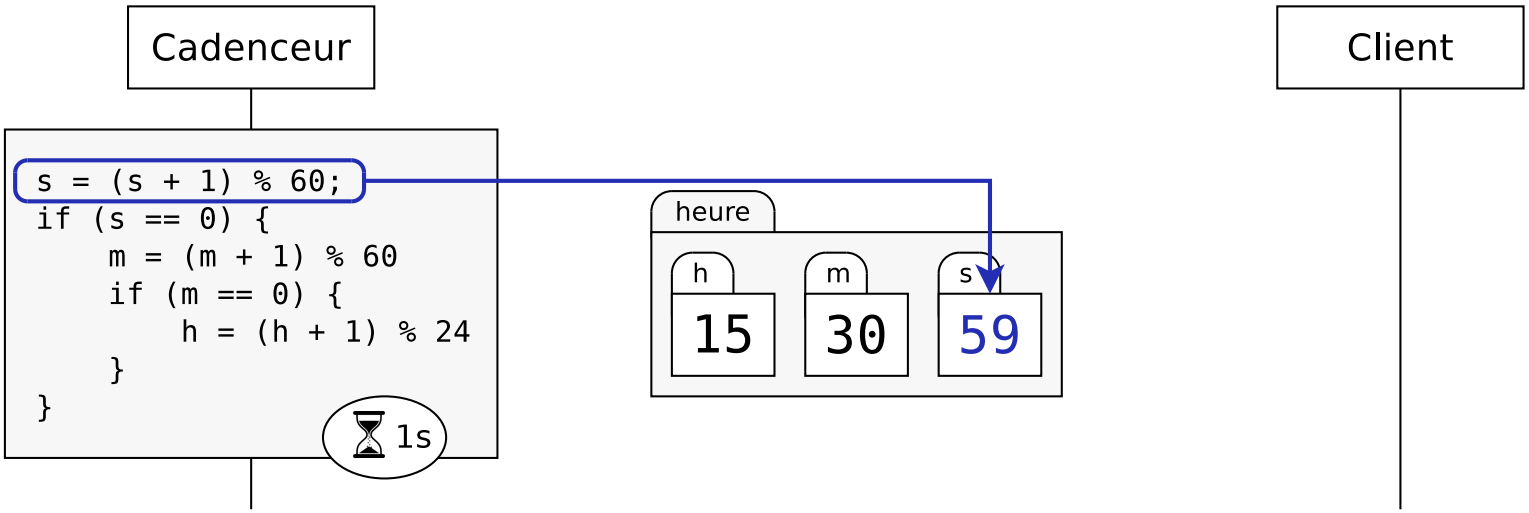
$\Rightarrow$ risques de corruption de données.



Accès concurrents

Exécutions concurrentes dans un espace mémoire partagé

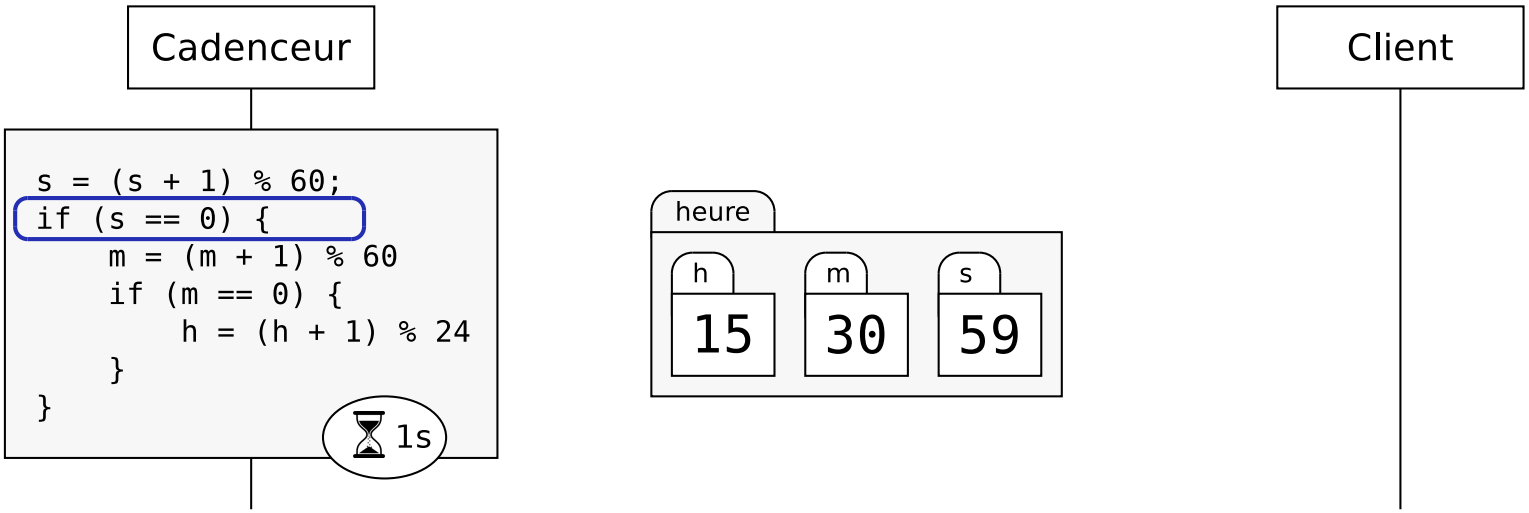
$\Rightarrow$ risques de corruption de données.



Accès concurrents

Exécutions concurrentes dans un espace mémoire partagé

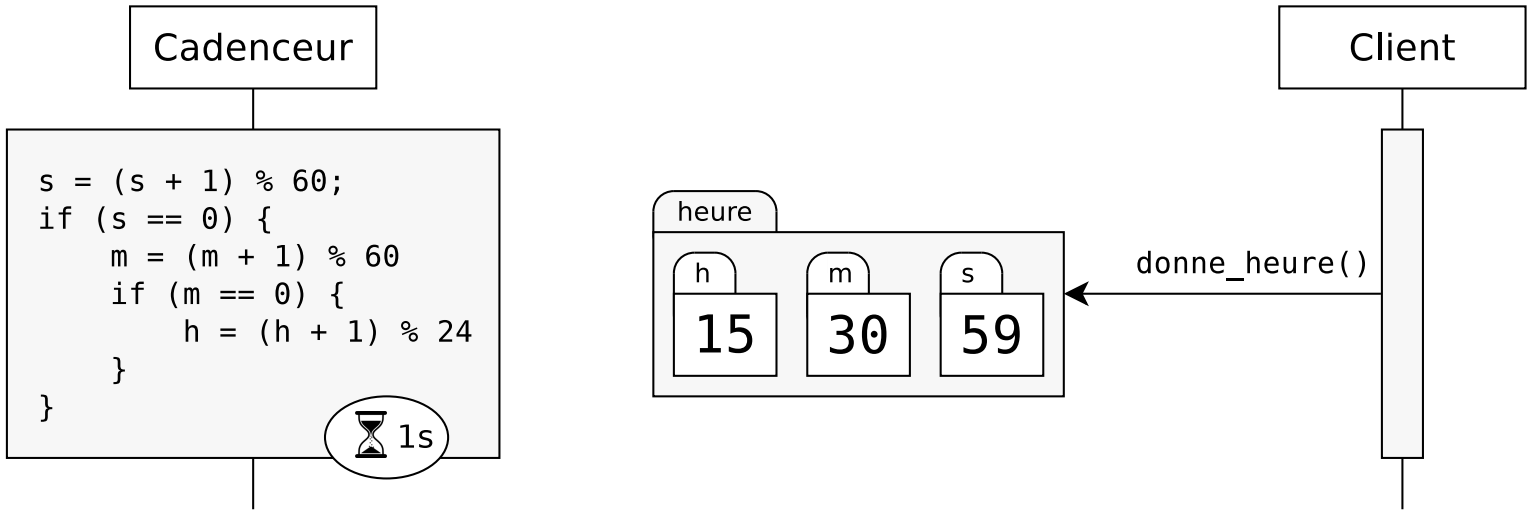
$\Rightarrow$ risques de corruption de données.



Accès concurrents

Exécutions concurrentes dans un espace mémoire partagé

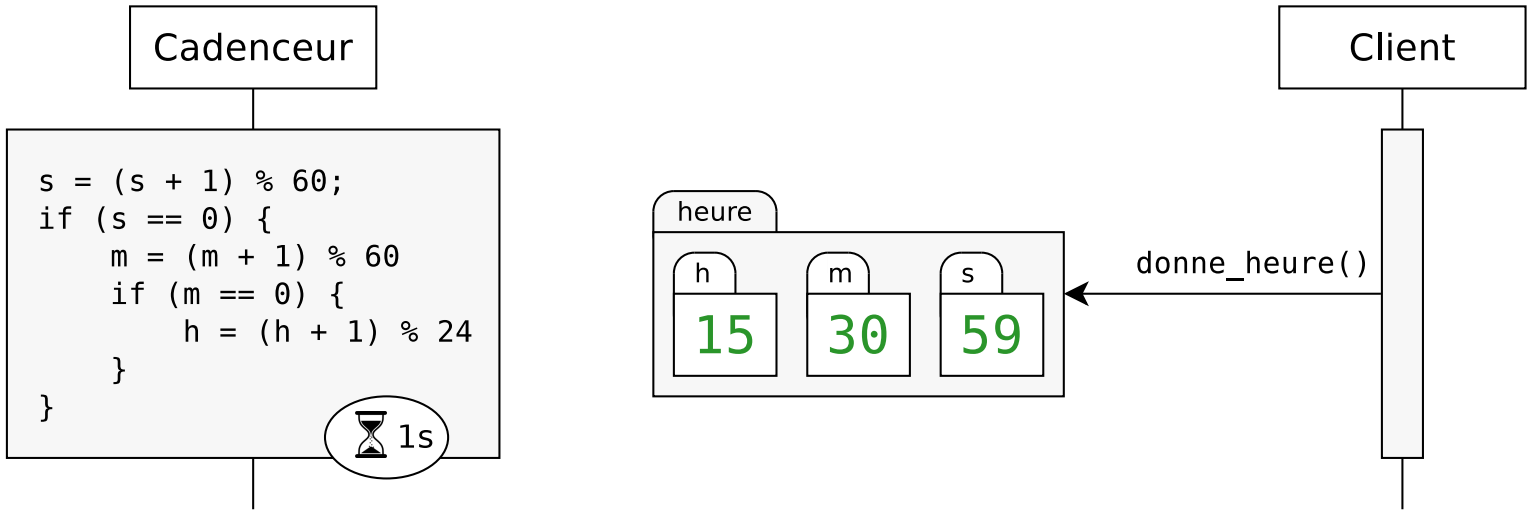
$\Rightarrow$ risques de corruption de données.



Accès concurrents

Exécutions concurrentes dans un espace mémoire partagé

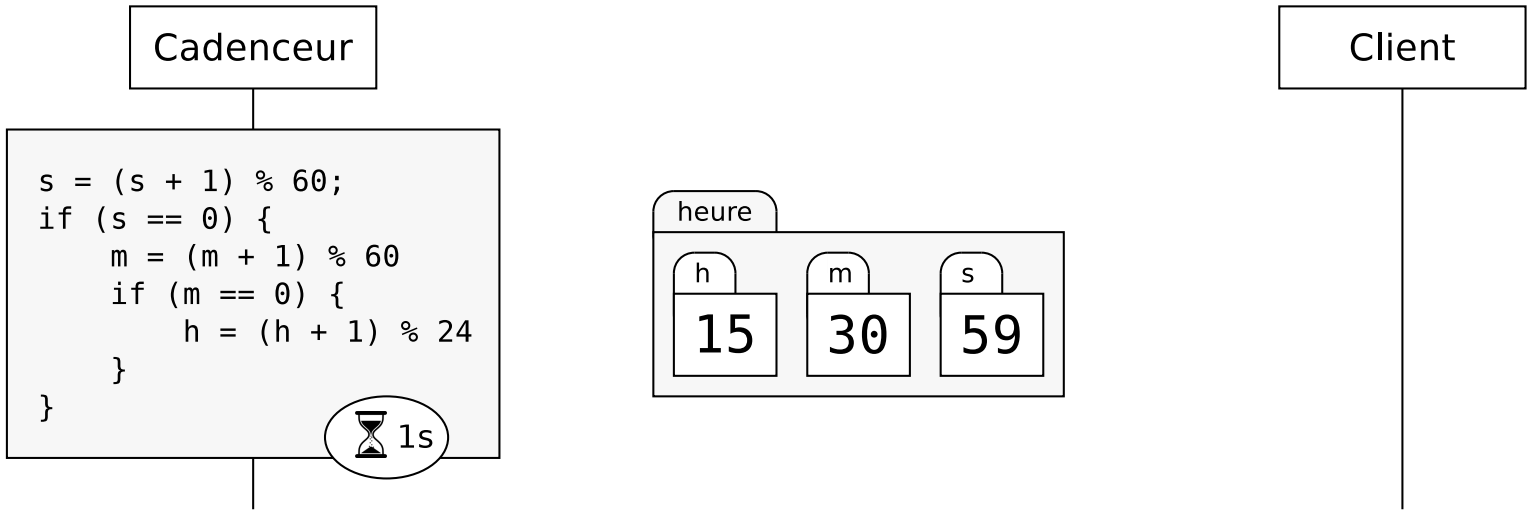
$\Rightarrow$ risques de corruption de données.



Accès concurrents

Exécutions concurrentes dans un espace mémoire partagé

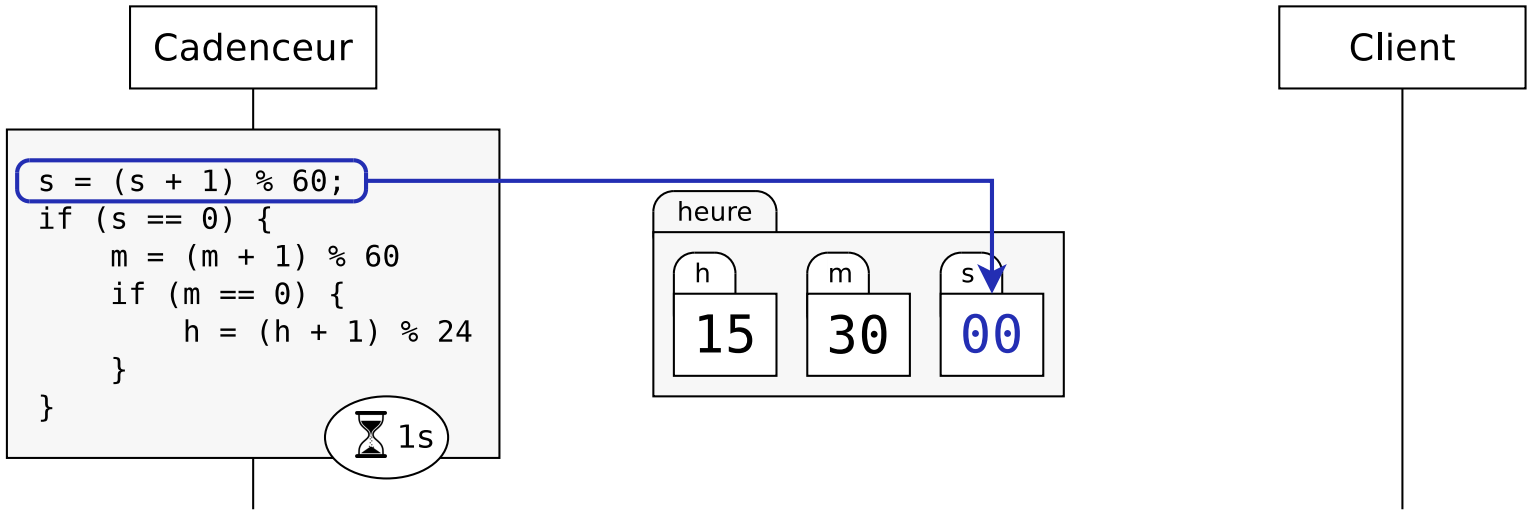
$\Rightarrow$ risques de corruption de données.



Accès concurrents

Exécutions concurrentes dans un espace mémoire partagé

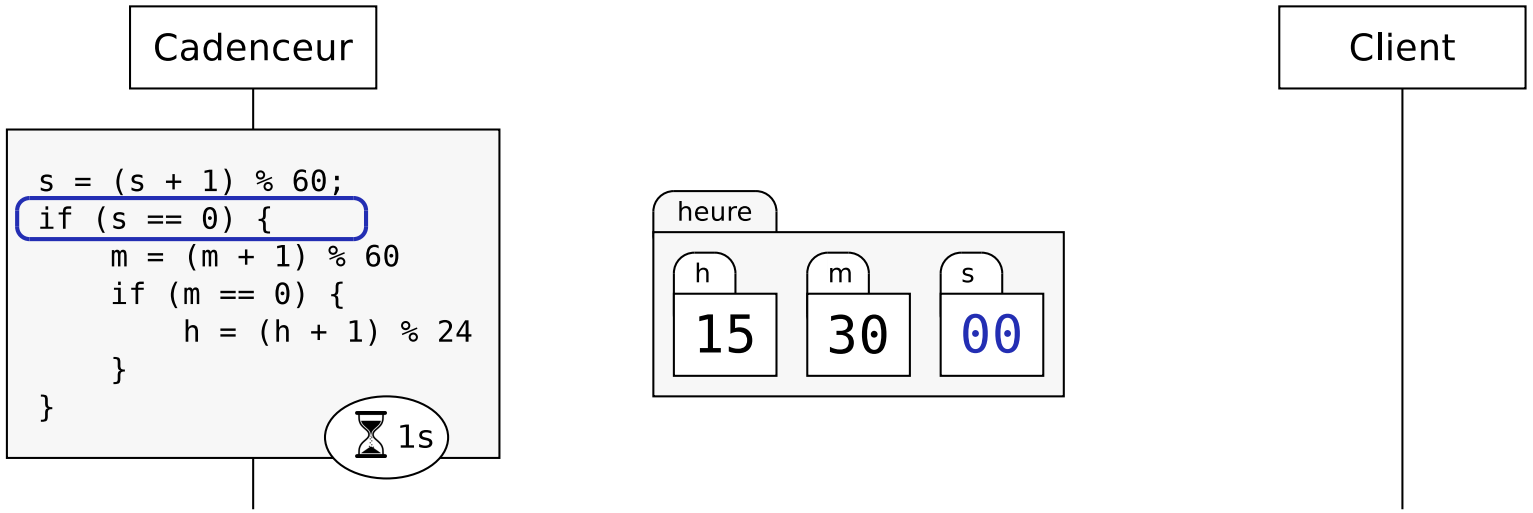
$\Rightarrow$ risques de corruption de données.



Accès concurrents

Exécutions concurrentes dans un espace mémoire partagé

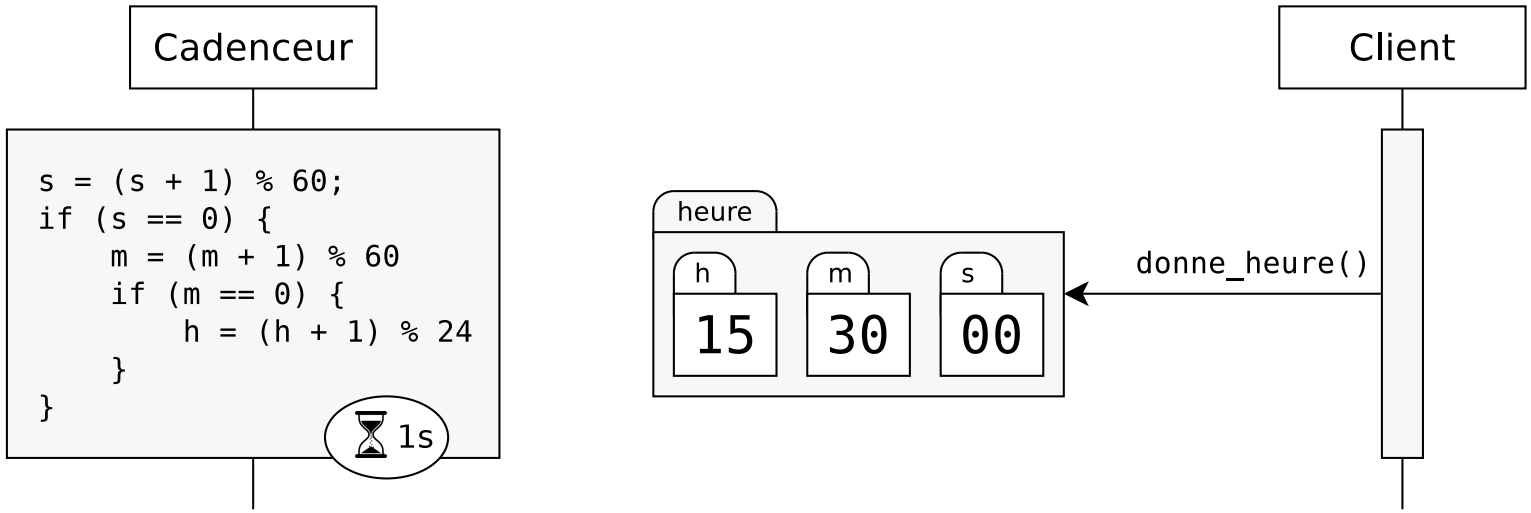
$\Rightarrow$ risques de corruption de données.



Accès concurrents

Exécutions concurrentes dans un espace mémoire partagé

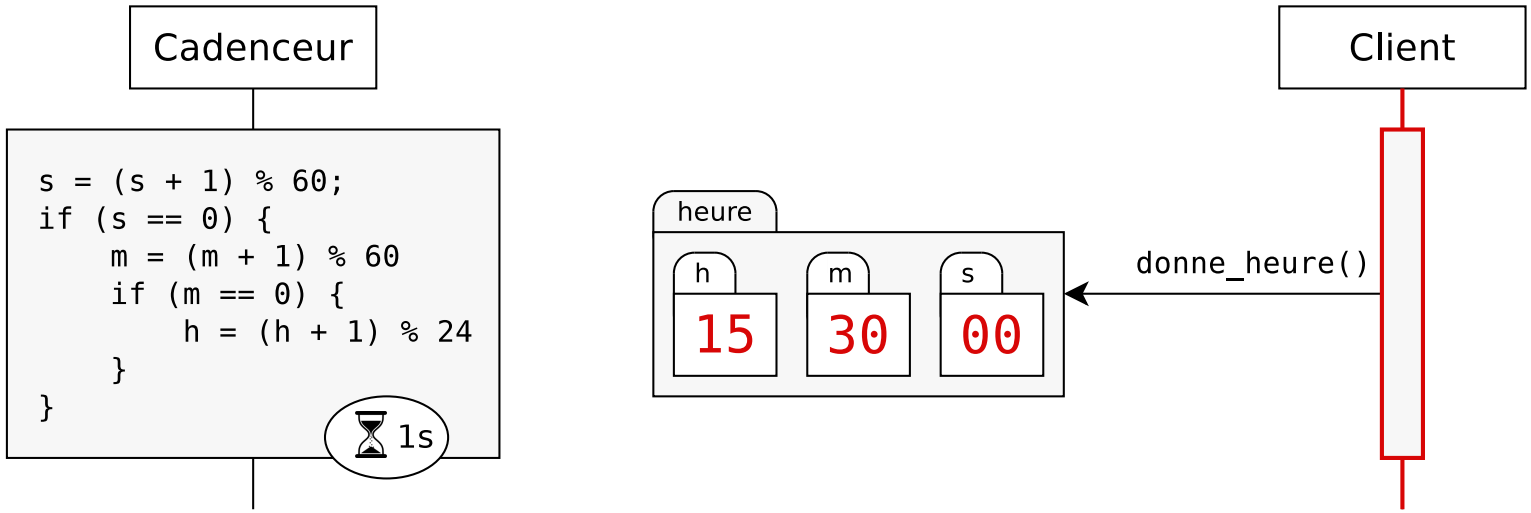
$\Rightarrow$ risques de corruption de données.



Accès concurrents

Exécutions concurrentes dans un espace mémoire partagé

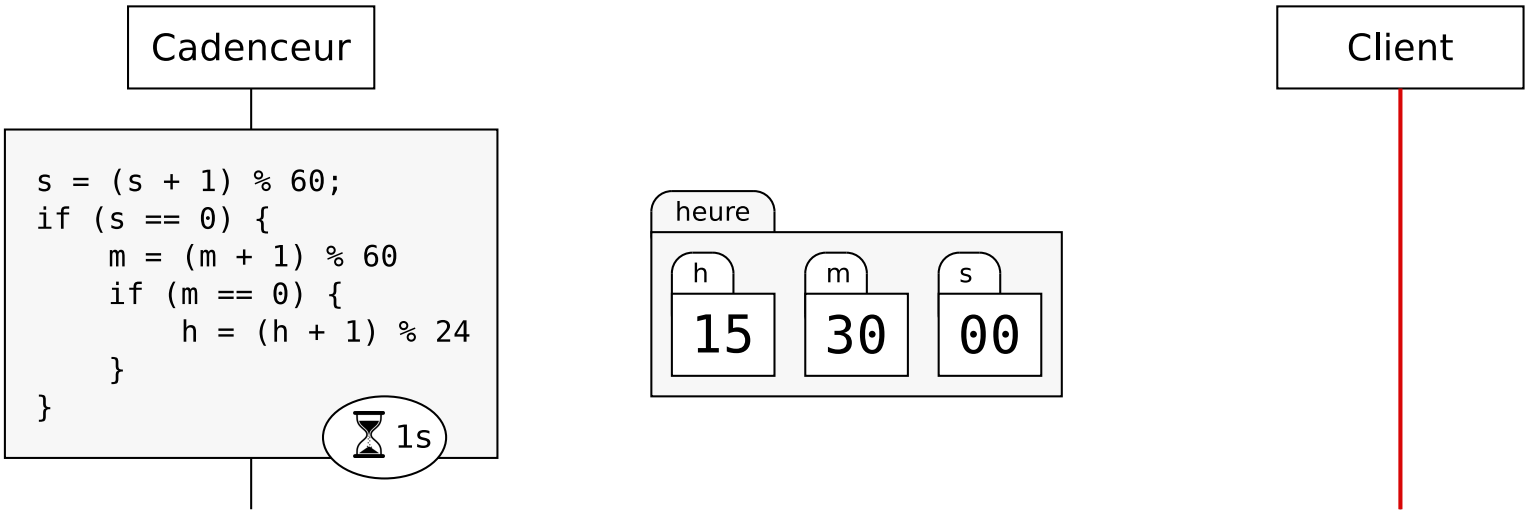
$\Rightarrow$ risques de corruption de données.



Accès concurrents

Exécutions concurrentes dans un espace mémoire partagé

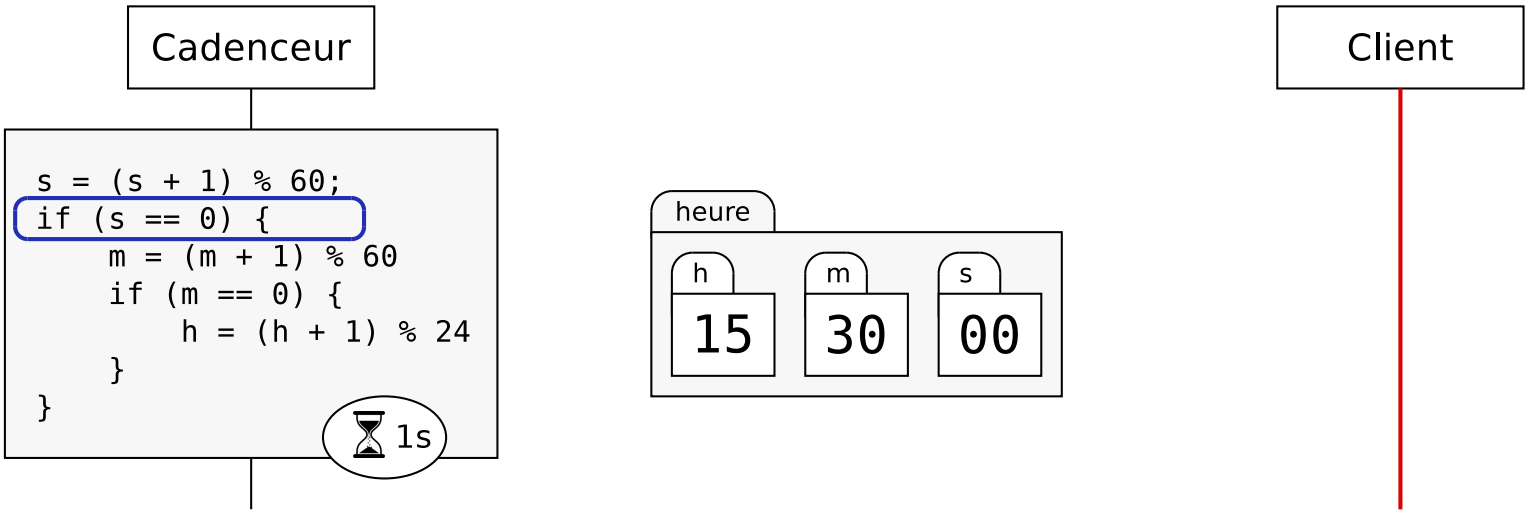
$\Rightarrow$ risques de corruption de données.



Accès concurrents

Exécutions concurrentes dans un espace mémoire partagé

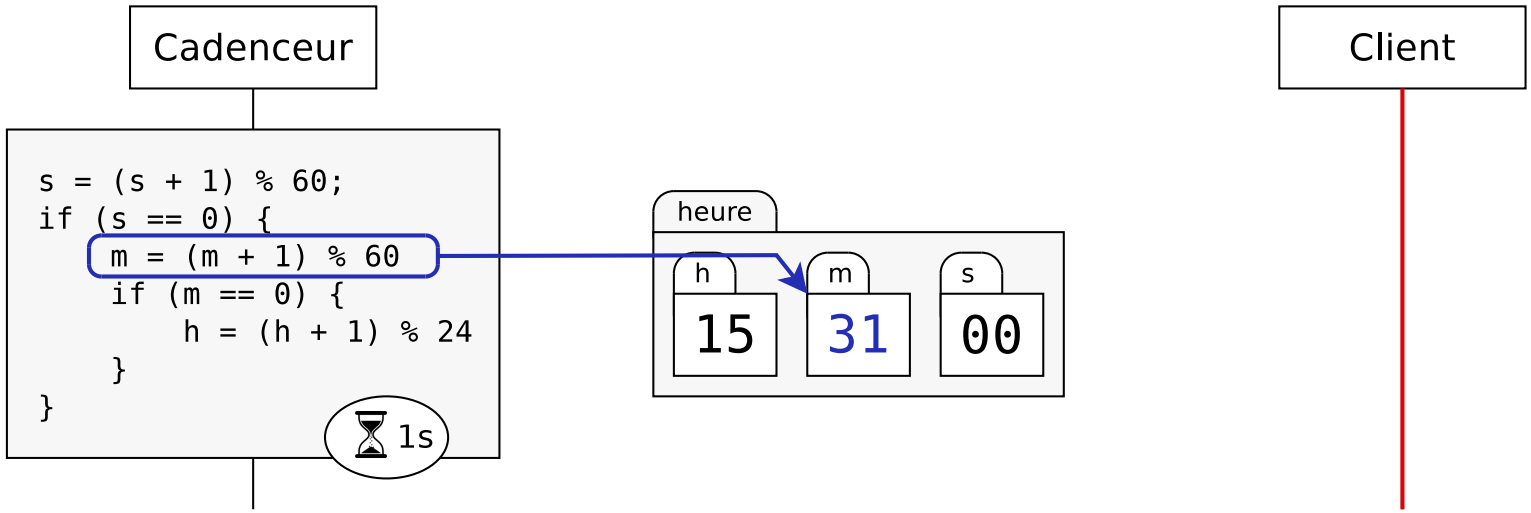
$\Rightarrow$ risques de corruption de données.



Accès concurrents

Exécutions concurrentes dans un espace mémoire partagé

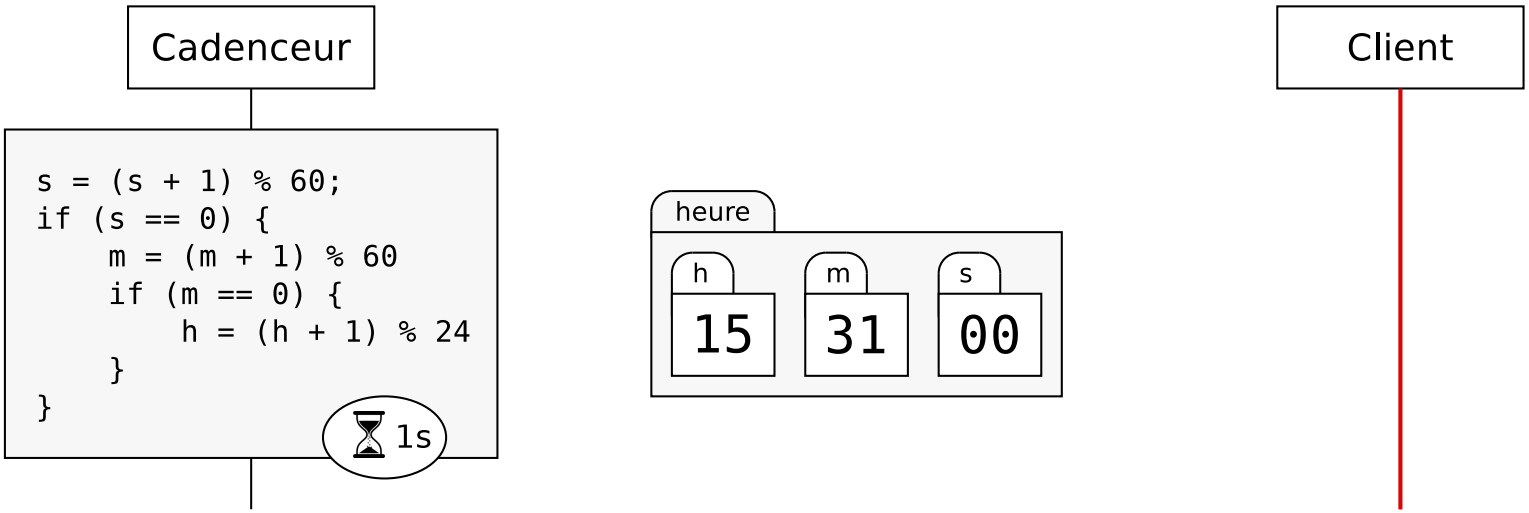
$\Rightarrow$ risques de corruption de données.



Accès concurrents

Exécutions concurrentes dans un espace mémoire partagé

$\Rightarrow$ risques de corruption de données.



Synchronisation des accès concurrents

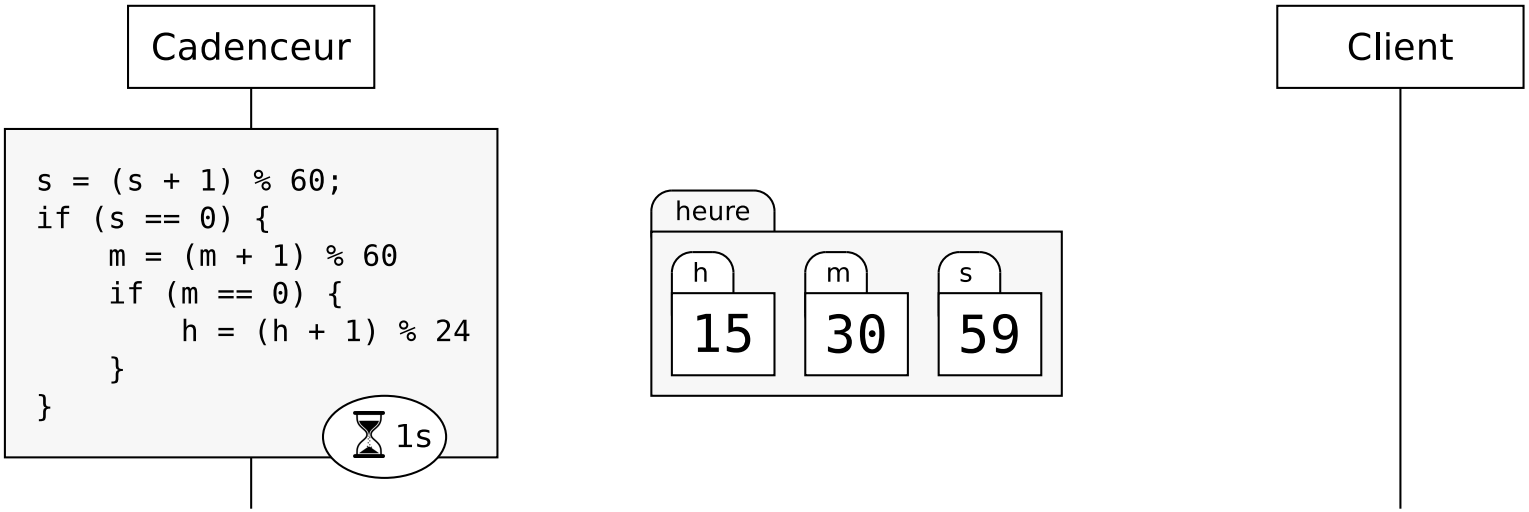
Ex : verrouillage des accès aux ressources critiques

(espace mémoire succeptible d'être accédé par différents threads)

Synchronisation des accès concurrents

Ex : verrouillage des accès aux ressources critiques

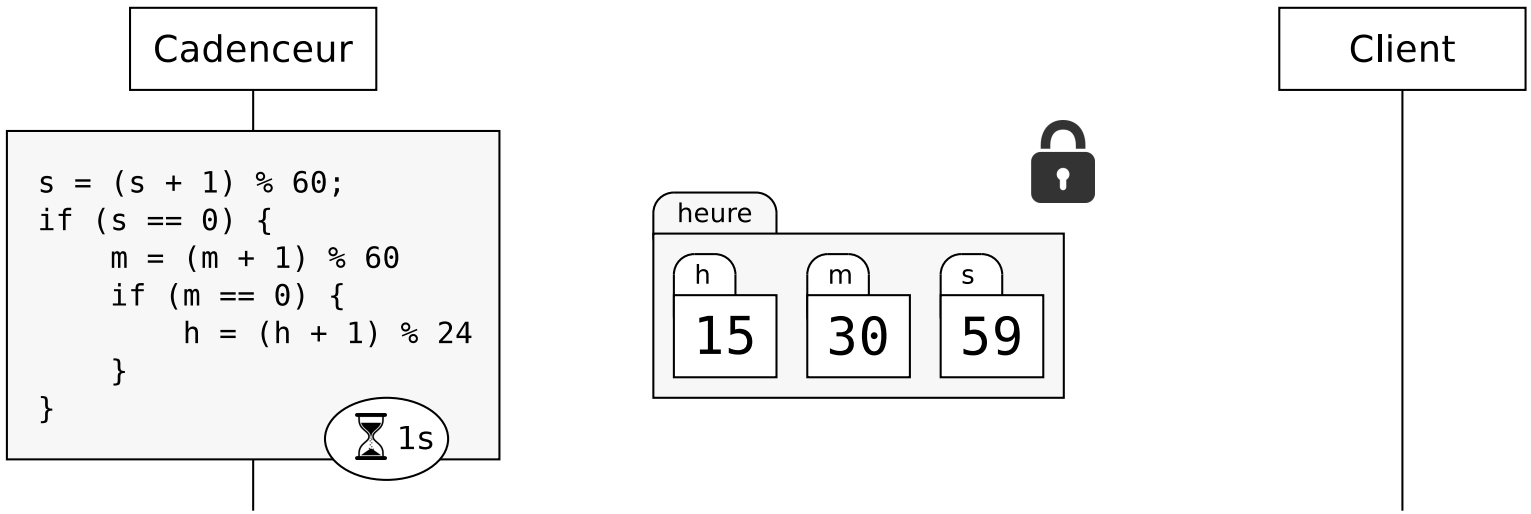
(espace mémoire succeptible d'être accédé par différents threads)



Synchronisation des accès concurrents

Ex : verrouillage des accès aux ressources critiques

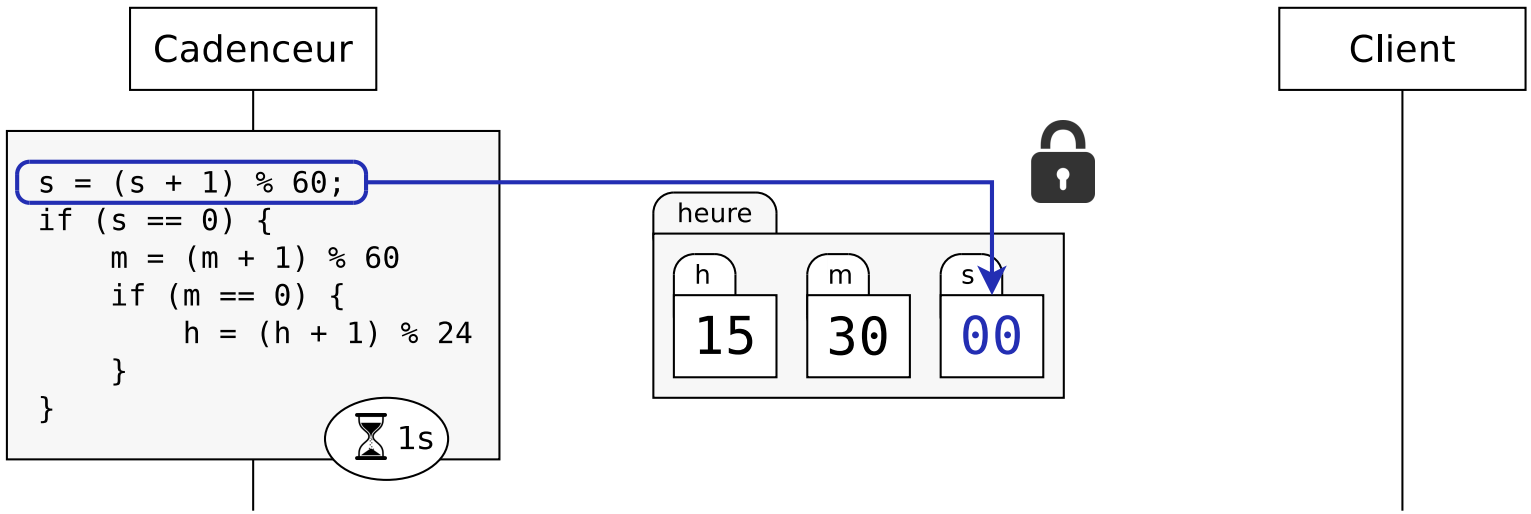
(espace mémoire succeptible d'être accédé par différents threads)



Synchronisation des accès concurrents

Ex : verrouillage des accès aux ressources critiques

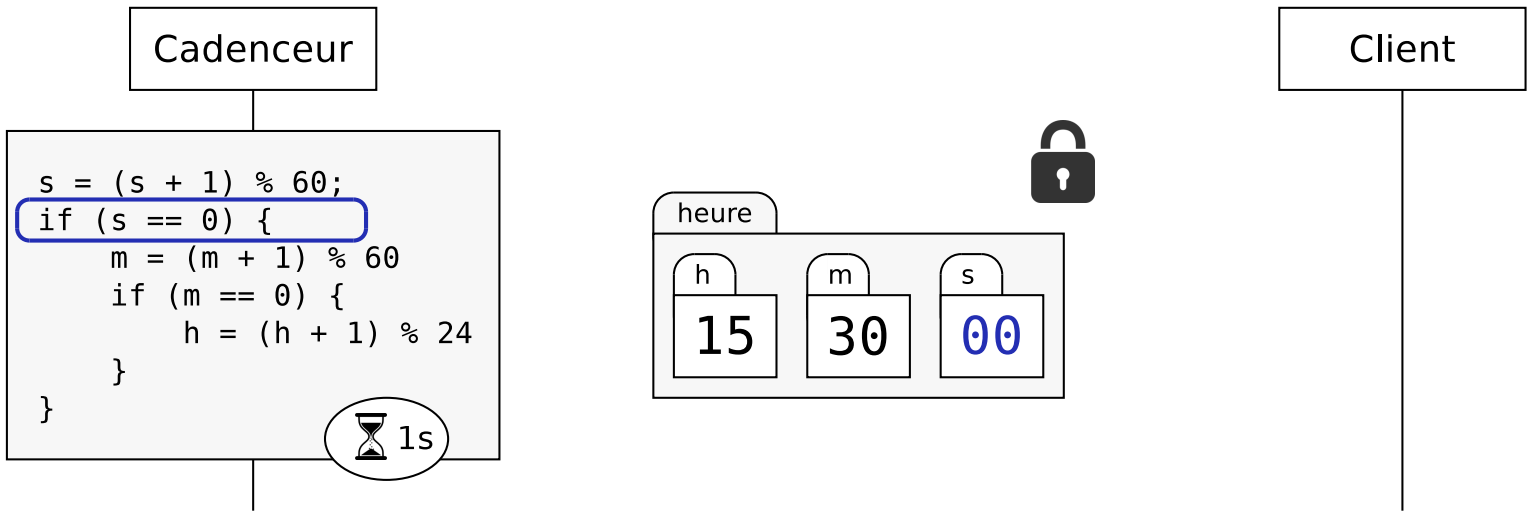
(espace mémoire succeptible d'être accédé par différents threads)



Synchronisation des accès concurrents

Ex : verrouillage des accès aux ressources critiques

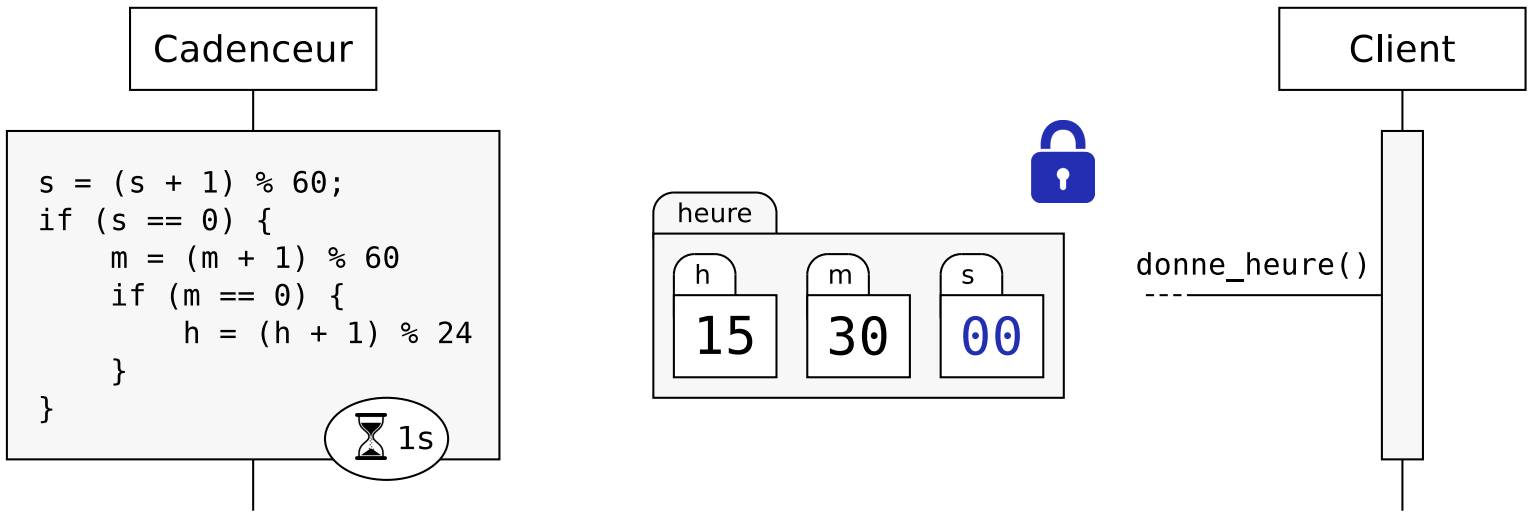
(espace mémoire succeptible d'être accédé par différents threads)



Synchronisation des accès concurrents

Ex : verrouillage des accès aux ressources critiques

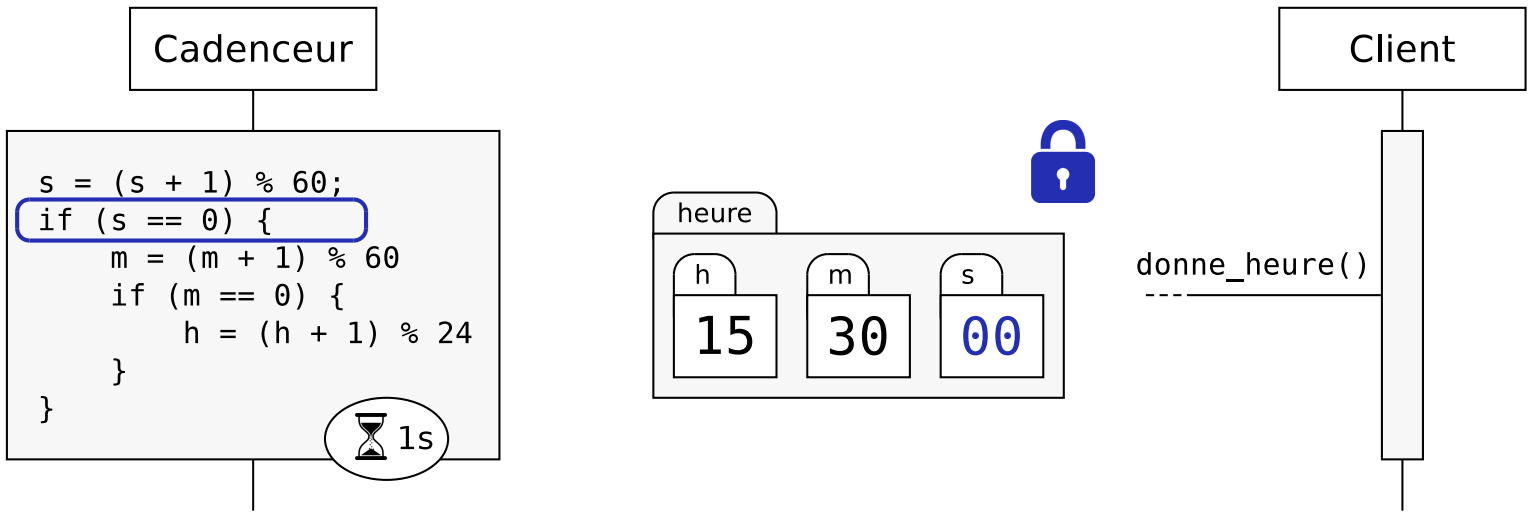
(espace mémoire succeptible d'être accédé par différents threads)



Synchronisation des accès concurrents

Ex : verrouillage des accès aux ressources critiques

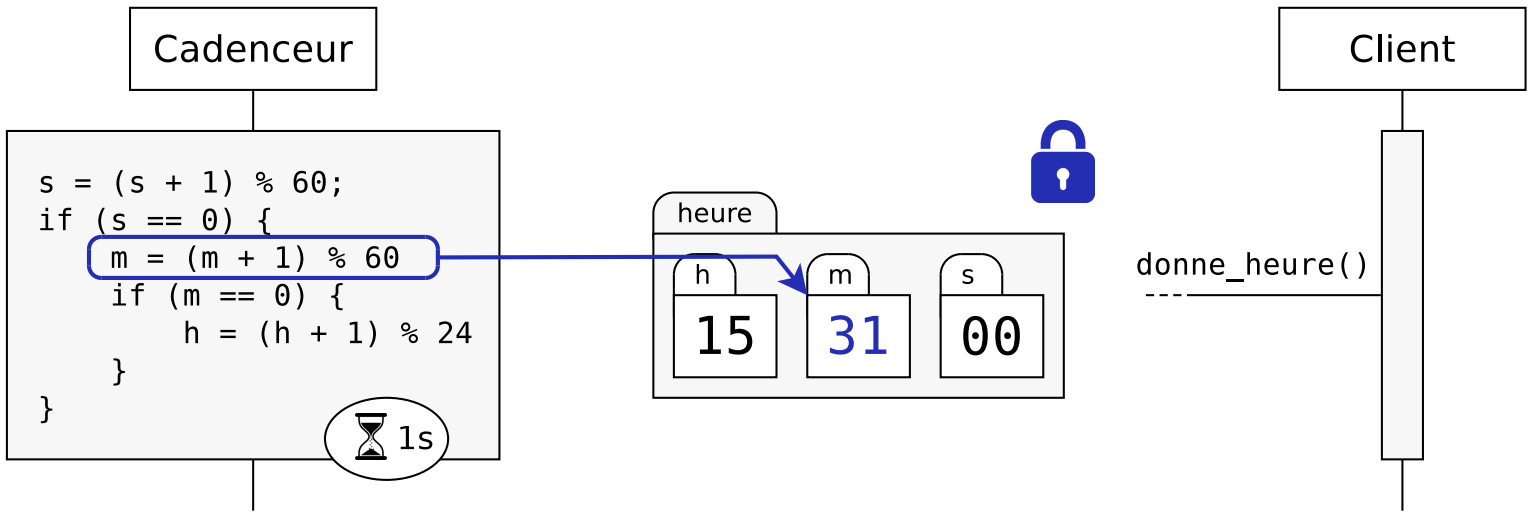
(espace mémoire succeptible d'être accédé par différents threads)



Synchronisation des accès concurrents

Ex : verrouillage des accès aux ressources critiques

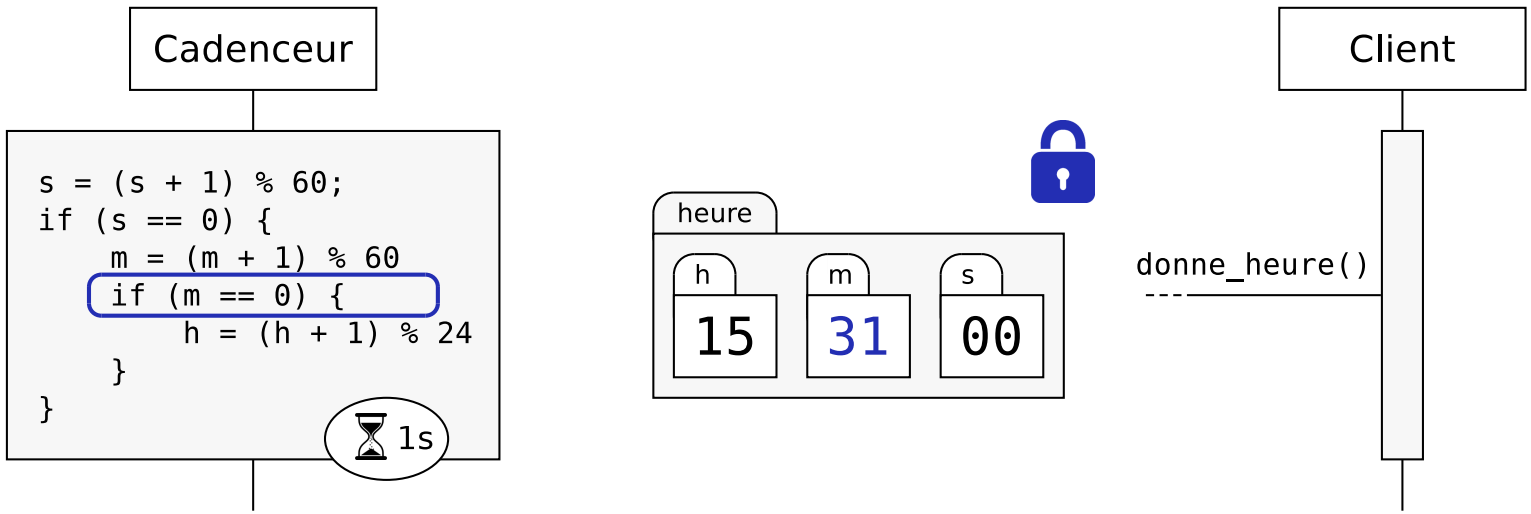
(espace mémoire succeptible d'être accédé par différents threads)



Synchronisation des accès concurrents

Ex : verrouillage des accès aux ressources critiques

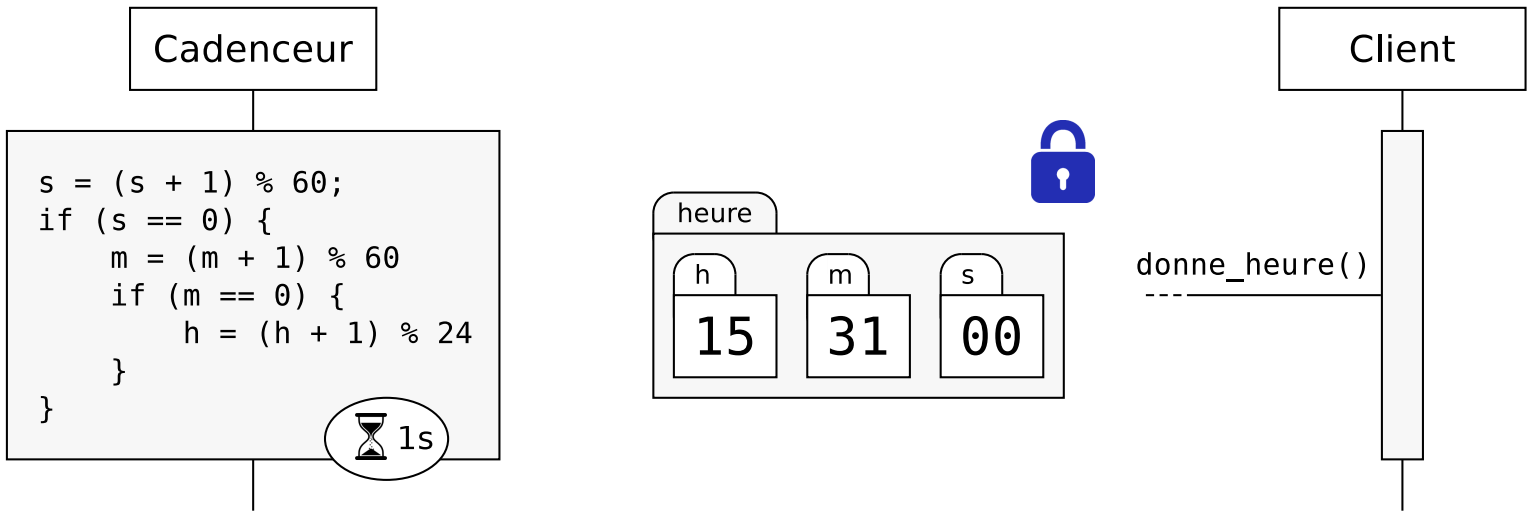
(espace mémoire succeptible d'être accédé par différents threads)



Synchronisation des accès concurrents

Ex : verrouillage des accès aux ressources critiques

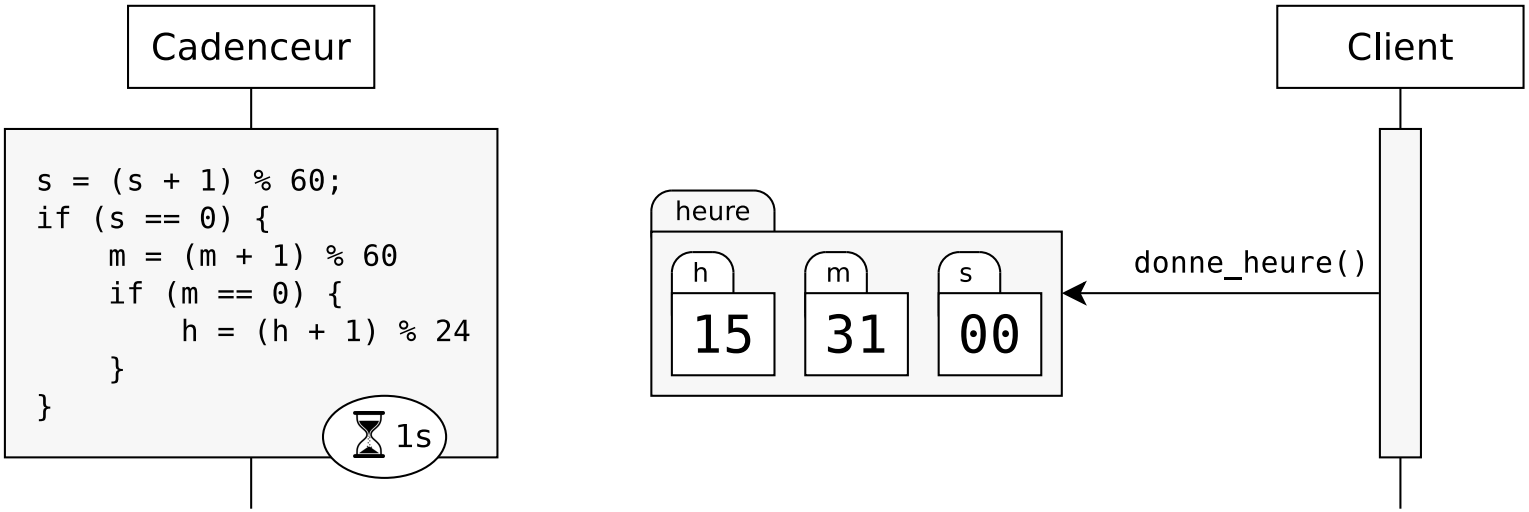
(espace mémoire succeptible d'être accédé par différents threads)



Synchronisation des accès concurrents

Ex : verrouillage des accès aux ressources critiques

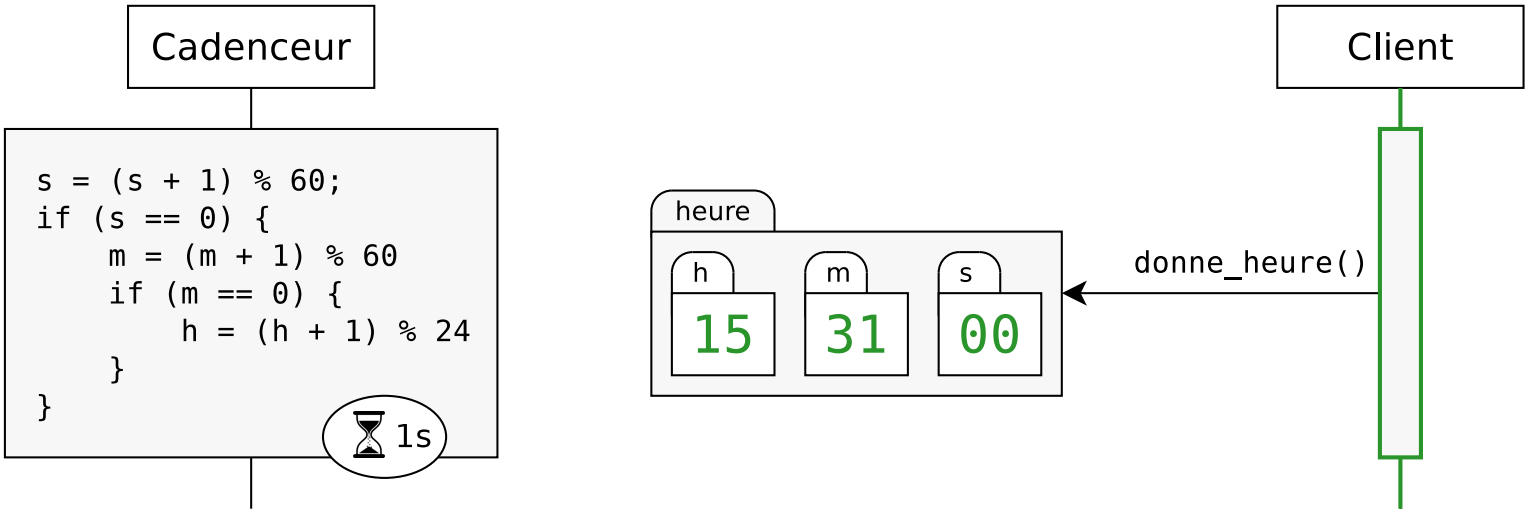
(espace mémoire succeptible d'être accédé par différents threads)



Synchronisation des accès concurrents

Ex : verrouillage des accès aux ressources critiques

(espace mémoire succeptible d'être accédé par différents threads)



ressource critique

entitée qui n'admet pas plusieurs accès simultanés

section critique

section de code qui implique une ressource critique

atomique

exécution indivisible (sans préemption)

réentrance

capacité d'admettre plusieurs exécutions simultanées

thread safe

capacité d'admettre des exécutions concurrentes

race condition

risque de résultats différents selon l'ordre des exécutions concurrentes

ressource critique

entitée qui n'admet pas plusieurs accès simultanés

section critique

section de code qui implique une ressource critique

atomique

exécution indivisible (sans préemption)

réentrance

capacité d'admettre plusieurs exécutions simultanées

thread safe

capacité d'admettre des exécutions concurrentes

race condition

risque de résultats différents selon l'ordre des exécutions concurrentes

ressource critique

entitée qui n'admet pas plusieurs accès simultanés

section critique

section de code qui implique une ressource critique

atomique

exécution indivisible (sans préemption)

réentrance

capacité d'admettre plusieurs exécutions simultanées

thread safe

capacité d'admettre des exécutions concurrentes

race condition

risque de résultats différents selon l'ordre des exécutions concurrentes

ressource critique

entitée qui n'admet pas plusieurs accès simultanés

section critique

section de code qui implique une ressource critique

atomique

exécution indivisible (sans préemption)

réentrance

capacité d'admettre plusieurs exécutions simultanées

thread safe

capacité d'admettre des exécutions concurrentes

race condition

risque de résultats différents selon l'ordre des exécutions concurrentes

ressource critique

entitée qui n'admet pas plusieurs accès simultanés

section critique

section de code qui implique une ressource critique

atomique

exécution indivisible (sans préemption)

réentrance

capacité d'admettre plusieurs exécutions simultanées

thread safe

capacité d'admettre des exécutions concurrentes

race condition

risque de résultats différents selon l'ordre des exécutions concurrentes

ressource critique

entitée qui n'admet pas plusieurs accès simultanés

section critique

section de code qui implique une ressource critique

atomique

exécution indivisible (sans préemption)

réentrance

capacité d'admettre plusieurs exécutions simultanées

thread safe

capacité d'admettre des exécutions concurrentes

race condition

risque de résultats différents selon l'ordre des exécutions concurrentes

Synchronisation des accès concurrents

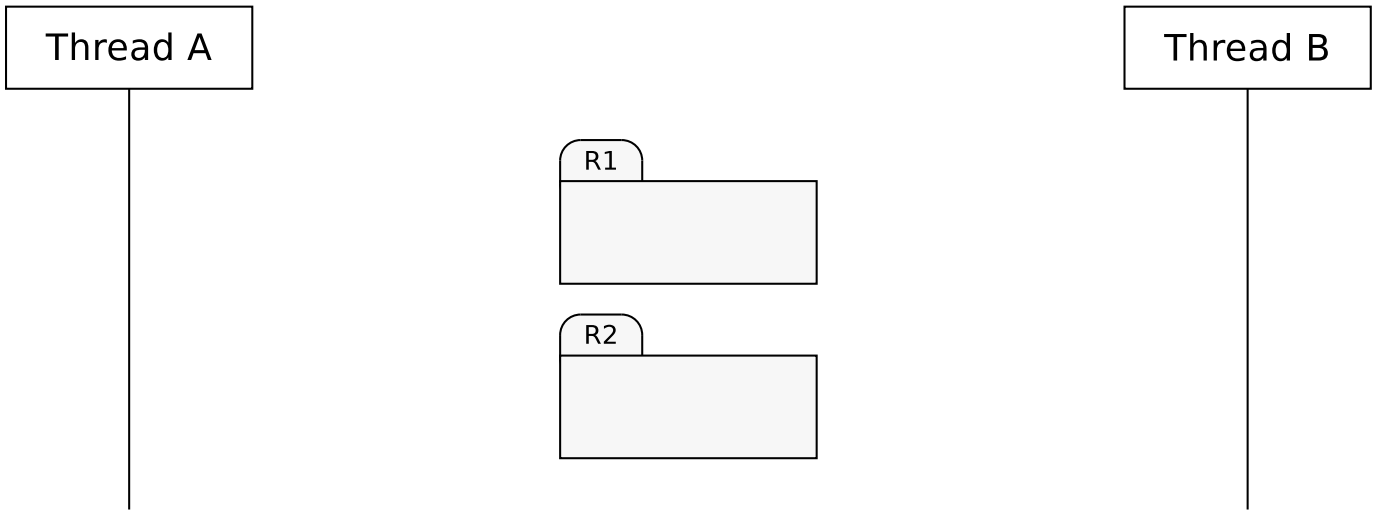
Ex : verrouillage des accès aux ressources critiques

$\Rightarrow$ risque d'interblocage (*deadlock*).

Synchronisation des accès concurrents

Ex : verrouillage des accès aux ressources critiques

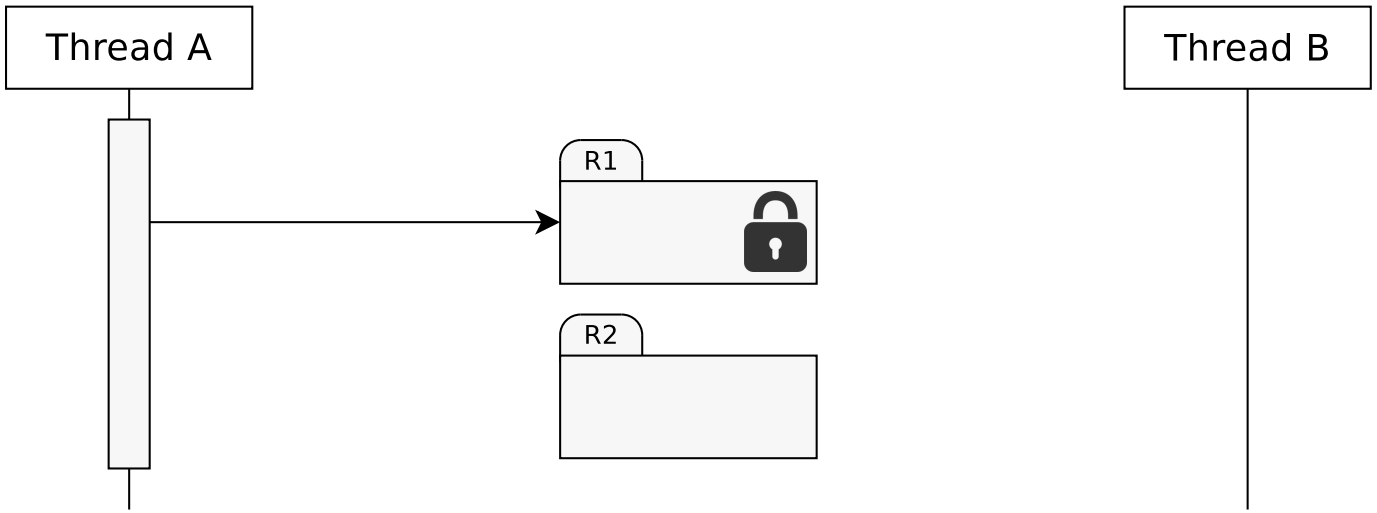
$\Rightarrow$ risque d'interblocage (*deadlock*).



Synchronisation des accès concurrents

Ex : verrouillage des accès aux ressources critiques

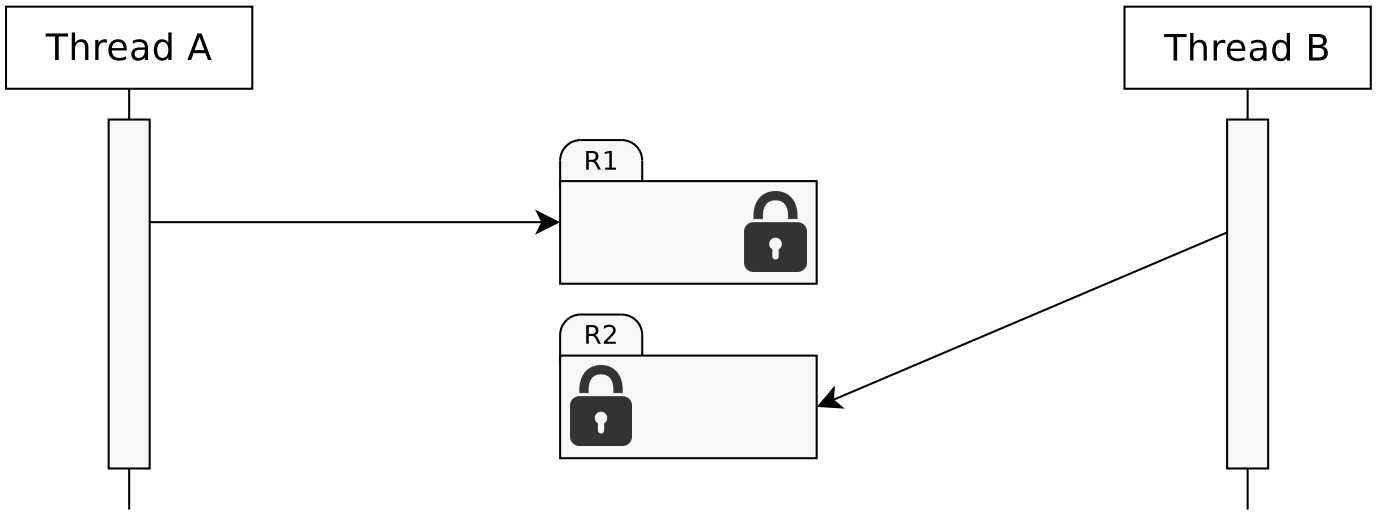
$\Rightarrow$ risque d'interblocage (*deadlock*).



Synchronisation des accès concurrents

Ex : verrouillage des accès aux ressources critiques

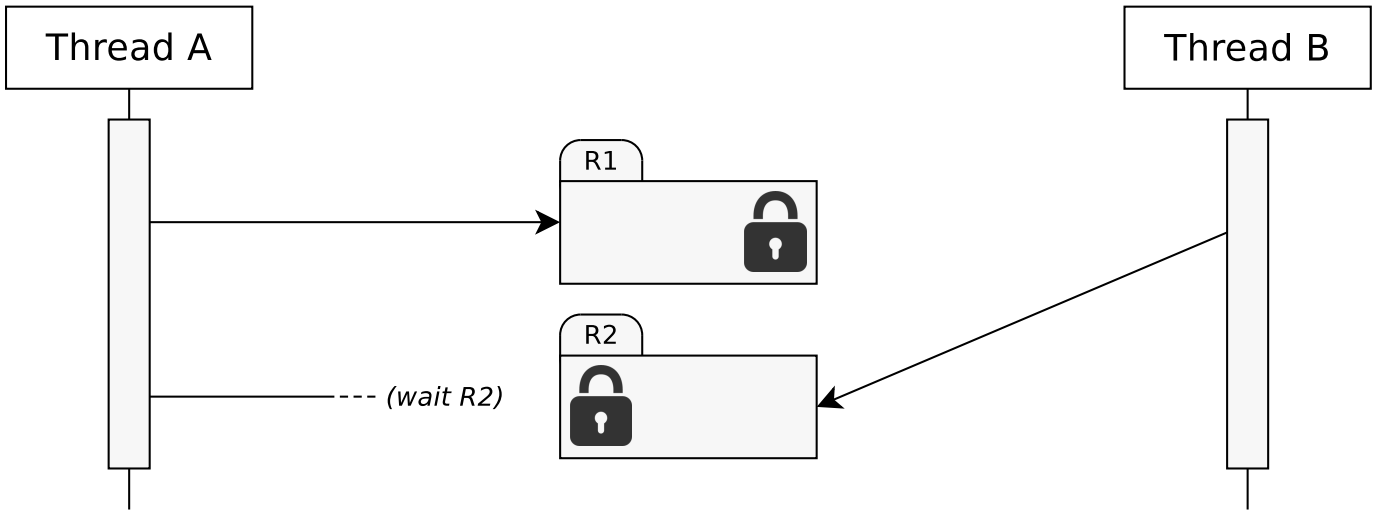
$\Rightarrow$ risque d'interblocage (*deadlock*).



Synchronisation des accès concurrents

Ex : verrouillage des accès aux ressources critiques

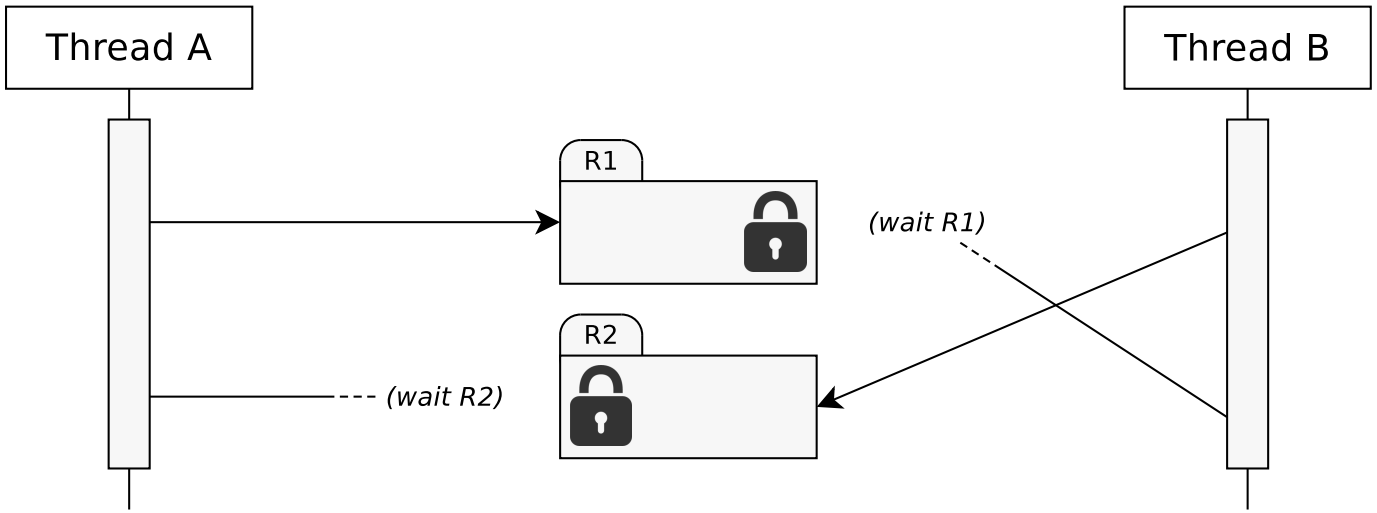
$\Rightarrow$ risque d'interblocage (*deadlock*).



Synchronisation des accès concurrents

Ex : verrouillage des accès aux ressources critiques

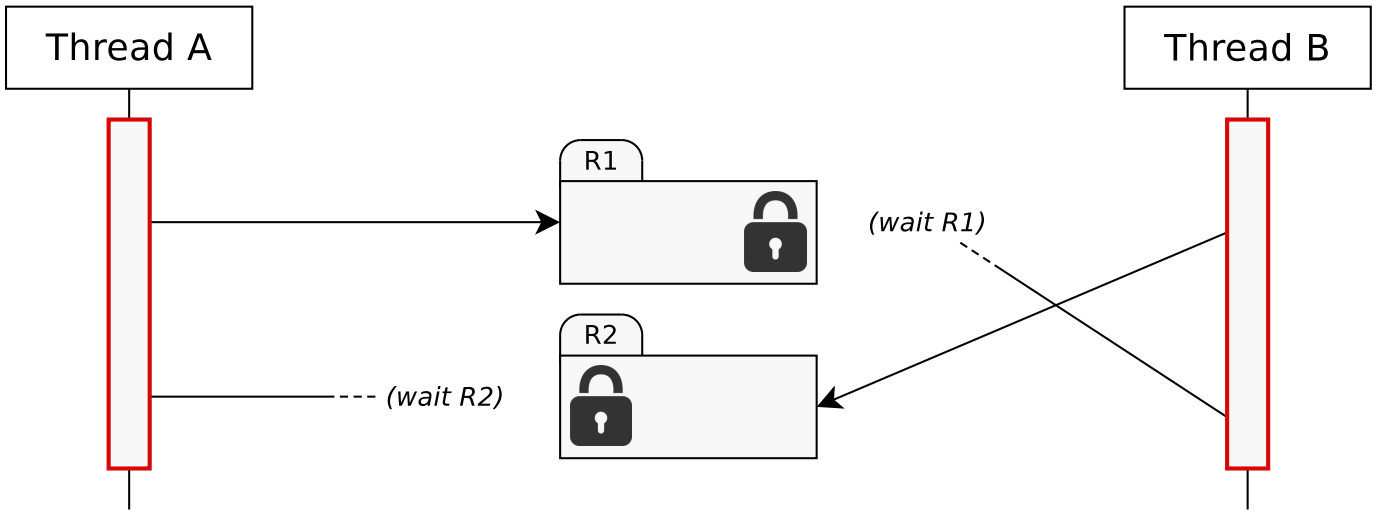
$\Rightarrow$ risque d'interblocage (*deadlock*).



Synchronisation des accès concurrents

Ex : verrouillage des accès aux ressources critiques

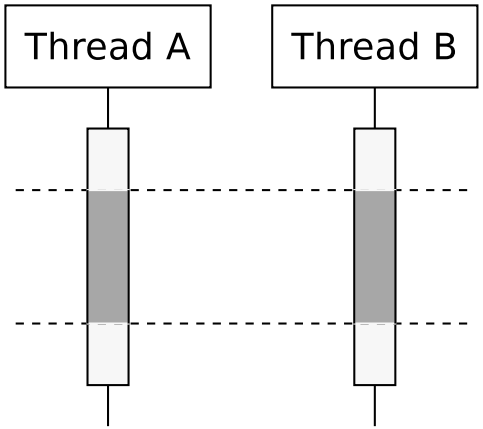
$\Rightarrow$ risque d'interblocage (*deadlock*).



Plan

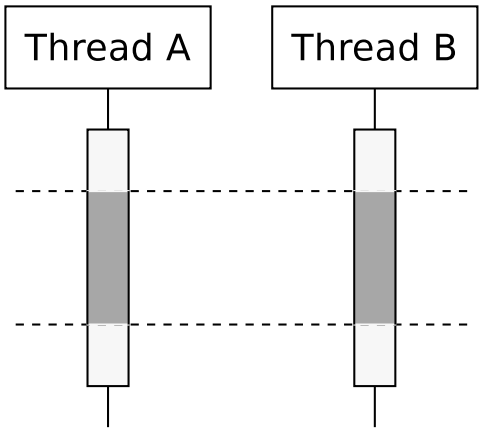
Besoins en synchronisation

Besoins en synchronisation

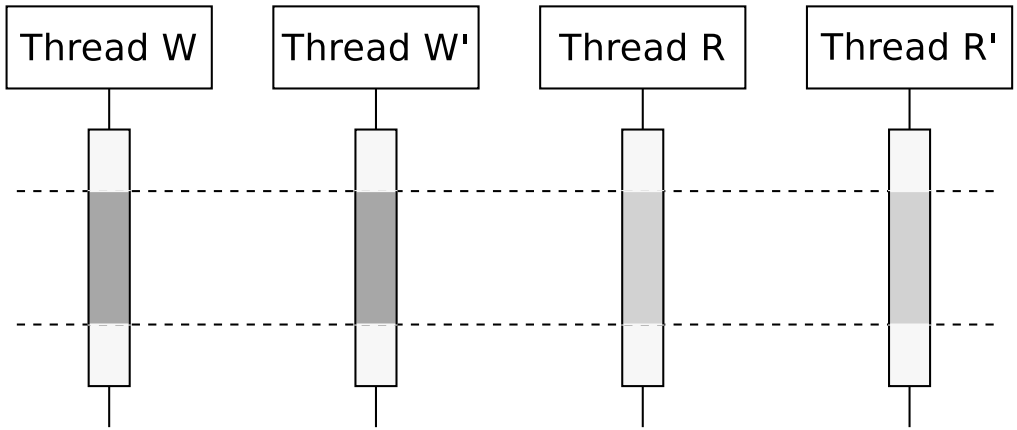


Section Critique

Besoins en synchronisation

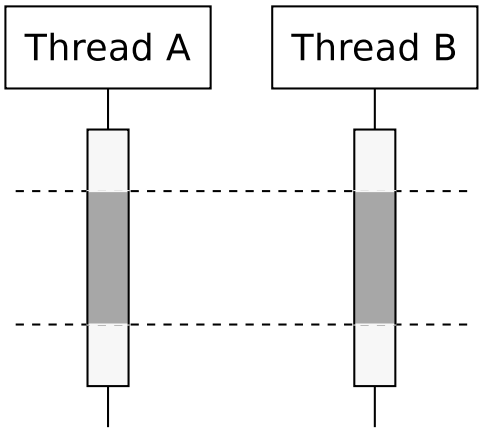


Section Critique

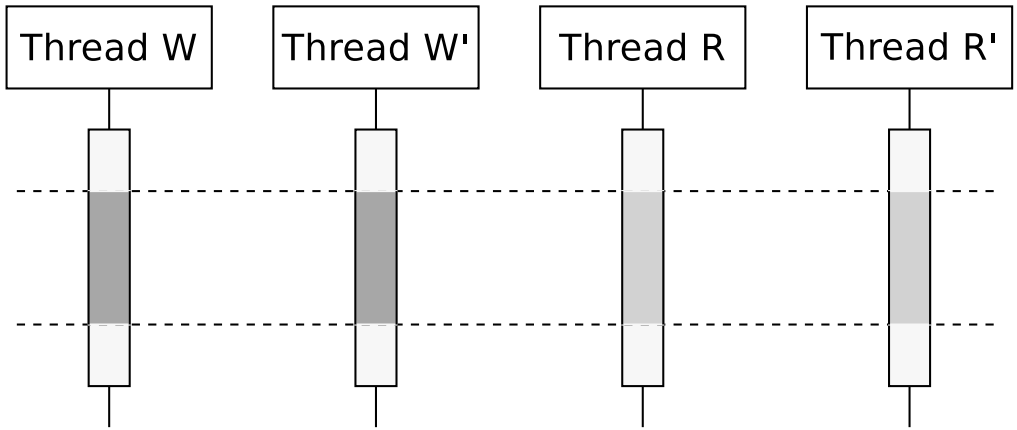


Lecteurs/Écrivains

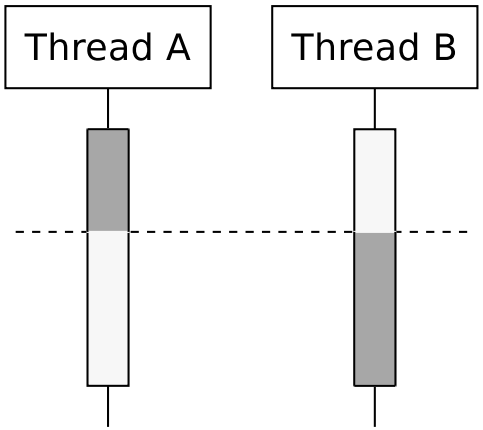
Besoins en synchronisation



Section Critique

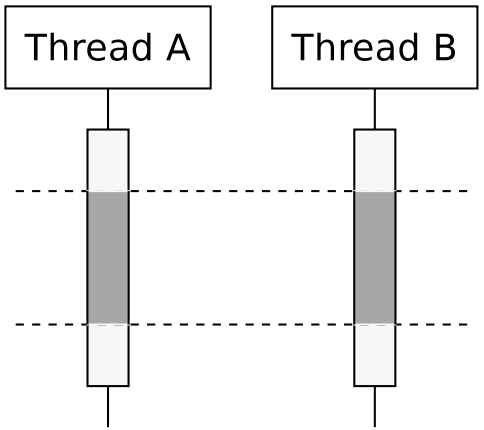


Lecteurs/Écrivains

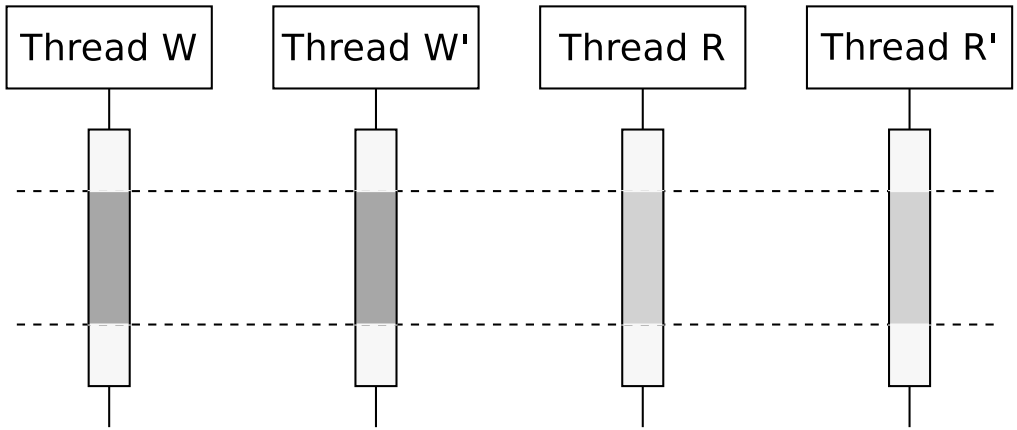


Condition

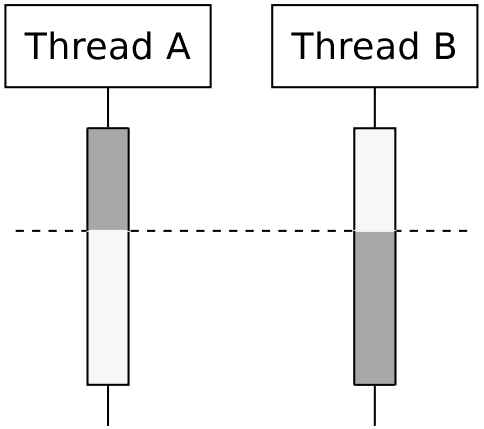
Besoins en synchronisation



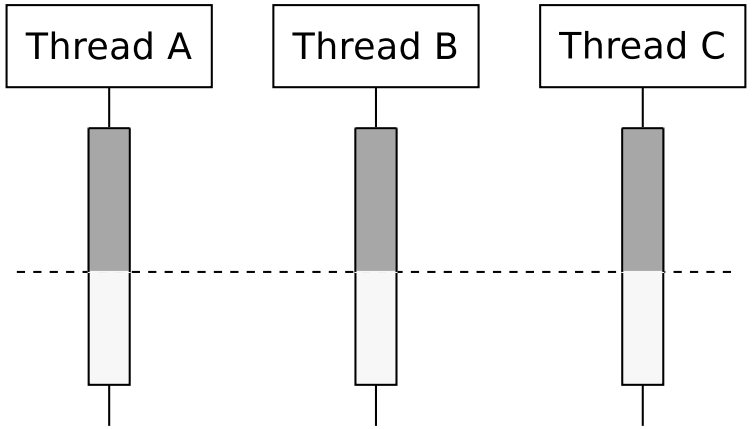
Section Critique



Lecteurs/Écrivains

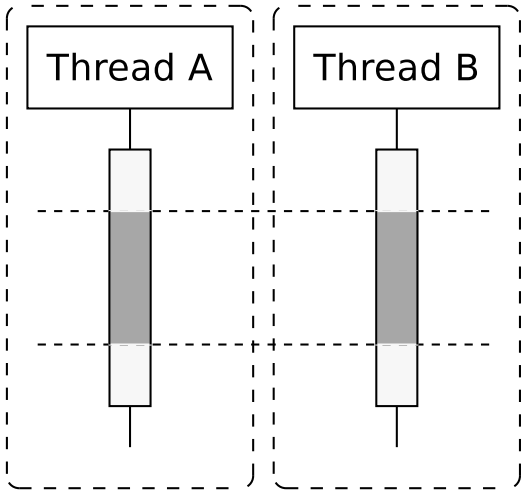


Condition

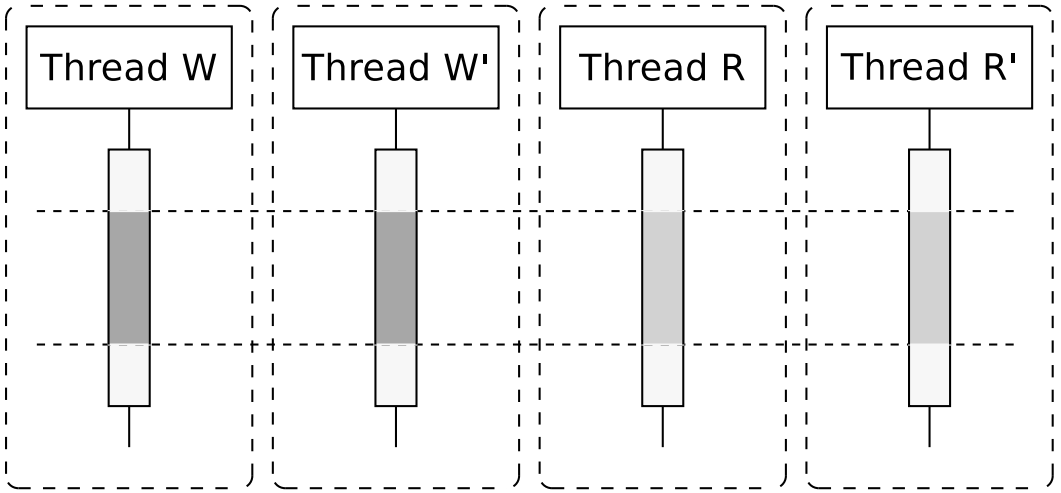


Rendez-Vous

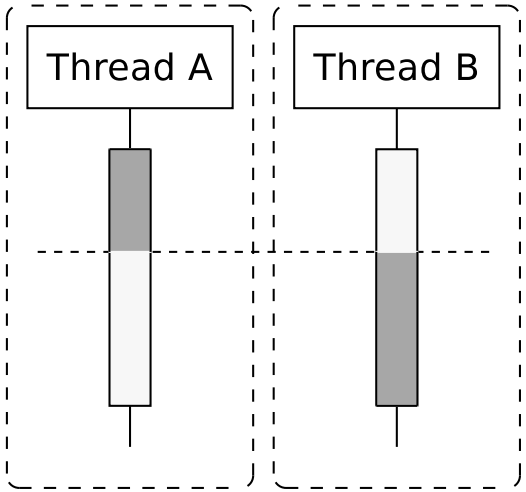
Besoins en synchronisation



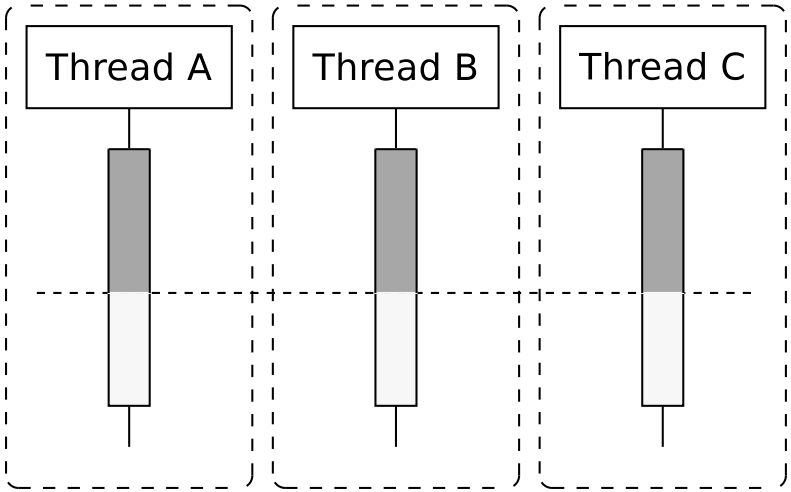
Section Critique



Lecteurs/Écrivains



Condition

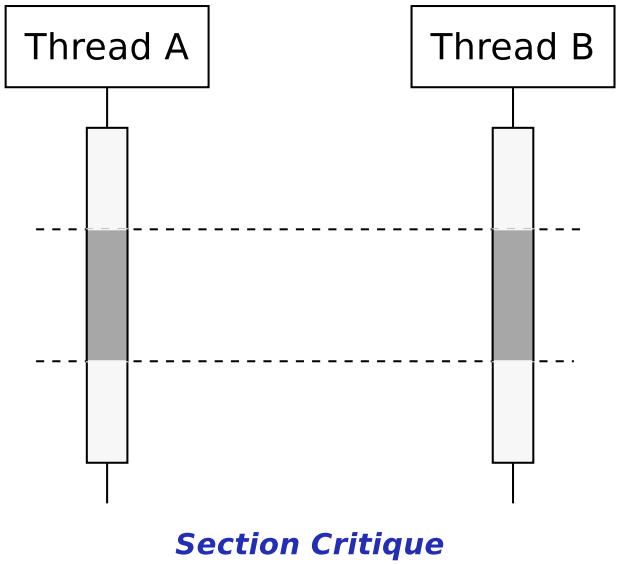


Rendez-Vous

Inter Processus ?

Programmation d'une synchronisation

- Exemple accès à une section critique :

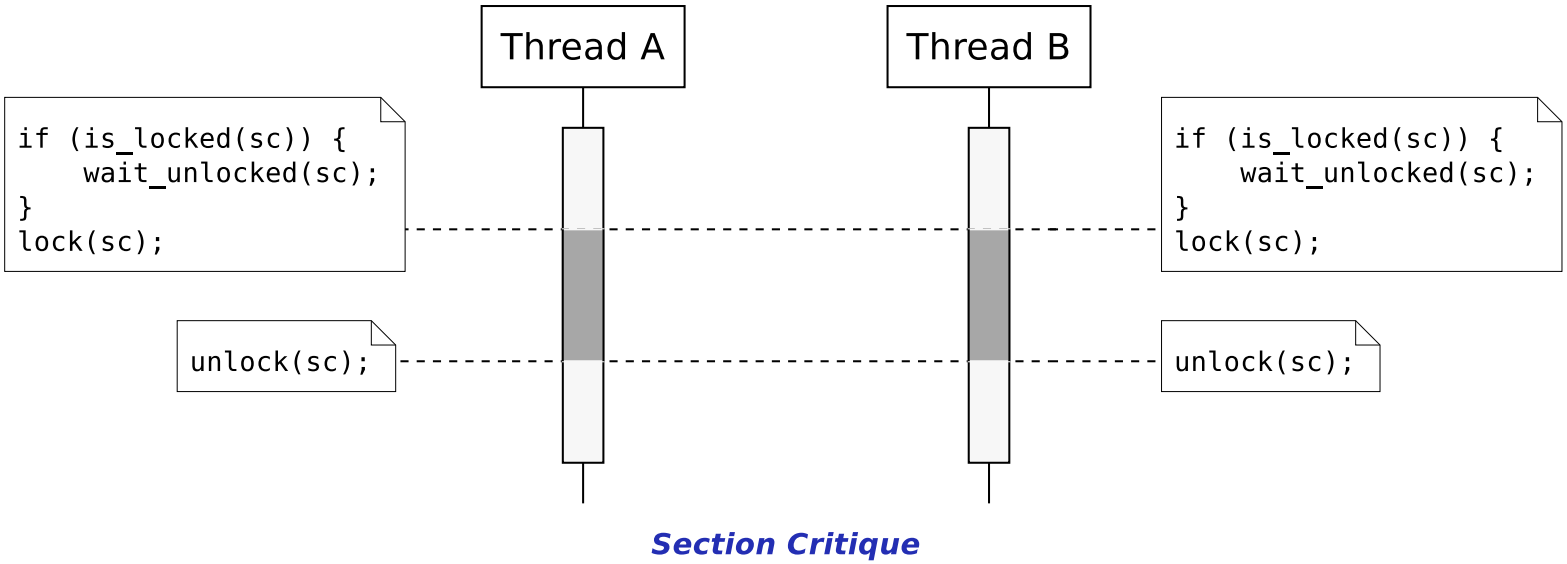


⇒ besoin d'instructions « *test and set* » atomiques

⇒ utilisation des mécanismes de la plateforme d'exécution

Programmation d'une synchronisation

- Exemple accès à une section critique :

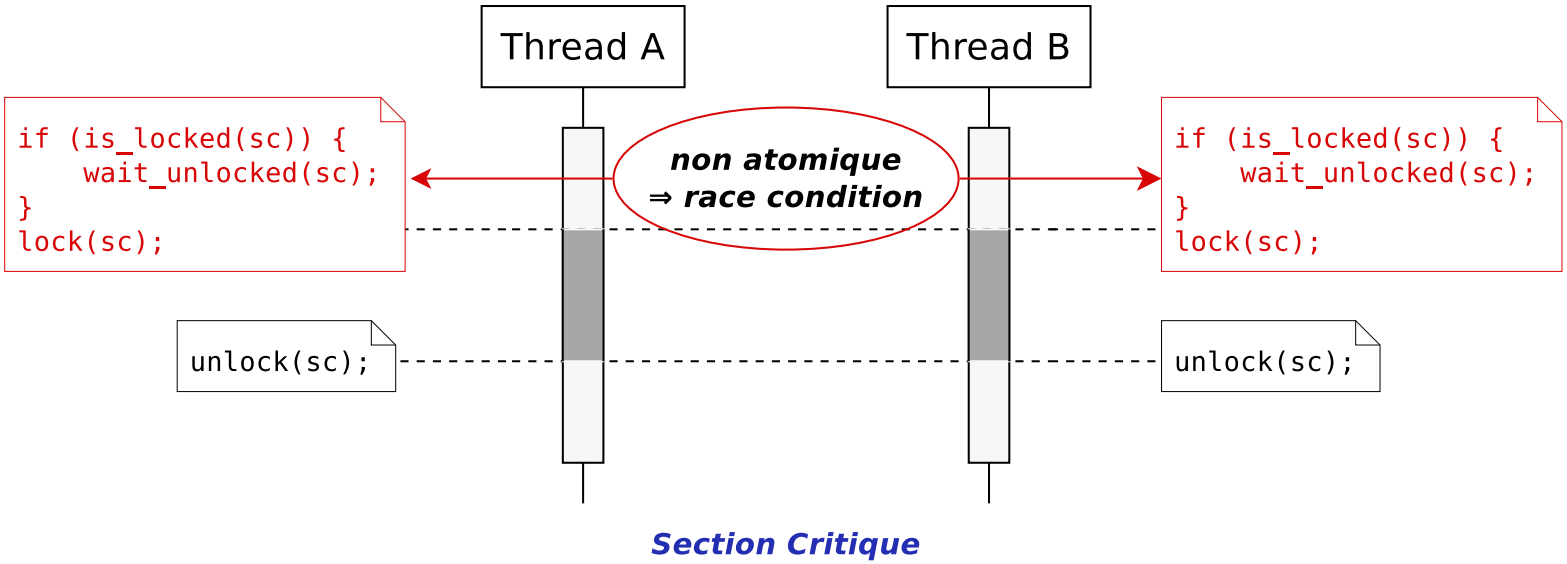


⇒ besoin d'instructions « *test and set* » atomiques

⇒ utilisation des mécanismes de la plateforme d'exécution

Programmation d'une synchronisation

- Exemple accès à une section critique :

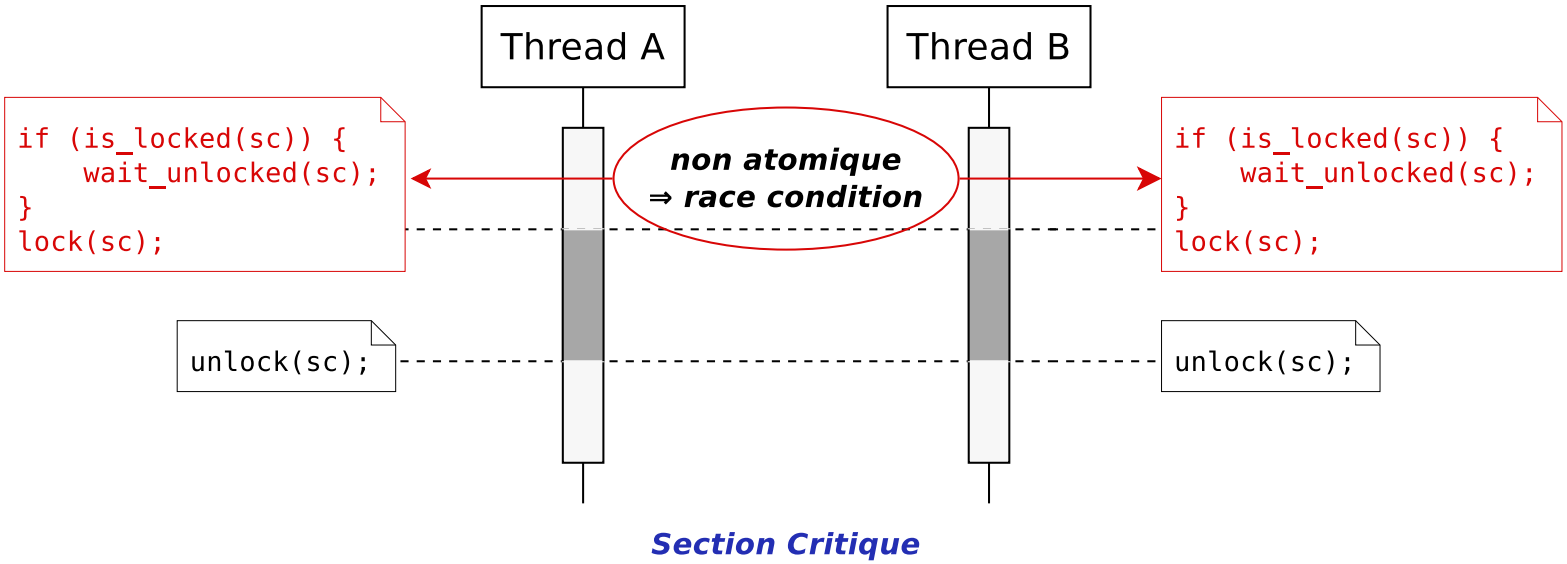


⇒ besoin d'instructions « *test and set* » atomiques

⇒ utilisation des mécanismes de la plateforme d'exécution

Programmation d'une synchronisation

- Exemple accès à une section critique :

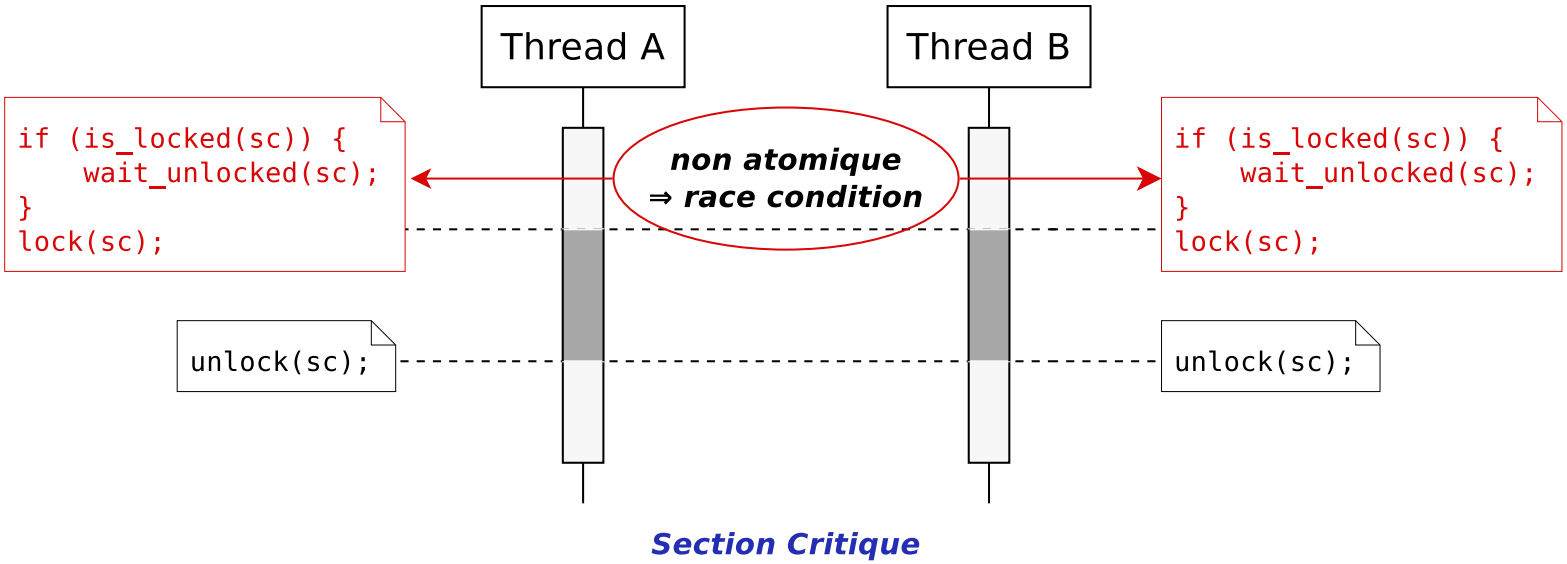


⇒ besoin d'instructions ≪ *test and set* ≫ atomiques

⇒ utilisation des mécanismes de la plateforme d'exécution

Programmation d'une synchronisation

- Exemple accès à une section critique :



⇒ besoin d'instructions ≪ *test and set* ≫ atomiques

⇒ utilisation des mécanismes de la plateforme d'exécution

Mécanismes de synchronisation POSIX

Threads (même espace mémoire) :

- verrous d'exclusion mutuelle (*mutex*)
- verrous de lecture/écriture (*rwlock*)
- variables condition (*cond*)
- barrières de synchronisation (*barrier*)

Inter-Processus :

- sémaphores (*sem*) (*)

(*) IPC (*Inter Process Communication*)

Mécanismes de synchronisation POSIX

Threads (même espace mémoire) :

- verrous d'exclusion mutuelle (*mutex*)
- verrous de lecture/écriture (*rwlock*)
- variables condition (*cond*)
- barrières de synchronisation (*barrier*)

Inter-Processus :

- sémaphores (*sem*) (*)

(*) IPC (*Inter Process Communication*)

Verrous d'exclusion mutuelle

Système POSIX :

- **mutex** : *mutual exclusion*

→ verrou libre ou verrouillé (tenu par un thread)

→ tenu par un seul thread à la fois

→ blocage sur demande de verrouillage si déjà tenu, jusqu'à libération

Programmation :

- type d'un mutex : `pthread_mutex_t`

Initialisation :

- `pthread_mutex_t mut = PTHREAD_MUTEX_INITIALIZER`
-
```
int pthread_mutex_init(
        pthread_mutex_t * mut,
        pthread_mutexattr_t * attr)
```

→ `attr = NULL` ⇒ valeurs par défaut

Verrous d'exclusion mutuelle

Système POSIX :

- **mutex** : *mutual exclusion*

→ verrou libre ou verrouillé (tenu par un thread)

→ tenu par un seul thread à la fois

→ blocage sur demande de verrouillage si déjà tenu, jusqu'à libération

Programmation :

- type d'un mutex : `pthread_mutex_t`

Initialisation :

- `pthread_mutex_t mut = PTHREAD_MUTEX_INITIALIZER`
-
```
int pthread_mutex_init(
        pthread_mutex_t * mut,
        pthread_mutexattr_t * attr)
```

→ `attr = NULL` ⇒ valeurs par défaut

Verrous d'exclusion mutuelle

Système POSIX :

- **mutex** : *mutual exclusion*

→ verrou libre ou verrouillé (tenu par un thread)

→ tenu par un seul thread à la fois

→ blocage sur demande de verrouillage si déjà tenu, jusqu'à libération

Programmation :

- type d'un mutex : `pthread_mutex_t`

Initialisation :

- `pthread_mutex_t mut = PTHREAD_MUTEX_INITIALIZER`
- ```
  int pthread_mutex_init(
              pthread_mutex_t * mut,
              pthread_mutexattr_t * attr)
  ```

→ `attr = NULL` ⇒ valeurs par défaut
  ```

Verrous d'exclusion mutuelle

Verrouillage :

- `int pthread_mutex_lock(pthread_mutex_t * mut)`

→ appel bloquant si mutex déjà tenu

- `int pthread_mutex_trylock(pthread_mutex_t * mut)`

→ non-bloquant, erreur `EBUSY` si mutex déjà tenu

- `int pthread_mutex_unlock(pthread_mutex_t * mut)`

→ dé-verrouillage (libération)

Libération mémoire :

- `int pthread_mutex_destroy(pthread_mutex_t * mut)`

→ le mutex doit être initialisé et déverrouillé

→ utile pour réinitialiser un mutex (avec `pthread_mutex_init`)

Verrous d'exclusion mutuelle

Verrouillage :

- `int pthread_mutex_lock(pthread_mutex_t * mut)`

→ appel bloquant si mutex déjà tenu

- `int pthread_mutex_trylock(pthread_mutex_t * mut)`

→ non-bloquant, erreur `EBUSY` si mutex déjà tenu

- `int pthread_mutex_unlock(pthread_mutex_t * mut)`

→ dé-verrouillage (libération)

Libération mémoire :

- `int pthread_mutex_destroy(pthread_mutex_t * mut)`

→ le mutex doit être initialisé et déverrouillé

→ utile pour réinitialiser un mutex (avec `pthread_mutex_init`)

Verrous d'exclusion mutuelle

Comportement après terminaison du thread qui le tient ?

→ cf. `man pthread_mutexattr_getrobust`

→ `PTHREAD_MUTEX_STALLED` (défaut) ou `PTHREAD_MUTEX_ROBUST`

Re-verrouillage ? Dé-verrouillage sur mutex non tenu ?

→ cf. `man pthread_mutex_lock`

Mutex type	*Robustness*	*Relock*	*Unlock (not owner)*
NORMAL	non-robust	deadlock	?
	robust	deadlock	error
ERRORCHECK	∼	error	error
RECURSIVE	∼	recursive	error

→ type de mutex configurable via : `pthread_mutexattr_t attr`
`pthread_mutexattr_init`, `pthread_mutexattr_destroy`
`pthread_mutexattr_gettype`, `pthread_mutexattr_settype`
`pthread_mutex_init`

Verrous d'exclusion mutuelle

Comportement après terminaison du thread qui le tient ?

→ cf. `man pthread_mutexattr_getrobust`

→ `PTHREAD_MUTEX_STALLED` (défaut) ou `PTHREAD_MUTEX_ROBUST`

Re-verrouillage ? Dé-verrouillage sur mutex non tenu ?

→ cf. `man pthread_mutex_lock`

Mutex type	*Robustness*	*Relock*	*Unlock (not owner)*
NORMAL	non-robust	deadlock	?
	robust	deadlock	error
ERRORCHECK	∼	error	error
RECURSIVE	∼	recursive	error

→ type de mutex configurable via : `pthread_mutexattr_t attr` `pthread_mutexattr_init`, `pthread_mutexattr_destroy` `pthread_mutexattr_gettype`, `pthread_mutexattr_settype` `pthread_mutex_init`

Verrous d'exclusion mutuelle

Comportement après terminaison du thread qui le tient ?

→ cf. `man pthread_mutexattr_getrobust`

→ `PTHREAD_MUTEX_STALLED` (défaut) ou `PTHREAD_MUTEX_ROBUST`

Re-verrouillage ? Dé-verrouillage sur mutex non tenu ?

→ cf. `man pthread_mutex_lock`

Mutex type	*Robustness*	*Relock*	*Unlock (not owner)*
NORMAL	non-robust	deadlock	?
	robust	deadlock	error
ERRORCHECK	∼	error	error
RECURSIVE	∼	recursive	error

→ type de mutex configurable via : `pthread_mutexattr_t attr` `pthread_mutexattr_init`, `pthread_mutexattr_destroy` `pthread_mutexattr_gettype`, `pthread_mutexattr_settype` `pthread_mutex_init`

Verrous de lecture/écriture

Système POSIX :

- **rwlock** : *read/write lock*

→ variante du mutex

→ indication d'intention d'intervention : lecture ou écriture

→ un seul écrivain possible (sans lecteur), plusieurs lecteurs possibles

Programmation :

- type d'un rwlock : `pthread_rwlock_t`

Initialisation :

- `pthread_rwlock_t rwl = PTHREAD_RWLOCK_INITIALIZER`
-
```
int pthread_rwlock_init(
        pthread_rwlock_t * rwl,
        pthread_rwlockattr_t * attr)
```

→ `attr = NULL` ⇒ valeurs par défaut

Verrous de lecture/écriture

Système POSIX :

- **rwlock** : *read/write lock*

→ variante du mutex

→ indication d'intention d'intervention : lecture ou écriture

→ un seul écrivain possible (sans lecteur), plusieurs lecteurs possibles

Programmation :

- type d'un rwlock : `pthread_rwlock_t`

Initialisation :

- `pthread_rwlock_t rwl = PTHREAD_RWLOCK_INITIALIZER`
-
```
int pthread_rwlock_init(
        pthread_rwlock_t * rwl,
        pthread_rwlockattr_t * attr)
```

→ `attr = NULL` ⇒ valeurs par défaut

Verrous de lecture/écriture

Système POSIX :

- **rwlock** : *read/write lock*

→ variante du mutex

→ indication d'intention d'intervention : lecture ou écriture

→ un seul écrivain possible (sans lecteur), plusieurs lecteurs possibles

Programmation :

- type d'un rwlock : `pthread_rwlock_t`

Initialisation :

- `pthread_rwlock_t rwl = PTHREAD_RWLOCK_INITIALIZER`
- ```
  int pthread_rwlock_init(
              pthread_rwlock_t * rwl,
              pthread_rwlockattr_t * attr)
  ```

→ `attr = NULL` ⇒ valeurs par défaut
  ```

Verrous de lecture/écriture

Verrouillage :

- `int pthread_rwlock_rdlock(pthread_rwlock_t * l)`

→ verrouillage en lecture, appel bloquant si rwlock déjà tenu en écriture

- `int pthread_rwlock_tryrdlock(pthread_rwlock_t * l)`

→ verrouillage en lecture, non-bloquant, erreur `EBUSY` si déjà tenu en écriture

- `int pthread_rwlock_wrlock(pthread_rwlock_t * l)`

→ verrouillage en écriture, appel bloquant si rwlock déjà tenu

- `int pthread_rwlock_trywrlock(pthread_rwlock_t * l)`

→ verrouillage en écriture, non-bloquant, erreur `EBUSY` si déjà tenu

- `int pthread_rwlock_unlock(pthread_rwlock_t * l)`

→ dé-verrouillage (libération)

Remarque : entre demandes d'accès en lecture et en écriture, écriture privilégiée.

Libération mémoire :

- `int pthread_rwlock_destroy(pthread_rwlock_t * l)`

Verrous de lecture/écriture

Verrouillage :

- `int pthread_rwlock_rdlock(pthread_rwlock_t * l)`

→ verrouillage en lecture, appel bloquant si rwlock déjà tenu en écriture

- `int pthread_rwlock_tryrdlock(pthread_rwlock_t * l)`

→ verrouillage en lecture, non-bloquant, erreur `EBUSY` si déjà tenu en écriture

- `int pthread_rwlock_wrlock(pthread_rwlock_t * l)`

→ verrouillage en écriture, appel bloquant si rwlock déjà tenu

- `int pthread_rwlock_trywrlock(pthread_rwlock_t * l)`

→ verrouillage en écriture, non-bloquant, erreur `EBUSY` si déjà tenu

- `int pthread_rwlock_unlock(pthread_rwlock_t * l)`

→ dé-verrouillage (libération)

Remarque : entre demandes d'accès en lecture et en écriture, écriture privilégiée.

Libération mémoire :

- `int pthread_rwlock_destroy(pthread_rwlock_t * l)`

Verrous de lecture/écriture

Verrouillage :

- `int pthread_rwlock_rdlock(pthread_rwlock_t * l)`

→ verrouillage en lecture, appel bloquant si rwlock déjà tenu en écriture

- `int pthread_rwlock_tryrdlock(pthread_rwlock_t * l)`

→ verrouillage en lecture, non-bloquant, erreur `EBUSY` si déjà tenu en écriture

- `int pthread_rwlock_wrlock(pthread_rwlock_t * l)`

→ verrouillage en écriture, appel bloquant si rwlock déjà tenu

- `int pthread_rwlock_trywrlock(pthread_rwlock_t * l)`

→ verrouillage en écriture, non-bloquant, erreur `EBUSY` si déjà tenu

- `int pthread_rwlock_unlock(pthread_rwlock_t * l)`

→ dé-verrouillage (libération)

Remarque : entre demandes d'accès en lecture et en écriture, écriture privilégiée.

Libération mémoire :

- `int pthread_rwlock_destroy(pthread_rwlock_t * l)`

Variables condition

Système POSIX :

- **cond** : *condition*

→ mécanisme d'attente d'une condition par un thread

→ signalisation de cette condition par un autre thread
(débloque le(s) thread(s) en attente)

Programmation :

- type d'une condition : `pthread_cond_t`

Initialisation :

- `pthread_cond_t cond = PTHREAD_COND_INITIALIZER`
-
```
int pthread_cond_init(
        pthread_cond_t * cond,
        pthread_condattr_t * attr)
```

→ NPTL ne gère pas d'attribut utile pour les conditions (`attr = NULL`)

Variables condition

Système POSIX :

- **cond** : *condition*

→ mécanisme d'attente d'une condition par un thread

→ signalisation de cette condition par un autre thread
(débloque le(s) thread(s) en attente)

Programmation :

- type d'une condition : `pthread_cond_t`

Initialisation :

- `pthread_cond_t cond = PTHREAD_COND_INITIALIZER`
-
```
int pthread_cond_init(
        pthread_cond_t * cond,
        pthread_condattr_t * attr)
```

→ NPTL ne gère pas d'attribut utile pour les conditions (`attr = NULL`)

Variables condition

Système POSIX :

- **cond** : *condition*

→ mécanisme d'attente d'une condition par un thread

→ signalisation de cette condition par un autre thread
(débloque le(s) thread(s) en attente)

Programmation :

- type d'une condition : `pthread_cond_t`

Initialisation :

- `pthread_cond_t cond = PTHREAD_COND_INITIALIZER`
- ```
  int pthread_cond_init(
              pthread_cond_t * cond,
              pthread_condattr_t * attr)
  ```

→ NPTL ne gère pas d'attribut utile pour les conditions (`attr = NULL`)
  ```

Variables condition

Attente d'une condition :

- `int pthread_cond_wait(`
  `pthread_cond_t * cond,`
  `pthread_mutex_t * mutex)`

→ appel bloquant, jusqu'au signalement de la condition

→ condition associée à un mutex (pour éviter concurrence d'accès à la condition)

Signalement d'une condition :

- `int pthread_cond_signal(pthread_cond_t * cond)`
- `int pthread_cond_broadcast(pthread_cond_t * cond)`

→ si une attente existe ⇒ consommation de la condition
sinon ⇒ signalement de la condition sans effet.

Libération mémoire :

- `int pthread_cond_destroy(pthread_cond_t * cond)`

Variables condition

Attente d'une condition :

- `int pthread_cond_wait(`
  `pthread_cond_t * cond,`
  `pthread_mutex_t * mutex)`

→ appel bloquant, jusqu'au signalement de la condition

→ condition associée à un mutex (pour éviter concurrence d'accès à la condition)

Signalement d'une condition :

- `int pthread_cond_signal(pthread_cond_t * cond)`
- `int pthread_cond_broadcast(pthread_cond_t * cond)`

→ si une attente existe ⇒ consommation de la condition
sinon ⇒ signalement de la condition sans effet.

Libération mémoire :

- `int pthread_cond_destroy(pthread_cond_t * cond)`

Variables condition

Attente d'une condition :

- `int pthread_cond_wait(`
  `pthread_cond_t * cond,`
  `pthread_mutex_t * mutex)`

→ appel bloquant, jusqu'au signalement de la condition

→ condition associée à un mutex (pour éviter concurrence d'accès à la condition)

Signalement d'une condition :

- `int pthread_cond_signal(pthread_cond_t * cond)`
- `int pthread_cond_broadcast(pthread_cond_t * cond)`

→ si une attente existe ⇒ consommation de la condition
sinon ⇒ signalement de la condition sans effet.

Libération mémoire :

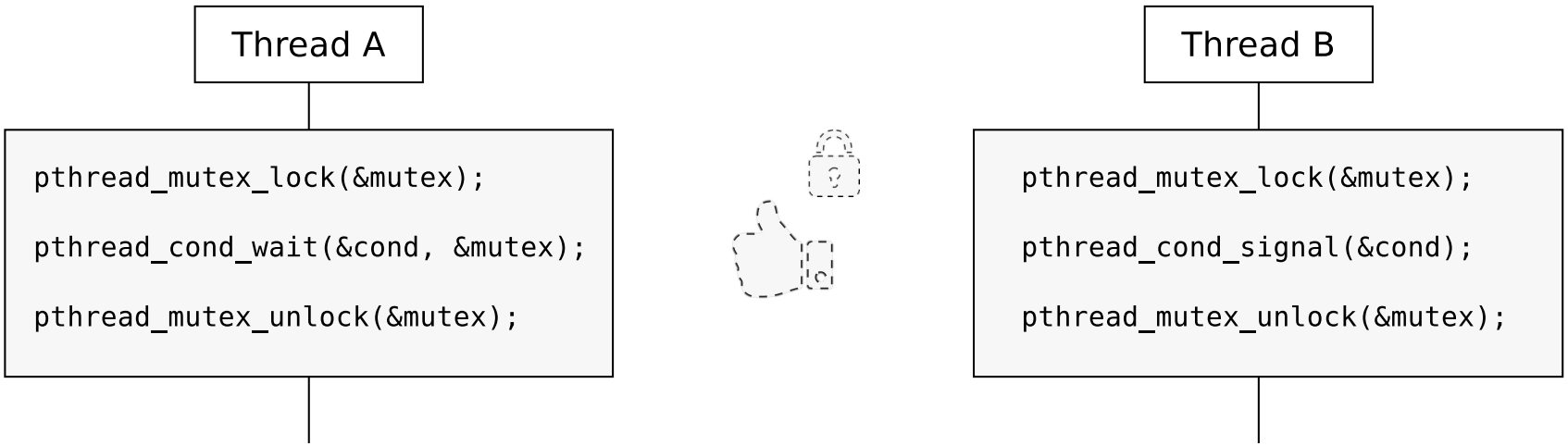
- `int pthread_cond_destroy(pthread_cond_t * cond)`

Variables condition

Illustration :

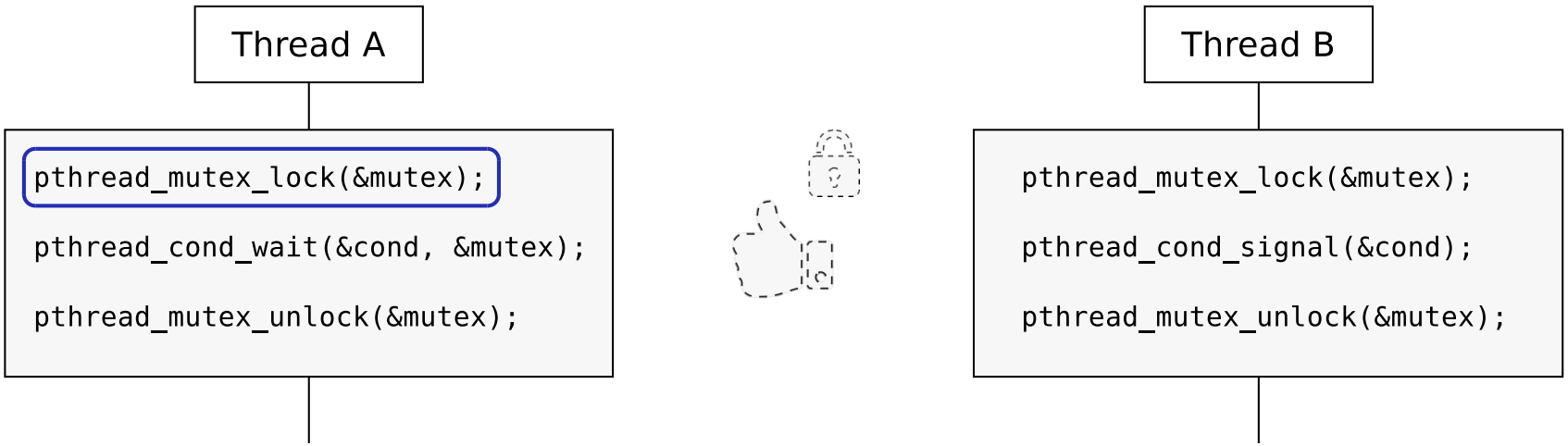
Variables condition

Illustration :



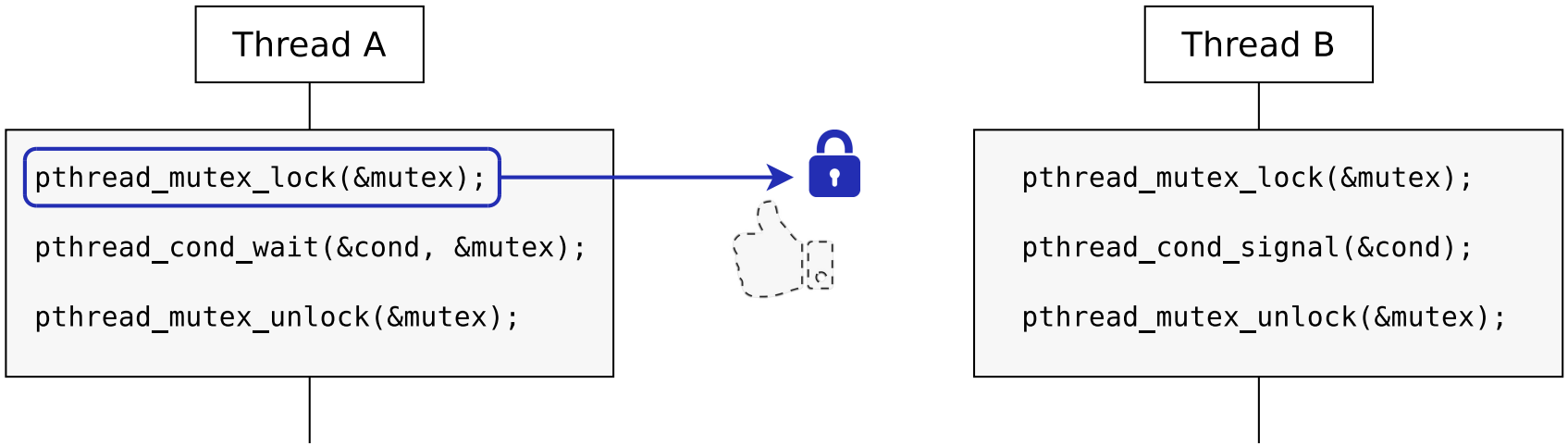
Variables condition

Illustration :



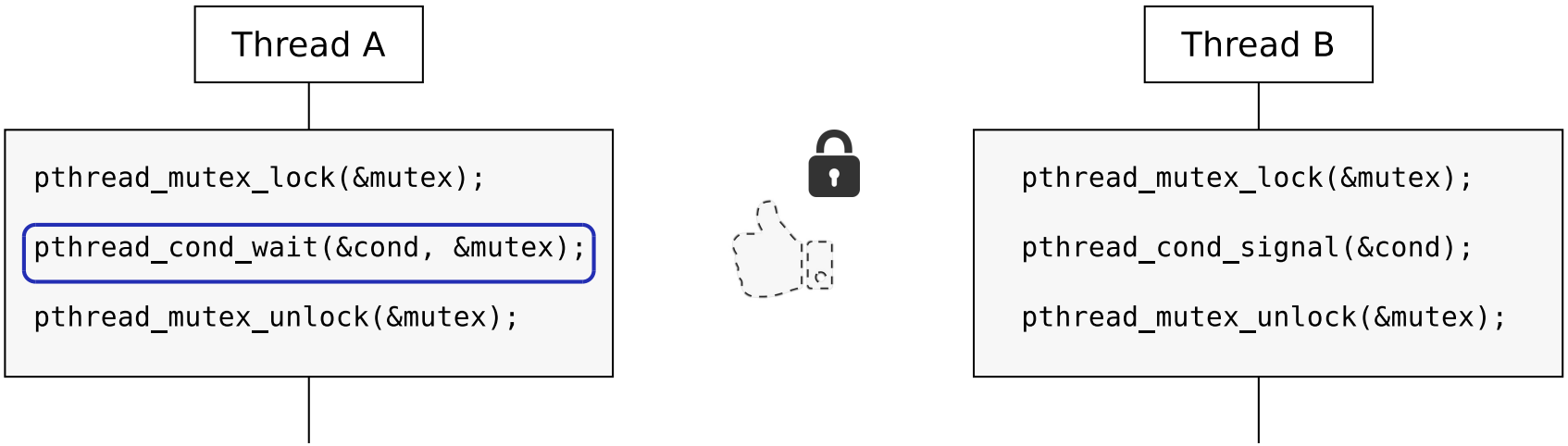
Variables condition

Illustration :



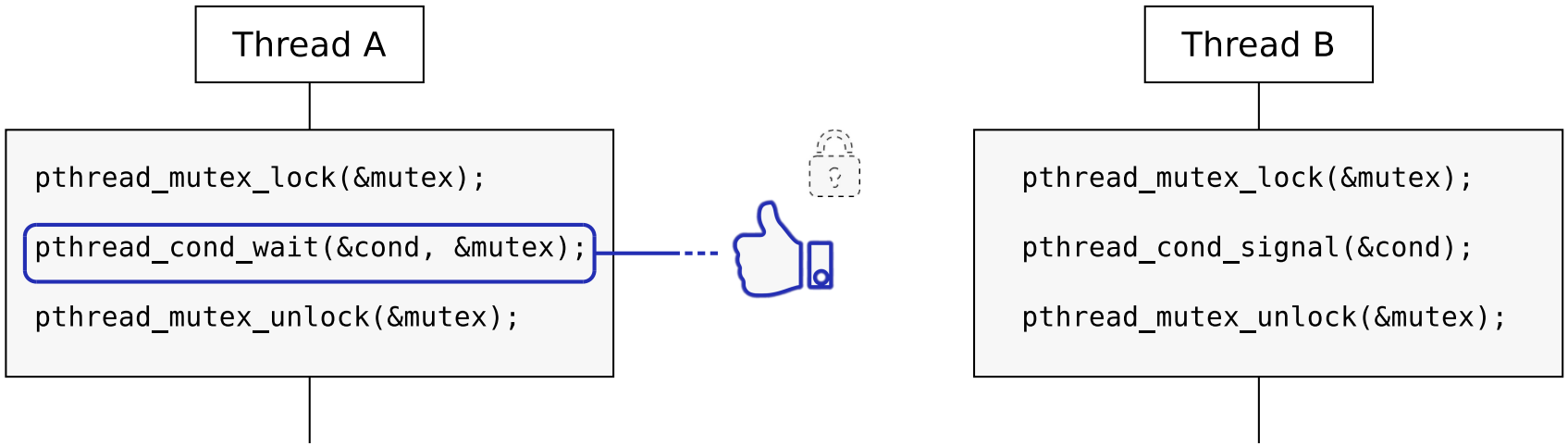
Variables condition

Illustration :



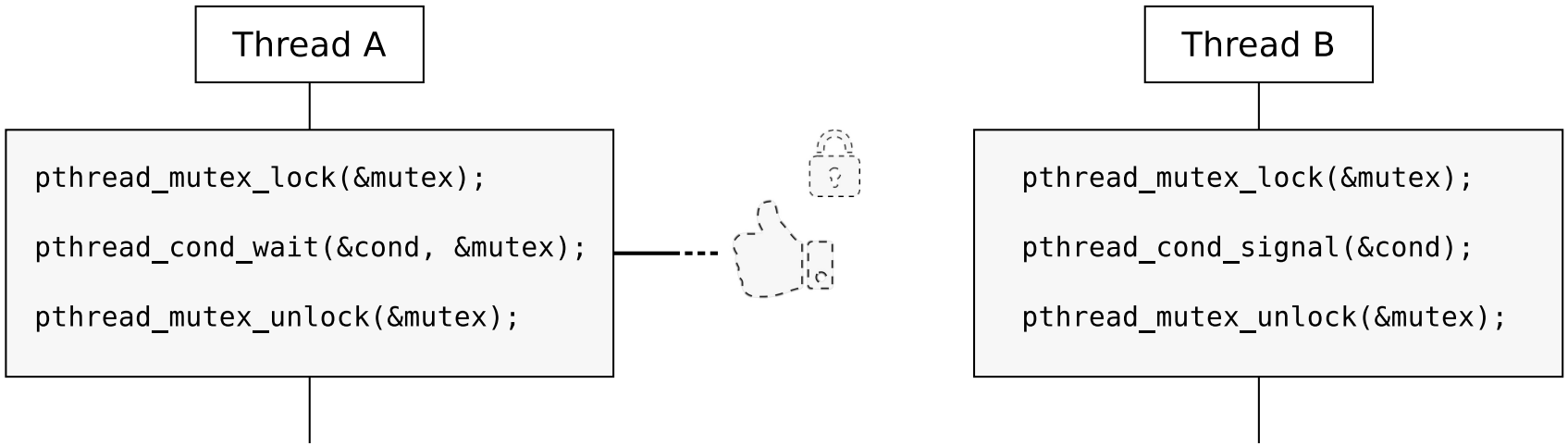
Variables condition

Illustration :



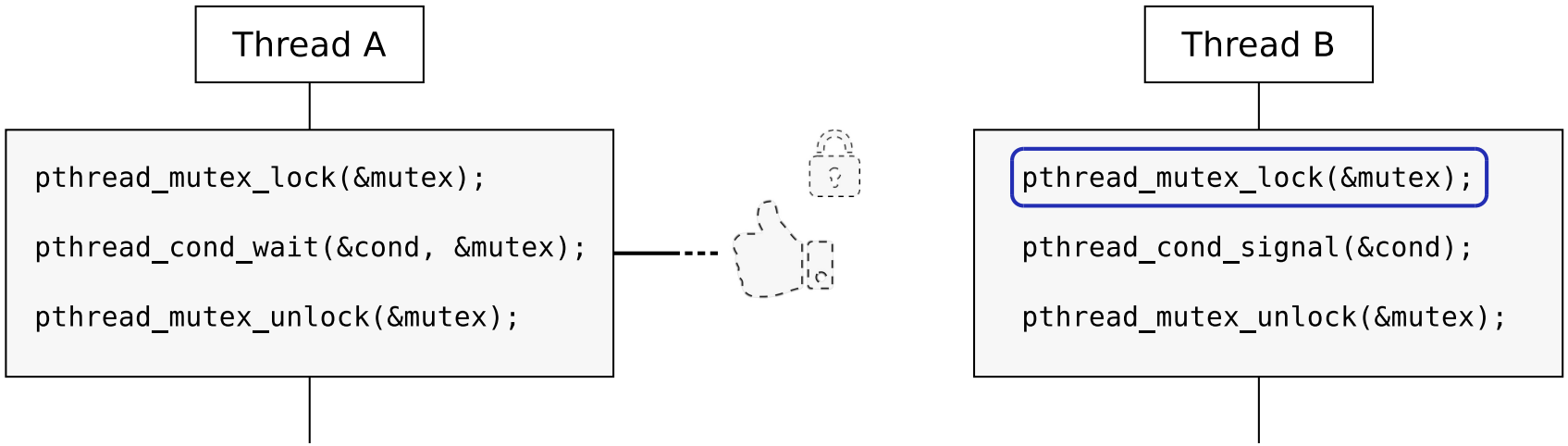
Variables condition

Illustration :



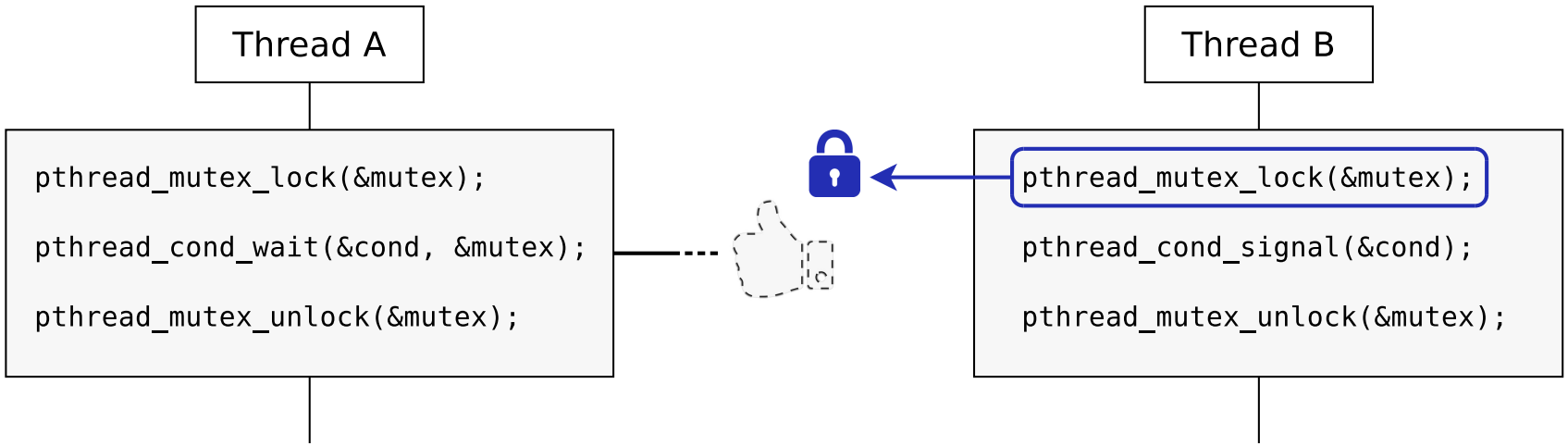
Variables condition

Illustration :



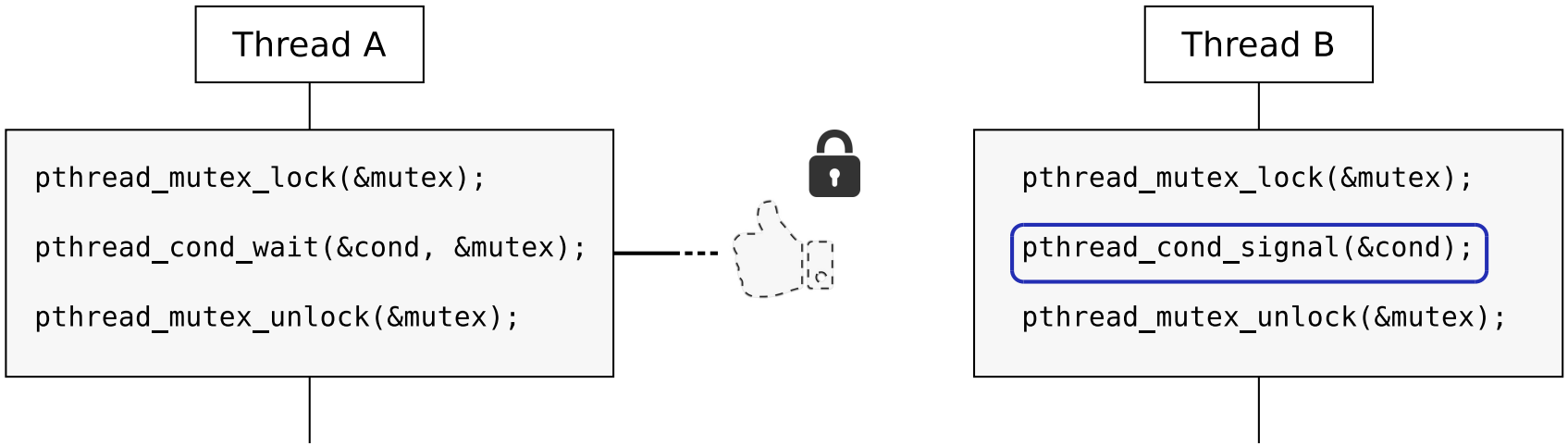
Variables condition

Illustration :



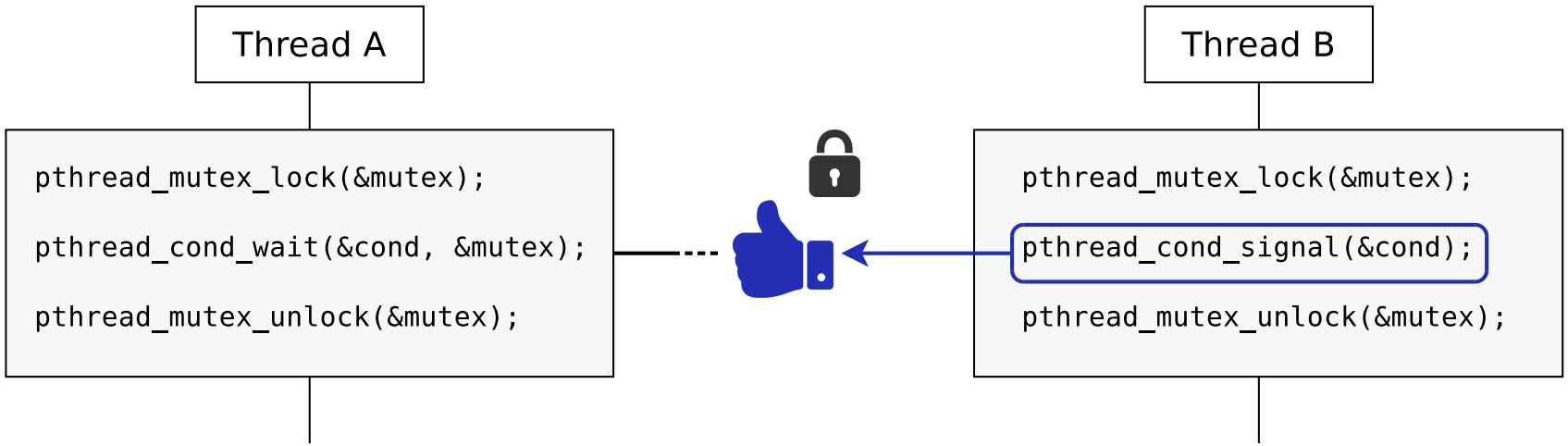
Variables condition

Illustration :



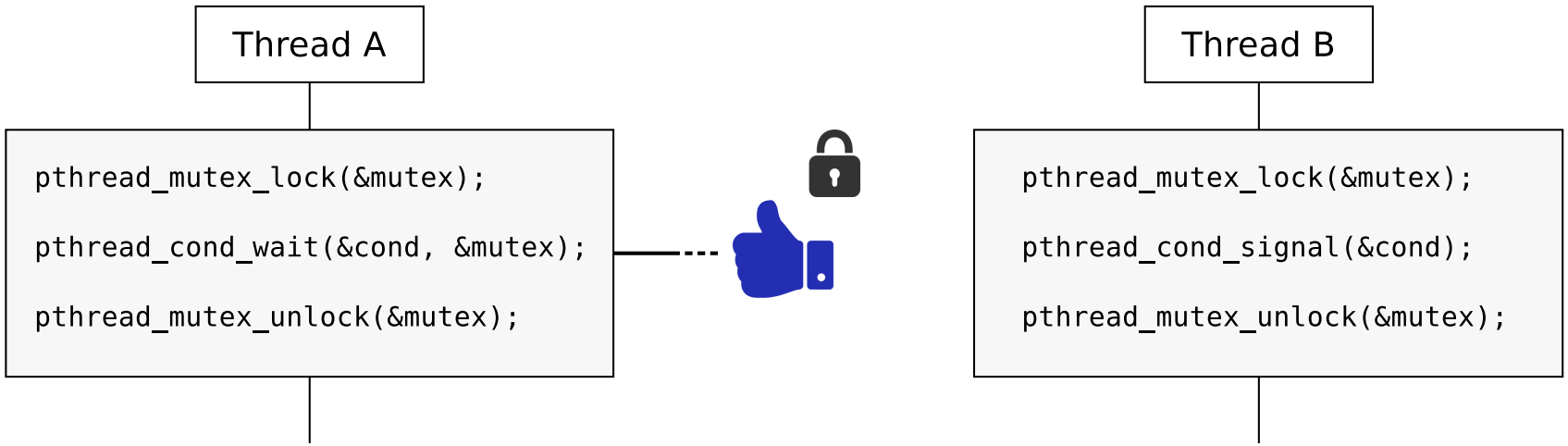
Variables condition

Illustration :



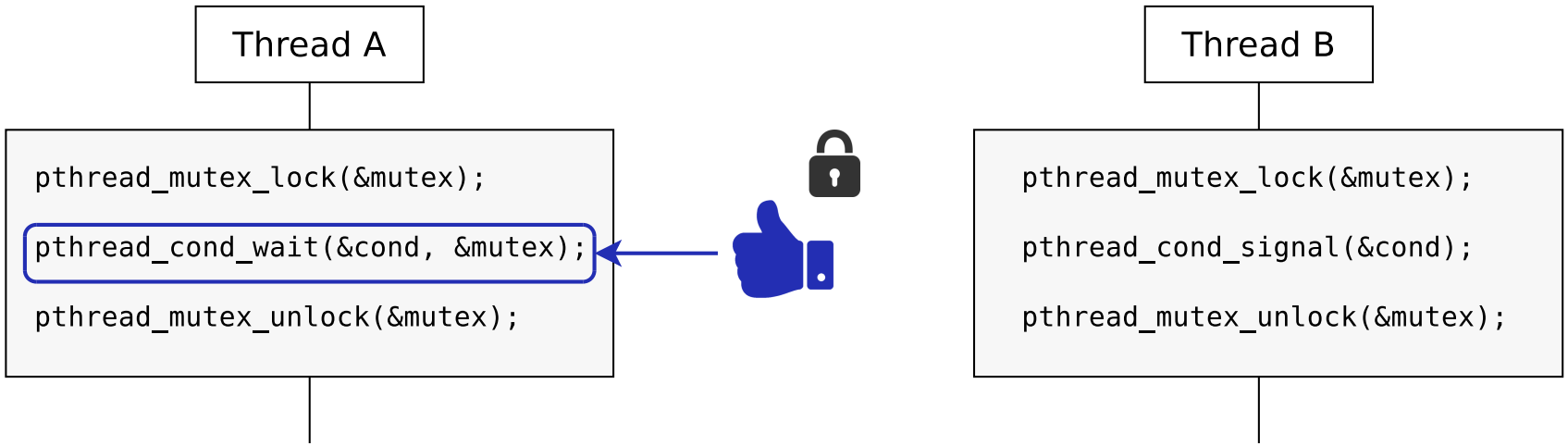
Variables condition

Illustration :



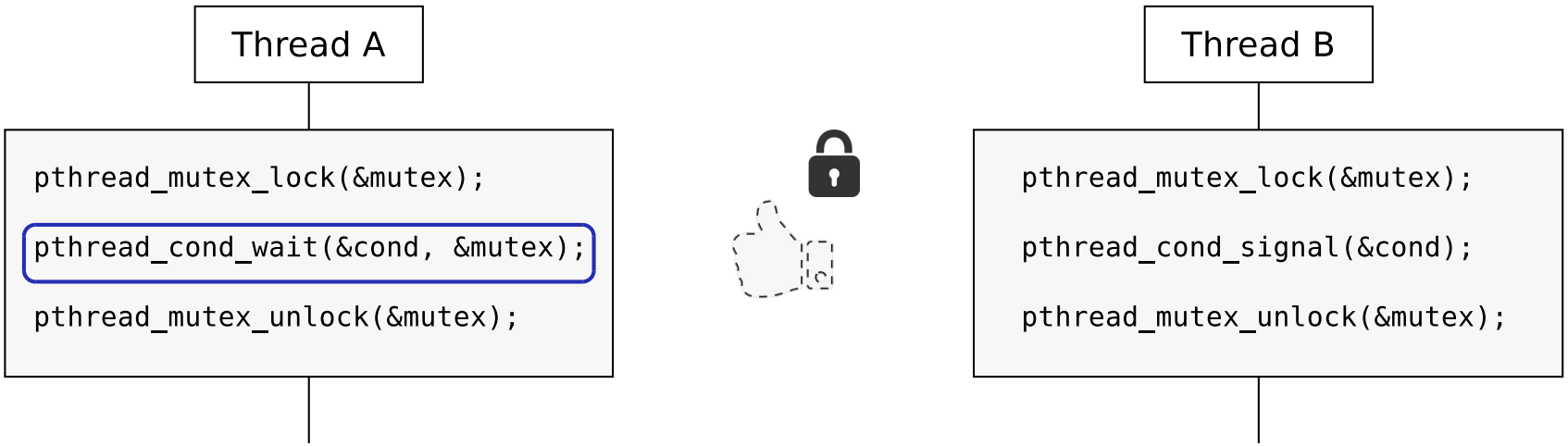
Variables condition

Illustration :



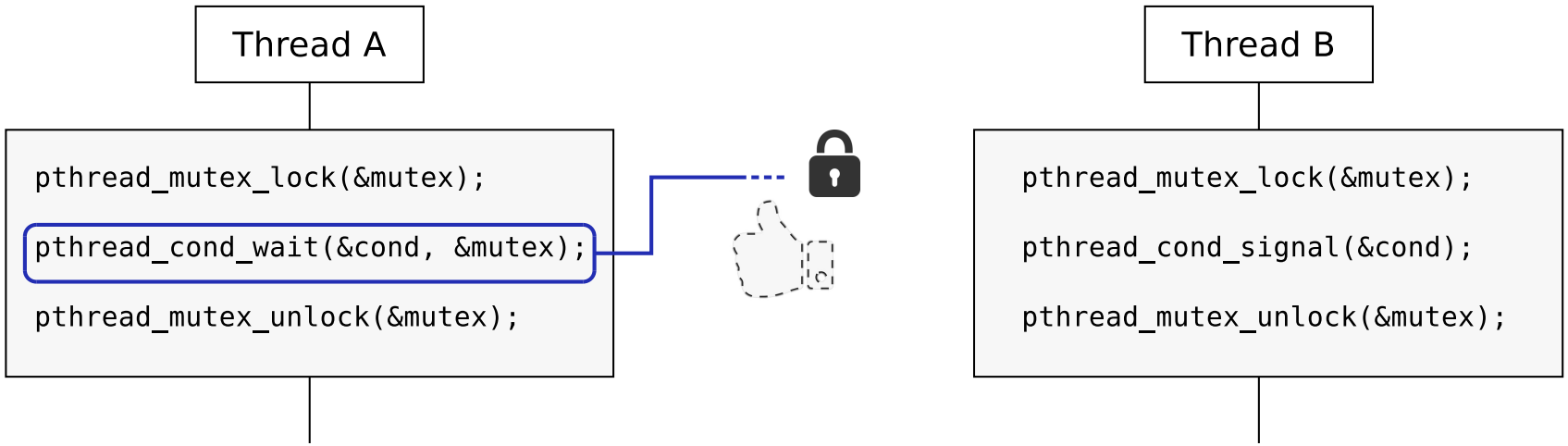
Variables condition

Illustration :



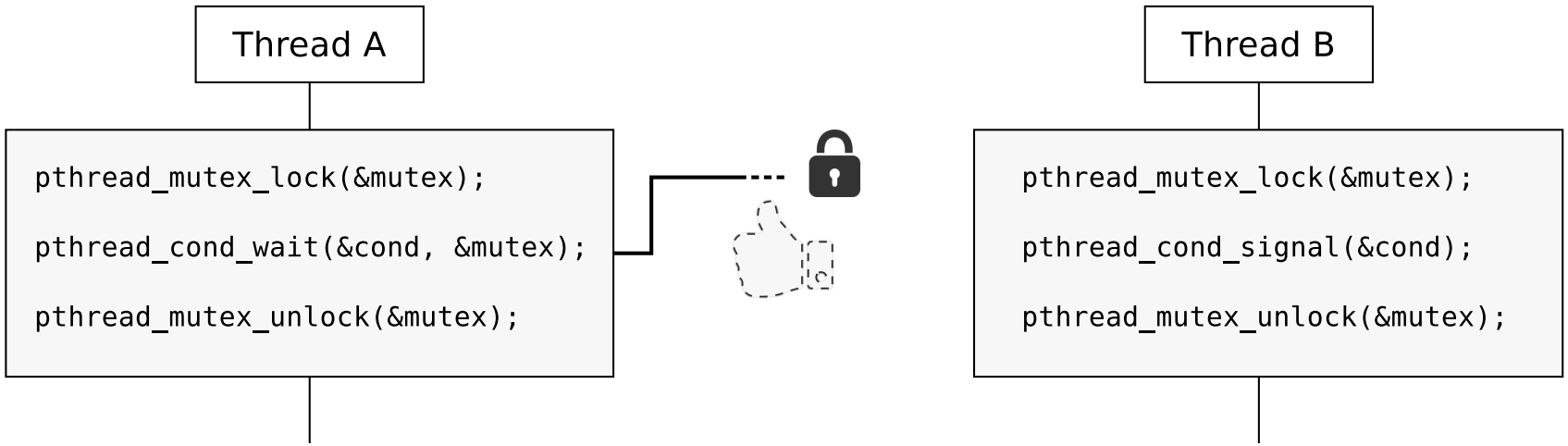
Variables condition

Illustration :



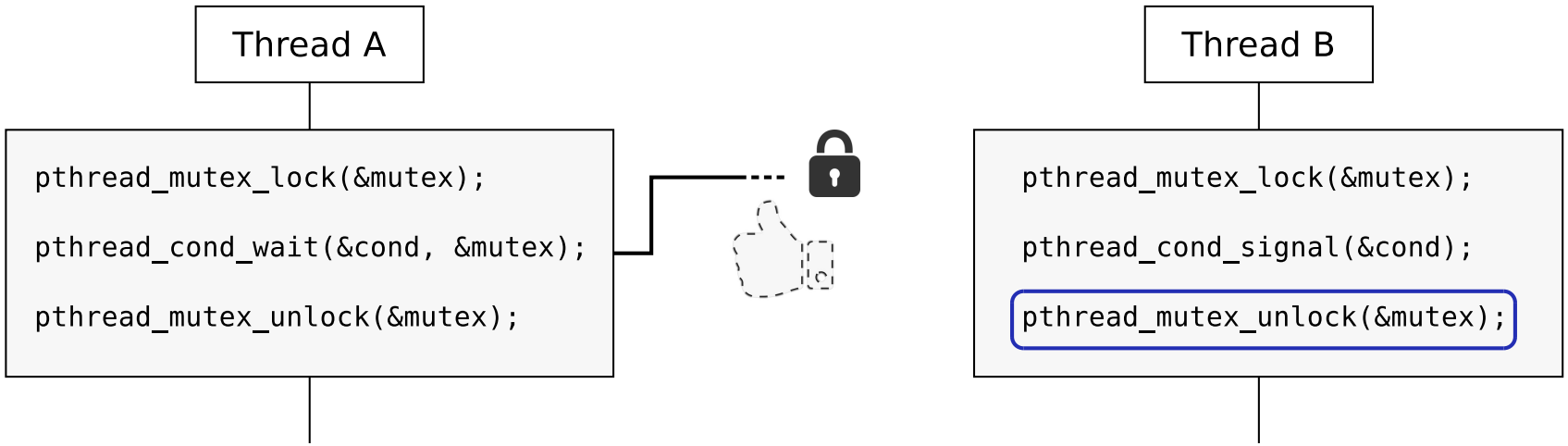
Variables condition

Illustration :



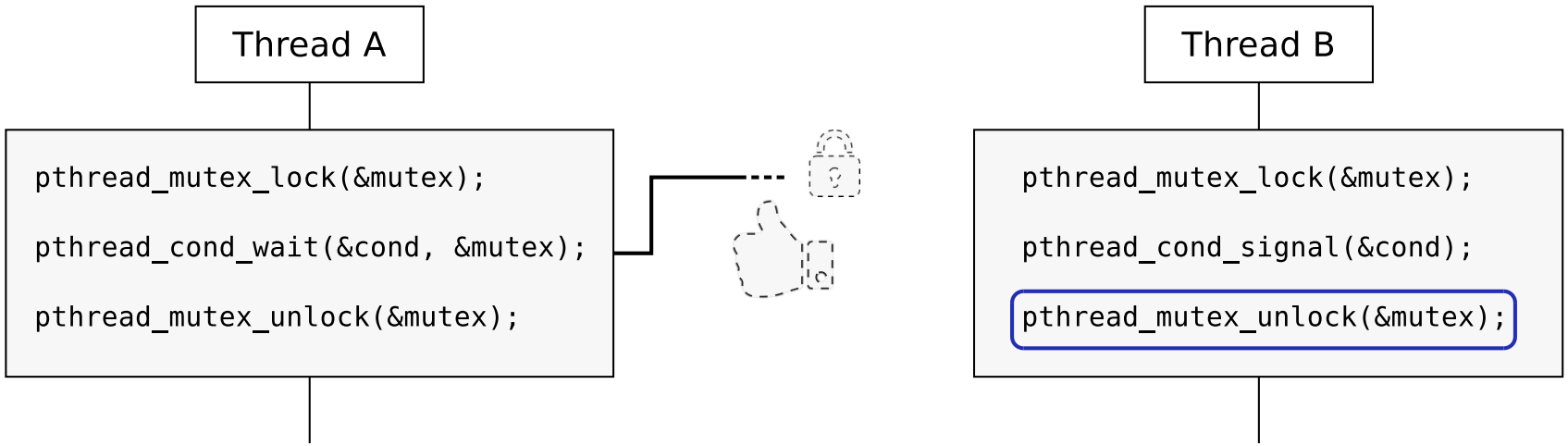
Variables condition

Illustration :



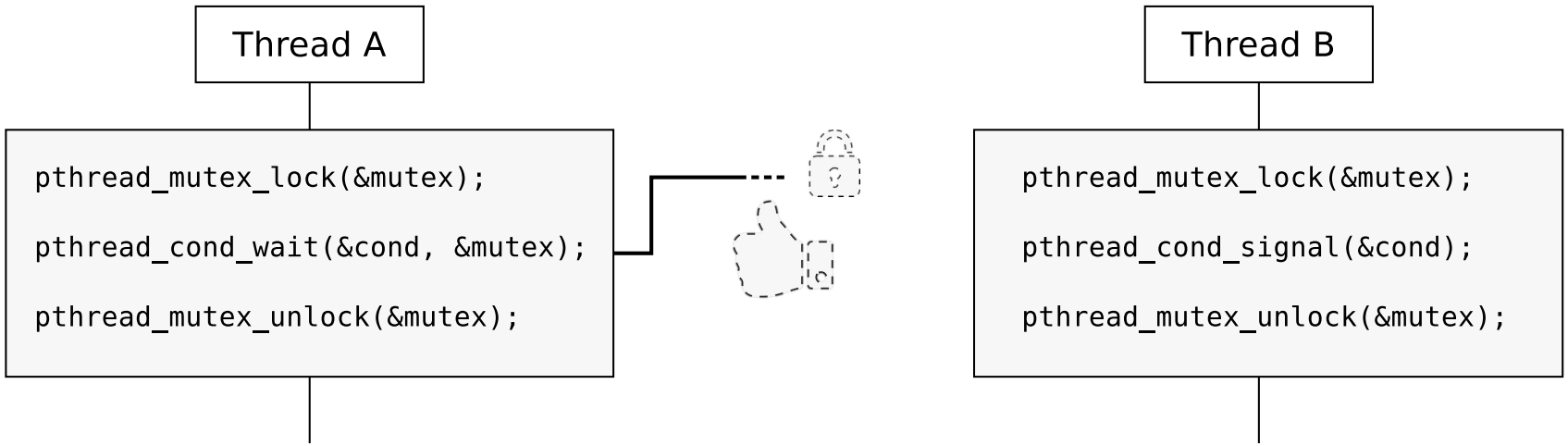
Variables condition

Illustration :



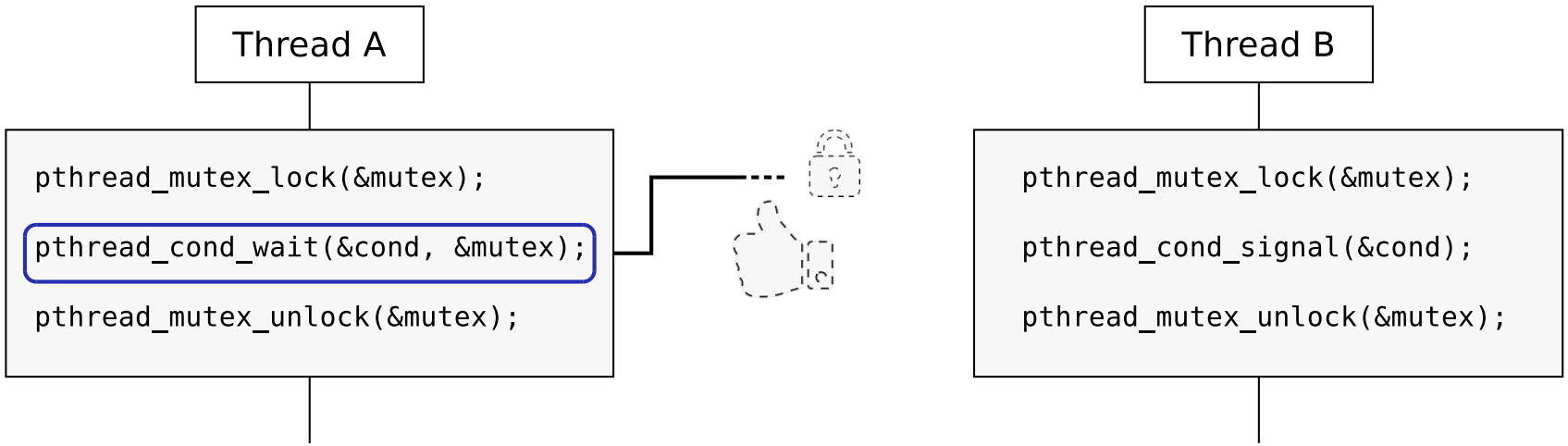
Variables condition

Illustration :



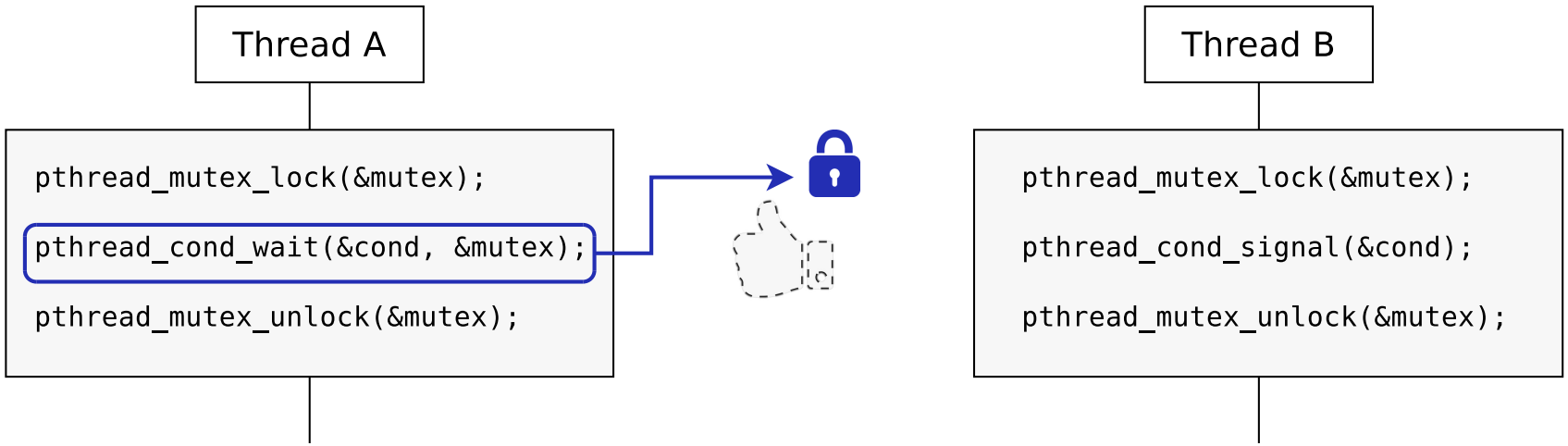
Variables condition

Illustration :



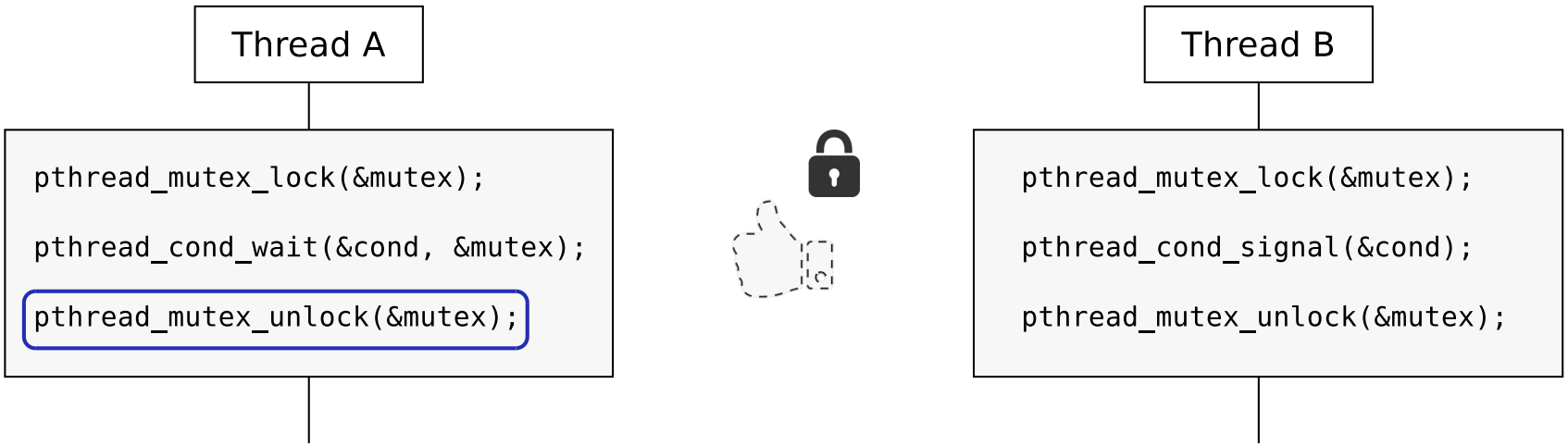
Variables condition

Illustration :



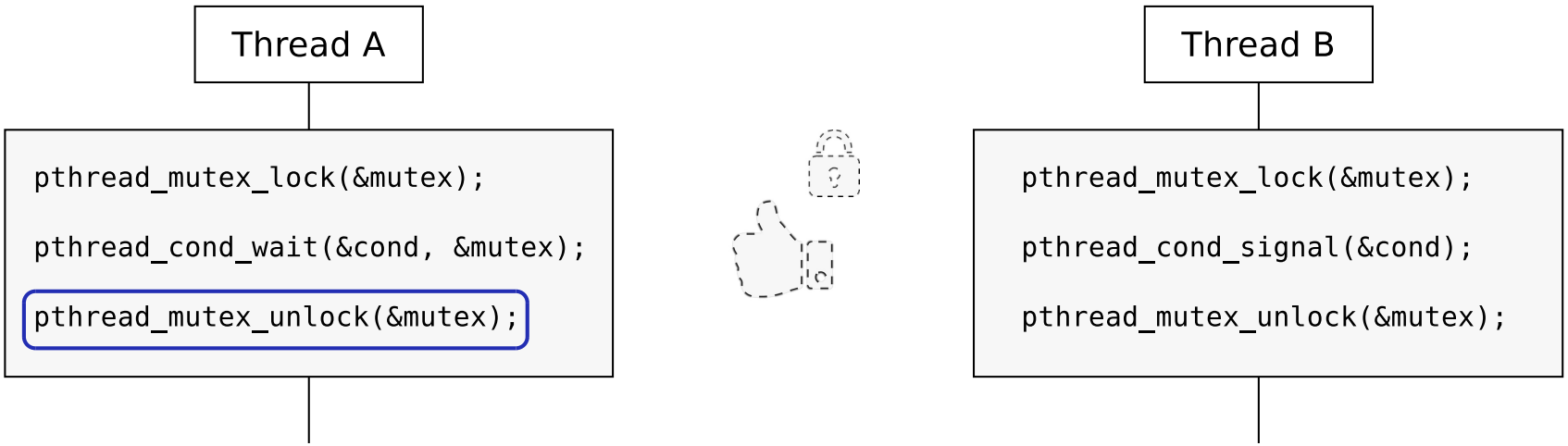
Variables condition

Illustration :



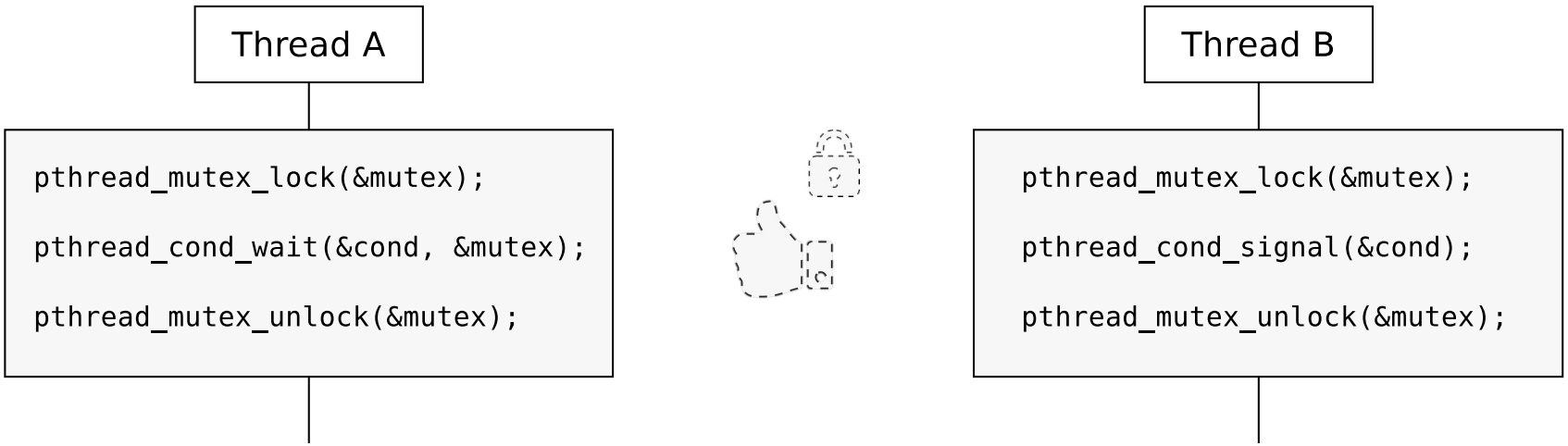
Variables condition

Illustration :



Variables condition

Illustration :



Barrières de synchronisation

Système POSIX :

- **barrier** : *barrier object*

→ mécanisme de synchronisation de plusieurs threads

→ nombre fixe de threads à synchroniser associé à la barrière

Programmation :

- type d'une barrière : `pthread_barrier_t`

Initialisation :

- ```
  int pthread_barrier_init(
              pthread_barrier_t * barrier,
              pthread_barrierattr_t * attr,
              unsigned int count)
  ```

→ `count` : nombre de threads à synchroniser

→ NPTL ne gère pas d'attribut utile pour les barrières (`attr = NULL`)
  ```

Barrières de synchronisation

Système POSIX :

- **barrier** : *barrier object*

→ mécanisme de synchronisation de plusieurs threads

→ nombre fixe de threads à synchroniser associé à la barrière

Programmation :

- type d'une barrière : `pthread_barrier_t`

Initialisation :

-
```
int pthread_barrier_init(
        pthread_barrier_t * barrier,
        pthread_barrierattr_t * attr,
        unsigned int count)
```

→ `count` : nombre de threads à synchroniser

→ NPTL ne gère pas d'attribut utile pour les barrières (`attr = NULL`)

Barrières de synchronisation

Système POSIX :

- **barrier** : *barrier object*

→ mécanisme de synchronisation de plusieurs threads

→ nombre fixe de threads à synchroniser associé à la barrière

Programmation :

- type d'une barrière : `pthread_barrier_t`

Initialisation :

- ```
  int pthread_barrier_init(
              pthread_barrier_t * barrier,
              pthread_barrierattr_t * attr,
              unsigned int count)
  ```

→ `count` : nombre de threads à synchroniser

→ NPTL ne gère pas d'attribut utile pour les barrières (`attr = NULL`)
  ```

Barrières de synchronisation

Attente à une barrière :

- `int pthread_barrier_wait(`
  `pthread_barrier_t * barrier)`

→ appel bloquant, jusqu'au nombre attendu de threads à synchroniser

Libération mémoire :

- `int pthread_barrier_destroy(`
  `pthread_barrier_t * barrier)`

→ la barrière doit être initialisée et sans attente sur elle

→ utile pour réinitialiser une barrière (avec `pthread_barrier_init`)

Barrières de synchronisation

Attente à une barrière :

- `int pthread_barrier_wait(`
  `pthread_barrier_t * barrier)`

→ appel bloquant, jusqu'au nombre attendu de threads à synchroniser

Libération mémoire :

- `int pthread_barrier_destroy(`
  `pthread_barrier_t * barrier)`

→ la barrière doit être initialisée et sans attente sur elle

→ utile pour réinitialiser une barrière (avec `pthread_barrier_init`)

Sémaphores

Principe (Edsger DIJKSTRA, 1965)

- sémaphore avec compteur, valeur = `n`
- demander un accès :

```
P() {
   début: si n > 0 { n-- }
          sinon { attente libération; goto début }
}
```

- libérer un accès :

```
V() {
   n++;
   si n > 0 { libère un thread en attente }
}
```

- `P` et `V` atomiques

Sémaphores

Principe (Edsger DIJKSTRA, 1965)

- sémaphore avec compteur, valeur = `n`
- demander un accès :

```
P() {
   début: si n > 0 { n-- }
          sinon { attente libération; goto début }
}
```

- libérer un accès :

```
V() {
   n++;
   si n > 0 { libère un thread en attente }
}
```

- `P` et `V` atomiques

Sémaphores

Principe (Edsger DIJKSTRA, 1965)

- sémaphore avec compteur, valeur = `n`
- demander un accès :

```
P() {
   début: si n > 0 { n-- }
          sinon { attente libération; goto début }
}
```

- libérer un accès :

```
V() {
   n++;
   si n > 0 { libère un thread en attente }
}
```

- `P` et `V` atomiques

Sémaphores

Principe (Edsger DIJKSTRA, 1965)

- sémaphore avec compteur, valeur = `n`
- demander un accès :

```
P() {
   début: si n > 0 { n-- }
          sinon { attente libération; goto début }
}
```

- libérer un accès :

```
V() {
   n++;
   si n > 0 { libère un thread en attente }
}
```

- `P` et `V` atomiques

Sémaphores

Principe (Edsger DIJKSTRA, 1965)

- sémaphore avec compteur, valeur = `n`
- demander un accès :

```
P() {
   début: si n > 0 { n-- }
          sinon { attente libération; goto début }
}
```

- libérer un accès :

```
V() {
   n++;
   si n > 0 { libère un thread en attente }
}
```

- `P` et `V` atomiques

Sémaphores

Système POSIX :

- **sem** : *semaphore*

→ IPC (*Inter Process Communication*)

→ `ENOSYS` ⇒ recompilation noyau nécessaire

→ accès ressoures IPC : `/dev/shm` (système de fichier virtuel `tmpfs`)

→ **sémaphore anonyme** : utilisable en mémoire partagée

→ **sémaphore nommé** : utilisable dans différents espaces mémoires

→ `man sem_overview`

Programmation :

- `<semaphore.h>`
- options gcc : `-pthread -lrt`
- type d'un sémaphore : `sem_t`

Sémaphores

Système POSIX :

- **sem** : *semaphore*

→ IPC (*Inter Process Communication*)

→ `ENOSYS` ⇒ recompilation noyau nécessaire

→ accès ressoures IPC : `/dev/shm` (système de fichier virtuel `tmpfs`)

→ **sémaphore anonyme** : utilisable en mémoire partagée

→ **sémaphore nommé** : utilisable dans différents espaces mémoires

→ `man sem_overview`

Programmation :

- `<semaphore.h>`
- options gcc : `-pthread -lrt`
- type d'un sémaphore : `sem_t`

Sémaphores

Système POSIX :

- **sem** : *semaphore*

→ IPC (*Inter Process Communication*)

→ `ENOSYS` ⇒ recompilation noyau nécessaire

→ accès ressoures IPC : `/dev/shm` (système de fichier virtuel `tmpfs`)

→ **sémaphore anonyme** : utilisable en mémoire partagée

→ **sémaphore nommé** : utilisable dans différents espaces mémoires

→ `man sem_overview`

Programmation :

- `<semaphore.h>`
- options gcc : `-pthread -lrt`
- type d'un sémaphore : `sem_t`

Sémaphores

Système POSIX :

- **sem** : *semaphore*

→ IPC (*Inter Process Communication*)

→ `ENOSYS` ⇒ recompilation noyau nécessaire

→ accès ressoures IPC : `/dev/shm` (système de fichier virtuel `tmpfs`)

→ **sémaphore anonyme** : utilisable en mémoire partagée

→ **sémaphore nommé** : utilisable dans différents espaces mémoires

→ `man sem_overview`

Programmation :

- `<semaphore.h>`
- options gcc : `-pthread -lrt`
- type d'un sémaphore : `sem_t`

Sémaphores anonymes

Initialisation :

- `int sem_init(`
  `sem_t * sem,`
  `int shared,`
  `unsigned int value)`

→ `shared` : partage entre différents processus

→ `value` : valeur initiale du compteur

Libération mémoire :

- `int sem_destroy(sem_t * sem)`

→ le sémaphore doit être initialisé et sans attente sur lui

→ utile pour réinitialiser un sémaphore (avec `sem_init`)

Sémaphores anonymes

Initialisation :

- `int sem_init(`
  `sem_t * sem,`
  `int shared,`
  `unsigned int value)`

→ `shared` : partage entre différents processus

→ `value` : valeur initiale du compteur

Libération mémoire :

- `int sem_destroy(sem_t * sem)`

→ le sémaphore doit être initialisé et sans attente sur lui

→ utile pour réinitialiser un sémaphore (avec `sem_init`)

Sémaphores nommés

Initialisation :

- `sem_t * sem_open(`
  `const char * name,`
  `int flags,`
  `mode_t mode,`
  `unsigned int value)`

→ `name` : identifiant du sémaphore (commence par '/')

→ `flags` : pour création (O_CREATE) (cf. `<fcntl.h>`)

→ `mode` : permissions (à la création) (cf. `man 2 open`, `<sys/stat.h>`)

→ `value` : valeur initiale du compteur

→ retourne `SEM_FAILED` si erreur (`errno` renseigné)

- `sem_t * sem_open(const char * name, int flags)`

→ accès à un sémaphore déjà créé

→ retourne `SEM_FAILED` si erreur (`errno` renseigné)

Sémaphores nommés

Initialisation :

- `sem_t * sem_open(`
  `const char * name,`
  `int flags,`
  `mode_t mode,`
  `unsigned int value)`

→ `name` : identifiant du sémaphore (commence par '/')

→ `flags` : pour création (O_CREATE) (cf. `<fcntl.h>`)

→ `mode` : permissions (à la création) (cf. `man 2 open`, `<sys/stat.h>`)

→ `value` : valeur initiale du compteur

→ retourne `SEM_FAILED` si erreur (`errno` renseigné)

- `sem_t * sem_open(const char * name, int flags)`

→ accès à un sémaphore déjà créé

→ retourne `SEM_FAILED` si erreur (`errno` renseigné)

Sémaphores nommés

Libération mémoire :

- `int sem_close(sem_t * sem)`

→ fermeture (le sémaphore reste à disposition d'autres processus)

- `int sem_unlink(const char * name)`

→ suppression de l'identifiant du sémaphore

→ ouverture impossible mais reste disponible pour processus en cours d'utilisation

→ destruction effective après le dernier `sem_close` (des process en court d'utilisation)

Sémaphores nommés

Libération mémoire :

- `int sem_close(sem_t * sem)`

→ fermeture (le sémaphore reste à disposition d'autres processus)

- `int sem_unlink(const char * name)`

→ suppression de l'identifiant du sémaphore

→ ouverture impossible mais reste disponible pour processus en cours d'utilisation

→ destruction effective après le dernier `sem_close` (des process en court d'utilisation)

Sémaphores

Accès :

- `int sem_wait(sem_t * sem)`

→ ∼ `P()` : demander un accès

→ appel bloquant si compteur du sémaphore à zéro

- `int sem_trywait(sem_t * sem)`

→ non-bloquant, erreur `EAGAIN` si compteur du sémaphore à zéro

- `int sem_post(sem_t * sem)`

→ ∼ `V()` : libérer un accès

Besoins en communication

Besoins en communication

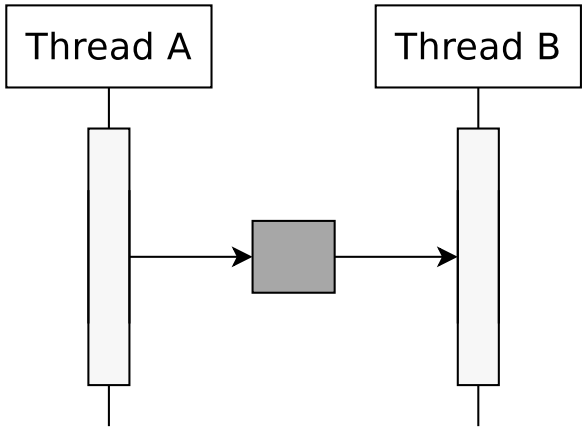


Tableau Noir

Besoins en communication

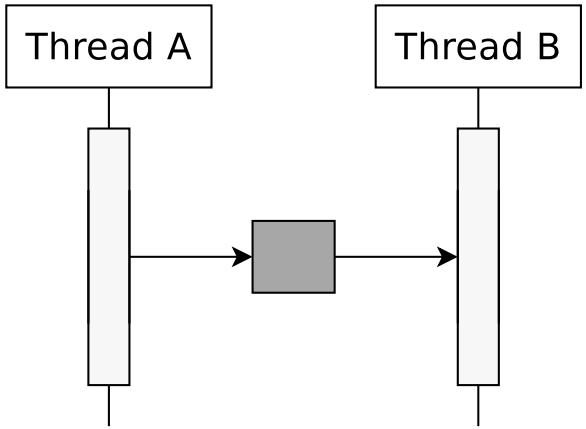
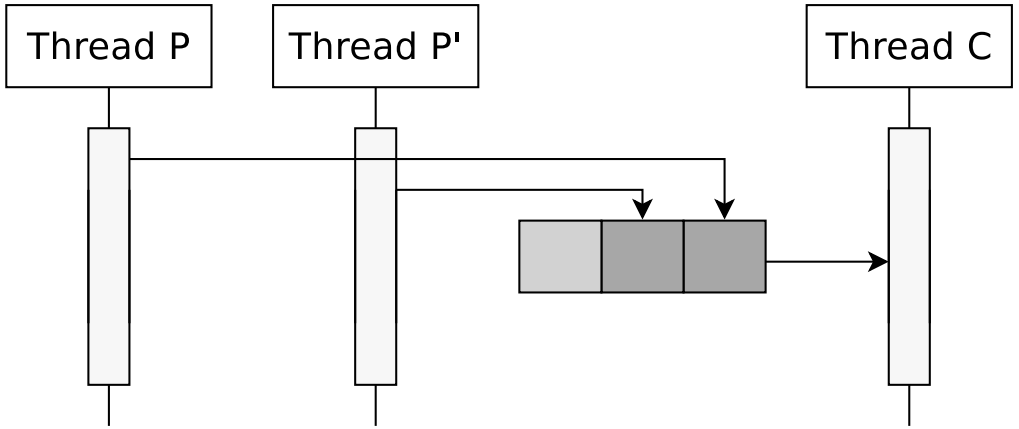


Tableau Noir



Producteurs/Consommateur

Besoins en communication

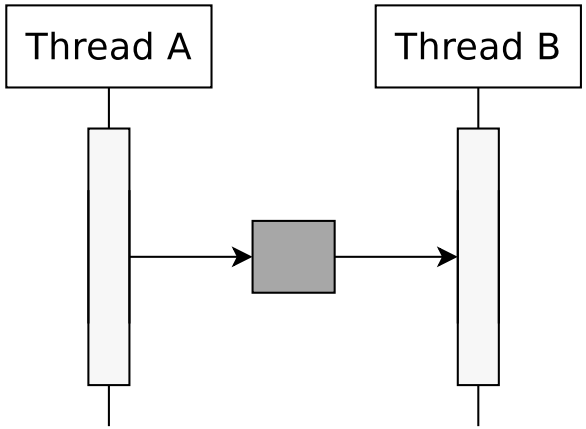
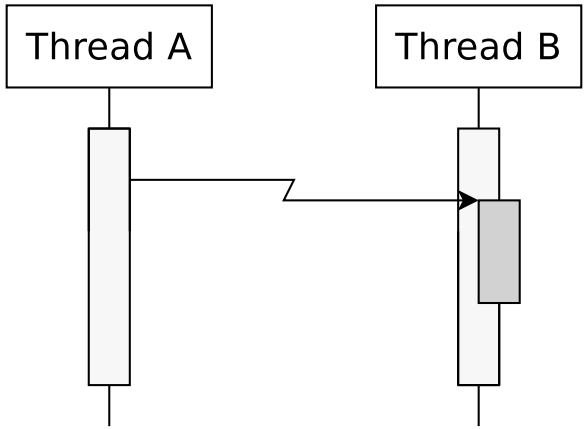
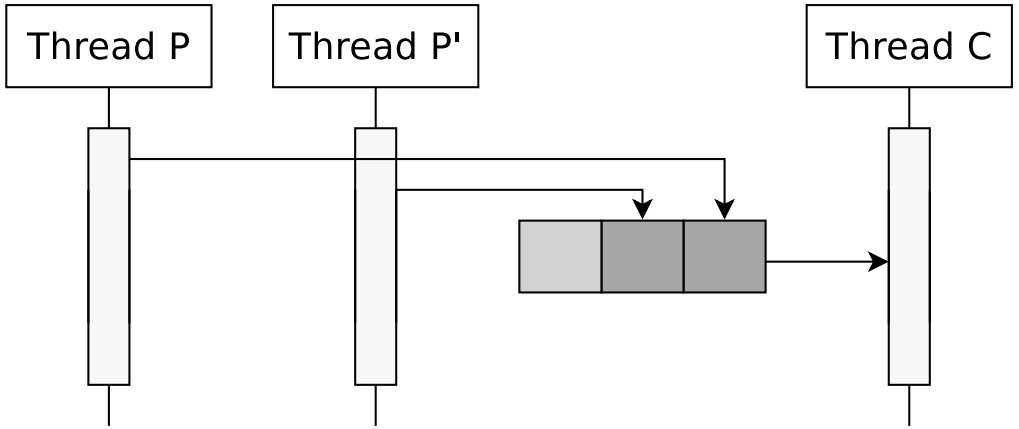


Tableau Noir



Signal Évènement



Producteurs/Consommateur

Besoins en communication

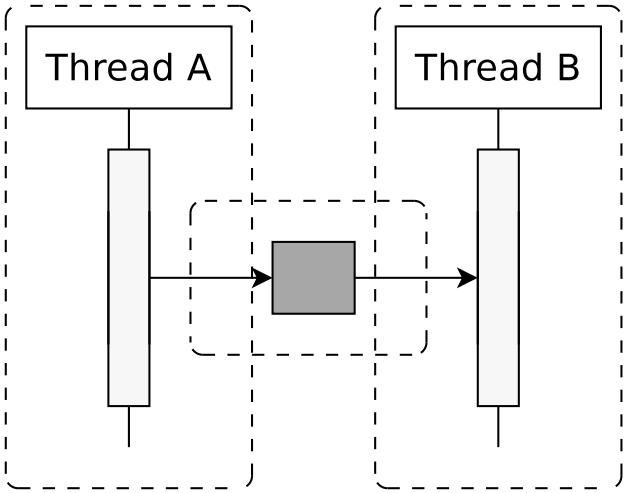
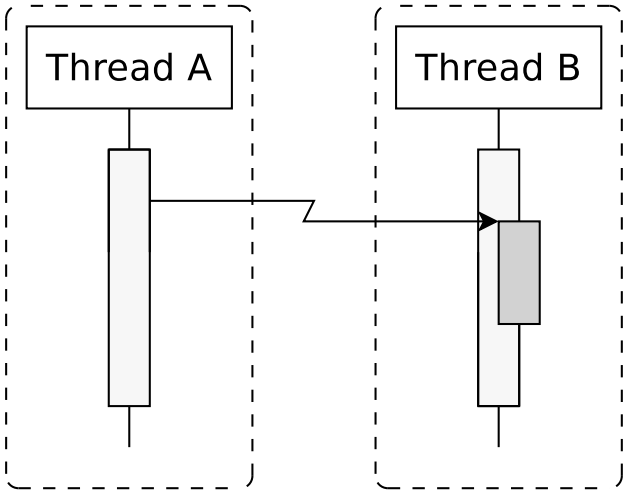
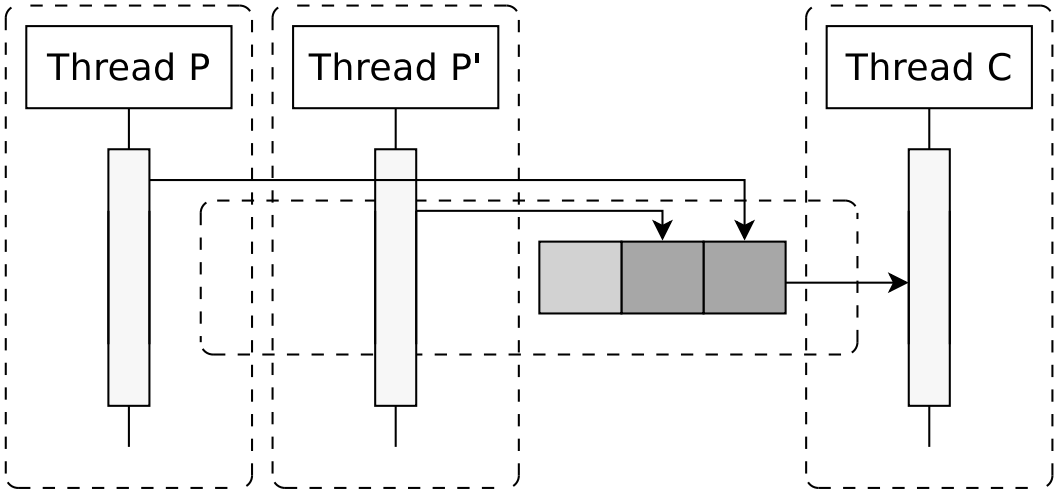


Tableau Noir



Signal Évènement



Producteurs/Consommateur

Inter Processus ?

Mécanismes de communication POSIX

Threads (même espace mémoire) :

- variables partagées

Inter-Processus :

- mémoires partagées (*shm - shared memory*) (*)
- files de messages (*mq - message queue*) (*)
- tubes de communication (*pipe*)
- signaux (*signal*)

(*) IPC (*Inter Process Communication*)

Mécanismes de communication POSIX

Threads (même espace mémoire) :

- variables partagées

Inter-Processus :

- mémoires partagées (*shm - shared memory*) (*)
- files de messages (*mq - message queue*) (*)
- tubes de communication (*pipe*)
- signaux (*signal*)

(*) IPC (*Inter Process Communication*)

Variables partagées

Système POSIX :

- variables globales protégées en accès

→ par des verrous d'exclusion mutuelle

→ par des verrous de lecture/écriture

→ par des sémaphores

Mémoires partagées

Système POSIX :

- **shm** : *shared memory*

→ IPC (*Inter Process Communication*)

→ `ENOSYS` ⇒ recompilation noyau nécessaire

→ accès ressoures IPC : `/dev/shm` (système de fichier virtuel `tmpfs`)

→ Principe :

1. ouvrir un segment de mémoire
2. projeter ce segment dans l'espace mémoire des processus

Programmation :

- option gcc : `-lrt`

Mémoires partagées

Système POSIX :

- **shm** : *shared memory*

→ IPC (*Inter Process Communication*)

→ `ENOSYS` ⇒ recompilation noyau nécessaire

→ accès ressoures IPC : `/dev/shm` (système de fichier virtuel `tmpfs`)

→ Principe :

1. ouvrir un segment de mémoire
2. projeter ce segment dans l'espace mémoire des processus

Programmation :

- option gcc : `-lrt`

Mémoires partagées

Système POSIX :

- **shm** : *shared memory*

→ IPC (*Inter Process Communication*)

→ `ENOSYS` ⇒ recompilation noyau nécessaire

→ accès ressoures IPC : `/dev/shm` (système de fichier virtuel `tmpfs`)

→ Principe :

1. ouvrir un segment de mémoire
2. projeter ce segment dans l'espace mémoire des processus

Programmation :

- option gcc : `-lrt`

Mémoires partagées

Initialisation :

- `sem_t * shm_open(const char * name, int flags, mode_t mode)`

→ `name` : identifiant de la mémoire partagée (commence par '/')

→ `flags` : accès (O_RDONLY, O_WRONLY, O_RDWR) et création (O_CREATE) (cf. `<fcntl.h>`)

→ `mode` : permissions (à la création) (cf. `man 2 open`, `<sys/stat.h>`)

→ retourne un descripteur de segment (∼ descripteur de fichier)

→ dimensionnement avec : `int ftruncate(int fd, off_t length)`

→ fermeture avec : `int close(int fd)`)

Libération mémoire :

- `int shm_unlink(const char * name)`

→ ouverture impossible mais reste disponible pour processus en cours d'utilisation

Mémoires partagées

Initialisation :

- `sem_t * shm_open(const char * name, int flags, mode_t mode)`

→ `name` : identifiant de la mémoire partagée (commence par '/')

→ `flags` : accès (O_RDONLY, O_WRONLY, O_RDWR) et création (O_CREATE) (cf. `<fcntl.h>`)

→ `mode` : permissions (à la création) (cf. `man 2 open`, `<sys/stat.h>`)

→ retourne un descripteur de segment (∼ descripteur de fichier)

→ dimensionnement avec : `int ftruncate(int fd, off_t length)`

→ fermeture avec : `int close(int fd)`)

Libération mémoire :

- `int shm_unlink(const char * name)`

→ ouverture impossible mais reste disponible pour processus en cours d'utilisation

Mémoires partagées

Initialisation :

- `sem_t * shm_open(const char * name, int flags, mode_t mode)`

→ `name` : identifiant de la mémoire partagée (commence par '/')

→ `flags` : accès (O_RDONLY, O_WRONLY, O_RDWR) et création (O_CREATE) (cf. `<fcntl.h>`)

→ `mode` : permissions (à la création) (cf. `man 2 open`, `<sys/stat.h>`)

→ retourne un descripteur de segment (∼ descripteur de fichier)

→ dimensionnement avec : `int ftruncate(int fd, off_t length)`

→ fermeture avec : `int close(int fd)`)

Libération mémoire :

- `int shm_unlink(const char * name)`

→ ouverture impossible mais reste disponible pour processus en cours d'utilisation

Mémoires partagées

Projection dans l'espace mémoire du processus :

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

→ retourne un pointeur sur la zone mémoire allouée pour la projection

→ retourne `MAP_FAILED` si erreur (`errno` renseigné)

→ `addr` : souhait d'adresse pour l'allocation (`NULL` par défaut)

→ `length` : taille de la projection en mémoire

→ `prot` : protection en lecture/écriture (`PROT_READ | PROT_WRITE`)

→ `flags = MAP_SHARED` (mémoire partagée)

→ `fd` : descripteur du segment partagé

→ `offset` : décallage dans le segment partagé (0 par défaut)

Files de messages

Système POSIX :

- **mq** : *message queue*

→ IPC (*Inter Process Communication*)

→ `ENOSYS` ⇒ recompilation noyau nécessaire

→ accès ressoures des files de messages : `/dev/mqueue`

→ transmission de messages (différentes tailles possibles)

→ `man mq_overview`

Programmation :

- `<mqueue.h>`
- options gcc : `-lrt`
- type d'un descripteur de file de message : `mqd_t`

Files de messages

Système POSIX :

- **mq** : *message queue*

→ IPC (*Inter Process Communication*)

→ `ENOSYS` ⇒ recompilation noyau nécessaire

→ accès ressoures des files de messages : `/dev/mqueue`

→ transmission de messages (différentes tailles possibles)

→ `man mq_overview`

Programmation :

- `<mqueue.h>`
- options gcc : `-lrt`
- type d'un descripteur de file de message : `mqd_t`

Files de messages

Système POSIX :

- **mq** : *message queue*

→ IPC (*Inter Process Communication*)

→ `ENOSYS` ⇒ recompilation noyau nécessaire

→ accès ressoures des files de messages : `/dev/mqueue`

→ transmission de messages (différentes tailles possibles)

→ `man mq_overview`

Programmation :

- `<mqueue.h>`
- options gcc : `-lrt`
- type d'un descripteur de file de message : `mqd_t`

Files de messages

Système POSIX :

- **mq** : *message queue*

→ IPC (*Inter Process Communication*)

→ `ENOSYS` ⇒ recompilation noyau nécessaire

→ accès ressoures des files de messages : `/dev/mqueue`

→ transmission de messages (différentes tailles possibles)

→ `man mq_overview`

Programmation :

- `<mqueue.h>`
- options gcc : `-lrt`
- type d'un descripteur de file de message : `mqd_t`

Files de messages

Initialisation :

- `mqd_t mq_open(`
  `const char * name,`
  `int flags,`
  `mode_t mode,`
  `struct mq_attr * attr)`

→ `name` : identifiant de la file de messages (commence par '/')

→ `flags` : accès (O_RDONLY, O_WRONLY, O_RDWR, O_NONBLOCK) et création (O_CREATE) (cf. `<fcntl.h>`)

→ `mode` : permissions (à la création) (cf. `man 2 open`, `<sys/stat.h>`)

→ `attr` : attributs (`NULL` ⇒ valeurs par défaut)

- `mqd_t mq_open(const char * name, int flags)`

→ accès à une file de messages déjà créée

Files de messages

Initialisation :

- `mqd_t mq_open(`
  `const char * name,`
  `int flags,`
  `mode_t mode,`
  `struct mq_attr * attr)`

→ `name` : identifiant de la file de messages (commence par '/')

→ `flags` : accès (O_RDONLY, O_WRONLY, O_RDWR, O_NONBLOCK) et création (O_CREATE) (cf. `<fcntl.h>`)

→ `mode` : permissions (à la création) (cf. `man 2 open`, `<sys/stat.h>`)

→ `attr` : attributs (`NULL` ⇒ valeurs par défaut)

- `mqd_t mq_open(const char * name, int flags)`

→ accès à une file de messages déjà créée

Files de messages

Attributs d'une file de messages :

- type `struct mq_attr` :

→ `mq_flags` : mode d'accès (0 : normal, `O_NONBLOCK` : non bloquant) (*)
→ `mq_maxmsg` : nombre max. de messages dans la file
→ `mq_msgsize` : taille max. d'un message dans la file
→ `mq_curmsgs` : nombre de messages actuellement présents (*)

-
```
int mq_setattr(
        mqd_t mqdes,
        const struct mq_attr * newattr,
        struct mq_attr * oldattr)
```

-
```
int mq_getattr(
        mqd_t mqdes,
        struct mq_attr * attr)
```

(*) ignoré pour `mq_open` ou `mq_setattr` (accessible uniquement en lecture)

Files de messages

Attributs d'une file de messages :

- type `struct mq_attr` :

→ `mq_flags` : mode d'accès (0 : normal, `O_NONBLOCK` : non bloquant) (*)

→ `mq_maxmsg` : nombre max. de messages dans la file

→ `mq_msgsize` : taille max. d'un message dans la file

→ `mq_curmsgs` : nombre de messages actuellement présents (*)

-
```
int mq_setattr(
            mqd_t mqdes,
            const struct mq_attr * newattr,
            struct mq_attr * oldattr)
```

-
```
int mq_getattr(
            mqd_t mqdes,
            struct mq_attr * attr)
```

(*) ignoré pour `mq_open` ou `mq_setattr` (accessible uniquement en lecture)

Files de messages

Libération mémoire :

- `int mq_close(mqd_t mqdes)`

→ fermeture (la file de messages reste à disposition d'autres processus)

- `int mq_unlink(const char * name)`

→ suppression de l'identifiant de la file de messages

→ ouverture impossible mais reste disponible pour processus en cours d'utilisation

→ destruction effective après le dernier `mq_close` (des process en cours d'utilisation)

Files de messages

Libération mémoire :

- `int mq_close(mqd_t mqdes)`

→ fermeture (la file de messages reste à disposition d'autres processus)

- `int mq_unlink(const char * name)`

→ suppression de l'identifiant de la file de messages

→ ouverture impossible mais reste disponible pour processus en cours d'utilisation

→ destruction effective après le dernier `mq_close` (des process en cours d'utilisation)

Files de messages

Envoie de messages :

- `int mq_send(`
  `mqd_t mqdes,`
  `const char * msg_ptr,`
  `size_t msg_len,`
  `unsigned int msg_prio)`

→ `msg_prio` : priorité ($\in$ `[0, sysconf(_SC_MQ_PRIO_MAX) - 1]`)

→ messages empilés dans l'ordre décroissant des priorités

→ appel bloquant si file pleine et file bloquante (en attente d'un espace suffisant)

Files de messages

Réception de messages :

- `size_t mq_receive(`
  `mqd_t mqdes,`
  `char * msg_ptr,`
  `size_t msg_len,`
  `unsigned int * msg_prio)`

→ retourne le nombre d'octets reçus

→ `msg_prio` : donne la priorité du message reçu (si $\neq$ `NULL`)

→ messages dépilés dans l'ordre décroissant des priorités

→ appel bloquant si pas de message et file bloquante (en attente d'un message)

Rq. taille buffer réception $\geq$ taille max. message possible dans file
(cf. `mq_getattr`, attribut `mq_msgsize`, = 8192 par défaut sous Linux)

Files de messages

Réception de messages :

- `size_t mq_receive(`
  `mqd_t mqdes,`
  `char * msg_ptr,`
  `size_t msg_len,`
  `unsigned int * msg_prio)`

→ retourne le nombre d'octets reçus

→ `msg_prio` : donne la priorité du message reçu (si $\neq$ `NULL`)

→ messages dépilés dans l'ordre décroissant des priorités

→ appel bloquant si pas de message et file bloquante (en attente d'un message)

Rq. taille buffer réception $\geq$ taille max. message possible dans file
(cf. `mq_getattr`, attribut `mq_msgsize`, $=$ 8192 par défaut sous Linux)

Files de messages

Notification de l'arrivée d'un message :

- `int mq_notify(`
  `mqd_t mqdes,`
  `const struct sigevent * sevp)`

→ enregistrement pour notification quand un message est disponible

→ `sevp` : configuration du type de notification

→ signal ou exécution d'une fonction par un thread

→ cf. `man sigevent`

Tubes de communication

Système POSIX :

- **pipe** : *pipe object*

→ transmission d'octets sous forme de flux

→ un descripteur d'entrée (écriture) et un descripteur de sortie (lecture)

→ communication unidirectionnelle et FIFO

→ `PIPE_BUF` (`<sys/limits.h>`) taille max. bloc données écrit atomique

→ **pipe** : tube "anonyme"

→ **named pipe** : tube nommé

→ `man 7 pipe`

Tubes de communication ”anonymes”

Programmation :

- `int pipe(int pipefd[2])`

→ `<unistd.h>`, `man 2 pipe`

→ création d'un tube de communication, paramètre renseigné en sortie de fonction

→ `pipefd[0]` : descripteur de sortie, lecture (*ro*)

→ `pipefd[1]` : descripteur d'entrée, écriture (*wo*)

→ $\sim$ descripteurs de fichier :
écriture : `man 3 write`, lecture : `man 3 read`, fermeture : `man 3 close`

→ tube en mémoire (pas sur le disque)

Rq. lien de filiation nécessaire entre les processus communicant par tube ”anonyme”
(car nécessité d'avoir eu une transmission des descripteurs d'entrée et de sortie)

Tubes de communication ”anonymes”

Programmation :

- `int pipe(int pipefd[2])`

→ `<unistd.h>`, `man 2 pipe`

→ création d’un tube de communication, paramètre renseigné en sortie de fonction

→ `pipefd[0]` : descripteur de sortie, lecture (*ro*)

→ `pipefd[1]` : descripteur d’entrée, écriture (*wo*)

→ $\sim$ descripteurs de fichier :
écriture : `man 3 write`, lecture : `man 3 read`, fermeture : `man 3 close`

→ tube en mémoire (pas sur le disque)

Rq. lien de filiation nécessaire entre les processus communicant par tube ”anonyme”
(car nécessité d’avoir eu une transmission des descripteurs d’entrée et de sortie)

Tubes de communication nommés

Programmation :

- `int mkfifo(const char * pathname, mode_t mode)`

→ `man 3 mkfifo` (utilitaire : `man 1 mkfifo`)

→ création d'un tube nommé (accessible par différents processus)

→ `pathname` : emplacement sur le disque

→ `mode` : permissions (à la création) (cf. `man 2 open`, `<sys/stat.h>`)

→ retourne un descripteur de fichier :
écriture : `man 3 write`, lecture : `man 3 read`, fermeture : `man 3 close`

→ ouverture d'un tube déjà créé (via `pathname`) : `man 3 open`

→ suppression d'un tube (sur le disque) : `man 3 unlink`

Signaux

Système POSIX :

- **signal** : *signal event*

→ communication inter-processus

→ liste de signaux standardisés

→ envoi d'un signal à un processus ⇒≠ comportements possibles :

1→ laisser le système traiter le signal (avec comportement par défaut)

2→ interrompre le processus pour exéc. d'une routine (gestionnaire de signal)

3→ ignorer le signal

Programmation :

- définition des signaux : `signal.h`

→ nombre de signaux : `NSIG`

→ chaque signal identifié par un numéro associé à un nom symbolique

→ utilisation des noms (constantes symboliques) à privilégier pour portabilité

Signaux

Système POSIX :

- **signal** : *signal event*

→ communication inter-processus

→ liste de signaux standardisés

→ envoi d'un signal à un processus ⇒≠ comportements possibles :

1→ laisser le système traiter le signal (avec comportement par défaut)

2→ interrompre le processus pour exéc. d'une routine (gestionnaire de signal)

3→ ignorer le signal

Programmation :

- définition des signaux : `signal.h`

→ nombre de signaux : `NSIG`

→ chaque signal identifié par un numéro associé à un nom symbolique

→ utilisation des noms (constantes symboliques) à privilégier pour portabilité

Signaux

Exemples de signaux : (cf. `man 7 signal`)

`SIGABORT`	*Core*	suite à un `abort`
`SIGALRM`	*Term*	suite à expiration `alarm` ou `setitimer`
`SIGSEGV`	*Core*	erreur de segmentation
`SIGCHLD`	*Ign*	un fils du processus est terminé ou stoppé
`SIGFPE`	*Core*	problème calcul arithmétique (*floating-point exception*)
`SIGINT`	*Term*	interruption depuis le clavier (ex : *Ctrl* + *c*)
`SIGKILL`	*Term*	tuer un processus (dernier recours, ni capturable ni ignorable)
`SIGQUIT`	*Core*	interruption depuis le clavier (ex : *Ctrl* + \)
`SIGSTOP`	*Stop*	suspendre un processus (ni capturable ni ignorable)
`SIGCONT`	*Cont*	relancer un processus stoppé (même si capturé ou ignoré)
`SIGTERM`	*Term*	terminer un processus
`SIGUSR1`	*Term*	à la disposition du programmeur (avec `SIGUSR2`)

Signaux

Emission d'un signal :

- `int kill(pid_t pid, int sig)`

→ `man 3 kill` (utilitaire : `man 1 kill`)

→ `pid` : PID du processus visé (ensemble de processus possible)

→ `sig` : numéro du signal à envoyer (utiliser constante symbolique)

→ pas d'empilement des signaux

Réception d'un signal :

- alternative :

→ `signal` : simple, mais attention à la compatibilité

→ `sigaction` : plus sophistiqué, mais à privilégier

Signaux

Emission d'un signal :

- `int kill(pid_t pid, int sig)`

→ `man 3 kill` (utilitaire : `man 1 kill`)

→ `pid` : PID du processus visé (ensemble de processus possible)

→ `sig` : numéro du signal à envoyer (utiliser constante symbolique)

→ pas d'empilement des signaux

Réception d'un signal :

- alternative :

→ `signal` : simple, mais attention à la compatibilité

→ `sigaction` : plus sophistiqué, mais à privilégier

Signaux

Réception d'un signal avec `signal` :

- `void (*signal(int sig, void (*func)(int)))(int)`

~ `typedef void (* func_t)(int);`
`func_t signal(int sig, func_t func);`

→ `sig` : numéro du signal concerné (utiliser constante symbolique)

→ `func` : pointeur gestionnaire de signal (fonction avec signal en paramètre)

→ retourne pointeur sur ancien gestionnaire de signal ou `SIG_ERR` (`errno` renseigné)

→ constantes symboliques de gestionnaires de signal : `SIG_IGN` et `SIG_DFL`

→ blocage du signal en cours de gestion

→ appels systèmes courts non interrompus, appels systèmes lents interrompus

- `int siginterrupt(int sig, int flag)`

→ `flag` $= 0 \Rightarrow$ appels systèmes lents interrompus relancés auto. (défaut sous Linux)

→ `flag` $\neq 0 \Rightarrow$ appels systèmes lents interrompus échouent (`errno = EINTR`)

→ doit être appelée après l'installation d'un gestionnaire de signal

Signaux

Réception d'un signal avec `signal` :

- `void (*signal(int sig, void (*func)(int)))(int)`

~ `typedef void (* func_t)(int);`
`func_t signal(int sig, func_t func);`

→ `sig` : numéro du signal concerné (utiliser constante symbolique)

→ `func` : pointeur gestionnaire de signal (fonction avec signal en paramètre)

→ retourne pointeur sur ancien gestionnaire de signal ou `SIG_ERR` (`errno` renseigné)

→ constantes symboliques de gestionnaires de signal : `SIG_IGN` et `SIG_DFL`

→ blocage du signal en cours de gestion

→ appels systèmes courts non interrompus, appels systèmes lents interrompus

- `int siginterrupt(int sig, int flag)`

→ `flag` $= 0 \Rightarrow$ appels systèmes lents interrompus relancés auto. (défaut sous Linux)

→ `flag` $\neq 0 \Rightarrow$ appels systèmes lents interrompus échouent (`errno = EINTR`)

→ doit être appelée après l'installation d'un gestionnaire de signal

Signaux

Réception d'un signal avec **`signal`** :

- `void (*signal(int sig, void (*func)(int)))(int)`

~ `typedef void (* func_t)(int);`
`func_t signal(int sig, func_t func);`

→ `sig` : numéro du signal concerné (utiliser constante symbolique)

→ `func` : pointeur gestionnaire de signal (fonction avec signal en paramètre)

→ retourne pointeur sur ancien gestionnaire de signal ou `SIG_ERR` (`errno` renseigné)

→ constantes symboliques de gestionnaires de signal : `SIG_IGN` et `SIG_DFL`

→ blocage du signal en cours de gestion

→ appels systèmes courts non interrompus, appels systèmes lents interrompus

- `int siginterrupt(int sig, int flag)`

→ `flag` $= 0 \Rightarrow$ appels systèmes lents interrompus relancés auto. (défaut sous Linux)

→ `flag` $\neq 0 \Rightarrow$ appels systèmes lents interrompus échouent (`errno = EINTR`)

→ doit être appelée après l'installation d'un gestionnaire de signal

Signaux

Réception d'un signal avec **`signal`** :

- `void (*signal(int sig, void (*func)(int)))(int)`

~ `typedef void (* func_t)(int);`
`func_t signal(int sig, func_t func);`

→ `sig` : numéro du signal concerné (utiliser constante symbolique)

→ `func` : pointeur gestionnaire de signal (fonction avec signal en paramètre)

→ retourne pointeur sur ancien gestionnaire de signal ou `SIG_ERR` (`errno` renseigné)

→ constantes symboliques de gestionnaires de signal : `SIG_IGN` et `SIG_DFL`

→ blocage du signal en cours de gestion

→ appels systèmes courts non interrompus, appels systèmes lents interrompus

- `int siginterrupt(int sig, int flag)`

→ `flag` $= 0 \Rightarrow$ appels systèmes lents interrompus relancés auto. (défaut sous Linux)

→ `flag` $\neq 0 \Rightarrow$ appels systèmes lents interrompus échouent (`errno = EINTR`)

→ doit être appelée après l'installation d'un gestionnaire de signal

Signaux

Réception d'un signal avec `sigaction` :

- `int sigaction(int sig, const struct sigaction * act, struct sigaction * oldact)`

→ `sig` : numéro du signal concerné (utiliser constante symbolique)

→ `act` : pointeur sur la structure à utiliser (`NULL` ⇒ pas de modification)

→ `oldact` : pointeur sur ancienne structure (pour sauvegarde, `NULL` ⇒ pas de sauvegarde)

- `struct sigaction`

→ `sa_handler` : pointeur sur le gestionnaire de signal (`SIG_IGN` et `SIG_DFL` possibles)

→ `sa_sigaction` : pointeur sur un gestionnaire de signal "avancé"

→ `sa_mask` : listes signaux bloqués durant exécution gestionnaire (type `sigset_t`)

→ `sa_flags` : configuration du gestionnaire

→ `SA_NODEFER` : ne pas bloquer le signal en cours de gestion

→ `SA_RESTART` : appels systèmes lents automatiquement relancés

→ `SA_NOCLDSTOP` : pour `SIGCHLD`, appelé si fils terminé (pas stoppé)

→ `SA_SIGINFO` : utilisation du gestionnaire de signal "avancé"

Signaux

Réception d'un signal avec **sigaction** :

- `int sigaction(int sig, const struct sigaction * act, struct sigaction * oldact)`

→ `sig` : numéro du signal concerné (utiliser constante symbolique)

→ `act` : pointeur sur la structure à utiliser (`NULL` ⇒ pas de modification)

→ `oldact` : pointeur sur ancienne structure (pour sauvegarde, `NULL` ⇒ pas de sauvegarde)

- `struct sigaction`

→ `sa_handler` : pointeur sur le gestionnaire de signal (`SIG_IGN` et `SIG_DFL` possibles)

→ `sa_sigaction` : pointeur sur un gestionnaire de signal "avancé"

→ `sa_mask` : listes signaux bloqués durant exécution gestionnaire (type `sigset_t`)

→ `sa_flags` : configuration du gestionnaire

→ `SA_NODEFER` : ne pas bloquer le signal en cours de gestion

→ `SA_RESTART` : appels systèmes lents automatiquement relancés

→ `SA_NOCLDSTOP` : pour `SIGCHLD`, appelé si fils terminé (pas stoppé)

→ `SA_SIGINFO` : utilisation du gestionnaire de signal "avancé"

Signaux

Réception d'un signal avec `sigaction` :

- `int sigaction(int sig, const struct sigaction * act, struct sigaction * oldact)`

→ `sig` : numéro du signal concerné (utiliser constante symbolique)

→ `act` : pointeur sur la structure à utiliser (`NULL` ⇒ pas de modification)

→ `oldact` : pointeur sur ancienne structure (pour sauvegarde, `NULL` ⇒ pas de sauvegarde)

- `struct sigaction`

→ `sa_handler` : pointeur sur le gestionnaire de signal (`SIG_IGN` et `SIG_DFL` possibles)

→ `sa_sigaction` : pointeur sur un gestionnaire de signal "avancé"

→ `sa_mask` : listes signaux bloqués durant exécution gestionnaire (type `sigset_t`)

→ `sa_flags` : configuration du gestionnaire

  → `SA_NODEFER` : ne pas bloquer le signal en cours de gestion

  → `SA_RESTART` : appels systèmes lents automatiquement relancés

  → `SA_NOCLDSTOP` : pour `SIGCHLD`, appelé si fils terminé (pas stoppé)

  → `SA_SIGINFO` : utilisation du gestionnaire de signal "avancé"

Signaux

Manipulation des listes de signaux :

- type : `sigset_t`

→ `int sigemptyset(sigset_t * set)` : vider un ensemble

→ `int sigfillset(sigset_t * set)` : remplir un ensemble (tous les signaux)

→ `int sigaddset(sigset_t * set, int signum)` : ajouter un signal

→ `int sigdelset(sigset_t * set, int signum)` : enlever un signal

→ `int sigismember(const sigset_t * set, int signum)` :
retourne 1 si le signal appartient à l'ensemble, 0 sinon

Signaux

Blocage des signaux :

- `int sigprocmask(`
  `int how,`
  `const sigset_t * set,`
  `sigset_t * oldset)`

→ bloquer/débloquer des signaux ou consulter le masque de blocage

→ `how` : action attendue (`SIG_BLOCK, SIG_UNBLOCK, SIG_SETMASK`)

→ `set` : ensemble de signaux (à ajouter, soustraire ou substituer au masque)

→ `oldset` : ancien masque (pour sauvegarde, si non `NULL`)

Signaux

Liste des signaux en attente :

- `int sigpending(sigset_t * set)`

→ `set` : ensemble des signaux en attente

Attente d'un signal :

- `int sigsuspend(const sigset_t * mask)`

→ blocage en attente d'un signal

→ `mask` : ensemble des signaux bloqués (pas ceux qu'on attend)

→ retourne -1, avec `errno = EINTR`

Signaux

Liste des signaux en attente :

- `int sigpending(sigset_t * set)`

→ `set` : ensemble des signaux en attente

Attente d'un signal :

- `int sigsuspend(const sigset_t * mask)`

→ blocage en attente d'un signal

→ `mask` : ensemble des signaux bloqués (pas ceux qu'on attend)

→ retourne -1, avec `errno = EINTR`

Signaux

Bonnes pratiques pour un gestionnaire de signal

- ANSI C : modification variables globales type `sig_atomic_t`

→ garantie d'accés atomique, sans interruption par un signal

→ + indicateur `volatile` sur la variable

- gestion blocage signaux permet tout accès variables globales

→ ex : `sa_mask` de `sigaction`

- appels systèmes : `signal-safety` (`async-signal-safe`)

→ + sauvegarder/restauter `errno` en entrée/sortie du gestionnaire

Pour aller plus loin :

- sauts non locaux : `sigsetjump`, `siglongjump`
- signaux temps réel

→ extension de `SIGUSR1` et `SIGUSR2`

→ empilement des occurences des signaux

→ priorité sur les signaux (+ utilisation gestionnaire signal "avancé" `sa_sigaction` de `sigaction`)

Signaux

Bonnes pratiques pour un gestionnaire de signal

- ANSI C : modification variables globales type `sig_atomic_t`

→ garantie d'accés atomique, sans interruption par un signal

→ + indicateur `volatile` sur la variable

- gestion blocage signaux permet tout accès variables globales

→ ex : `sa_mask` de `sigaction`

- appels systèmes : `signal-safety` (`async-signal-safe`)

→ + sauvegarder/restauter `errno` en entrée/sortie du gestionnaire

Pour aller plus loin :

- sauts non locaux : `sigsetjump`, `siglongjump`
- signaux temps réel

→ extension de `SIGUSR1` et `SIGUSR2`

→ empilement des occurences des signaux

→ priorité sur les signaux (+ utilisation gestionnaire signal "avancé" `sa_sigaction` de `sigaction`)

Signaux

Bonnes pratiques pour un gestionnaire de signal

- ANSI C : modification variables globales type `sig_atomic_t`

→ garantie d'accés atomique, sans interruption par un signal

→ + indicateur `volatile` sur la variable

- gestion blocage signaux permet tout accès variables globales

→ ex : `sa_mask` de `sigaction`

- appels systèmes : `signal-safety` (`async-signal-safe`)

→ + sauvegarder/restauter `errno` en entrée/sortie du gestionnaire

Pour aller plus loin :

- sauts non locaux : `sigsetjump`, `siglongjump`
- signaux temps réel

→ extension de `SIGUSR1` et `SIGUSR2`

→ empilement des occurences des signaux

→ priorité sur les signaux (+ utilisation gestionnaire signal "avancé" `sa_sigaction` de `sigaction`)

Signaux

Bonnes pratiques pour un gestionnaire de signal

- ANSI C : modification variables globales type `sig_atomic_t`

→ garantie d'accés atomique, sans interruption par un signal

→ + indicateur `volatile` sur la variable

- gestion blocage signaux permet tout accès variables globales

→ ex : `sa_mask` de `sigaction`

- appels systèmes : `signal-safety` (`async-signal-safe`)

→ + sauvegarder/restauter `errno` en entrée/sortie du gestionnaire

Pour aller plus loin :

- sauts non locaux : `sigsetjump`, `siglongjump`
- signaux temps réel

→ extension de `SIGUSR1` et `SIGUSR2`

→ empilement des occurences des signaux

→ priorité sur les signaux (+ utilisation gestionnaire signal "avancé" `sa_sigaction` de `sigaction`)

Signaux

Bonnes pratiques pour un gestionnaire de signal

- ANSI C : modification variables globales type `sig_atomic_t`

→ garantie d'accés atomique, sans interruption par un signal

→ + indicateur `volatile` sur la variable

- gestion blocage signaux permet tout accès variables globales

→ ex : `sa_mask` de `sigaction`

- appels systèmes : `signal-safety` (`async-signal-safe`)

→ + sauvegarder/restauter `errno` en entrée/sortie du gestionnaire

Pour aller plus loin :

- sauts non locaux : `sigsetjump`, `siglongjump`
- signaux temps réel

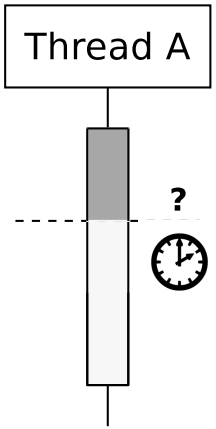
→ extension de `SIGUSR1` et `SIGUSR2`

→ empilement des occurences des signaux

→ priorité sur les signaux (+ utilisation gestionnaire signal "avancé" `sa_sigaction` de `sigaction`)

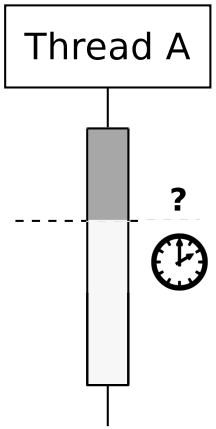
Besoins en temporisation

Besoins en temporisation

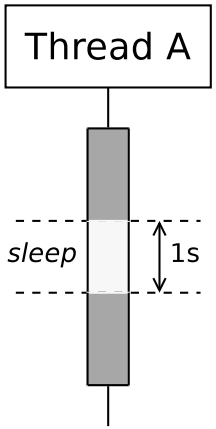


Horodatage

Besoins en temporisation

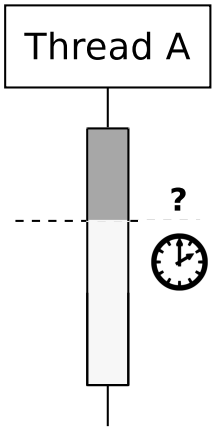


Horodatage

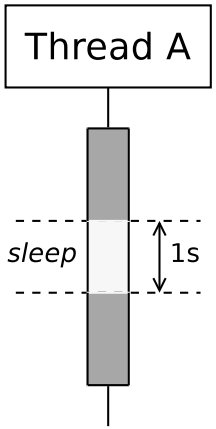


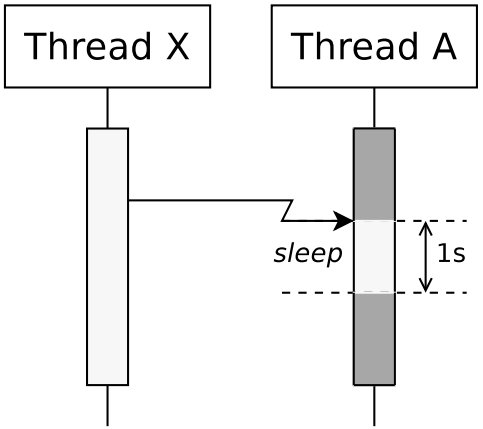
Endormissement

Besoins en temporisation



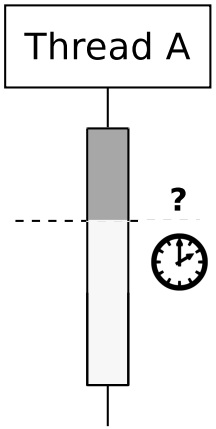
Horodatage



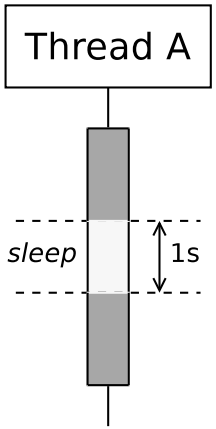


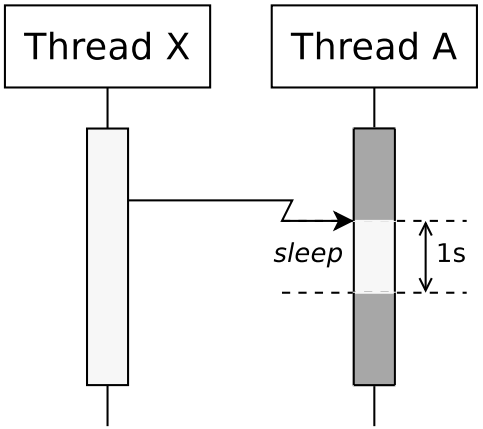
Endormissement

Besoins en temporisation

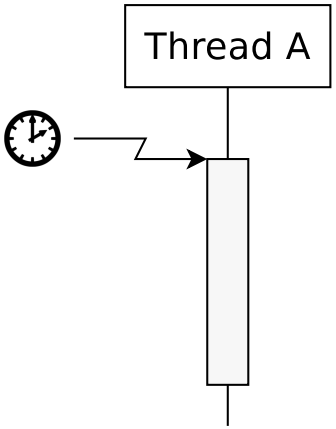


Horodatage



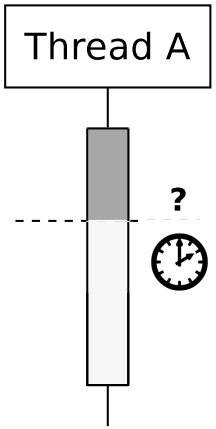


Endormissement

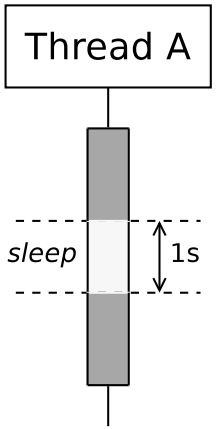


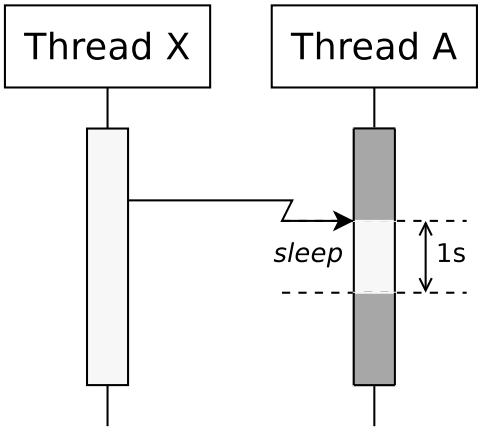
Alarme

Besoins en temporisation

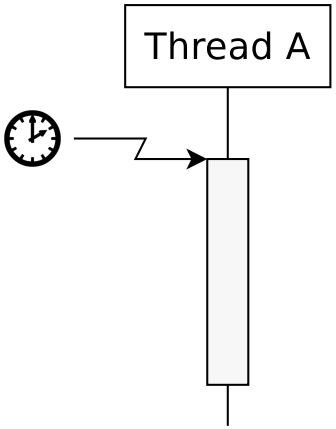


Horodatage

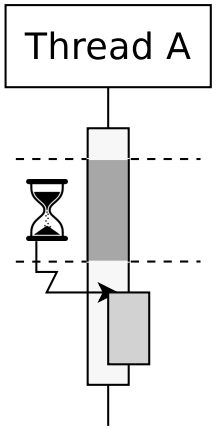




Endormissement

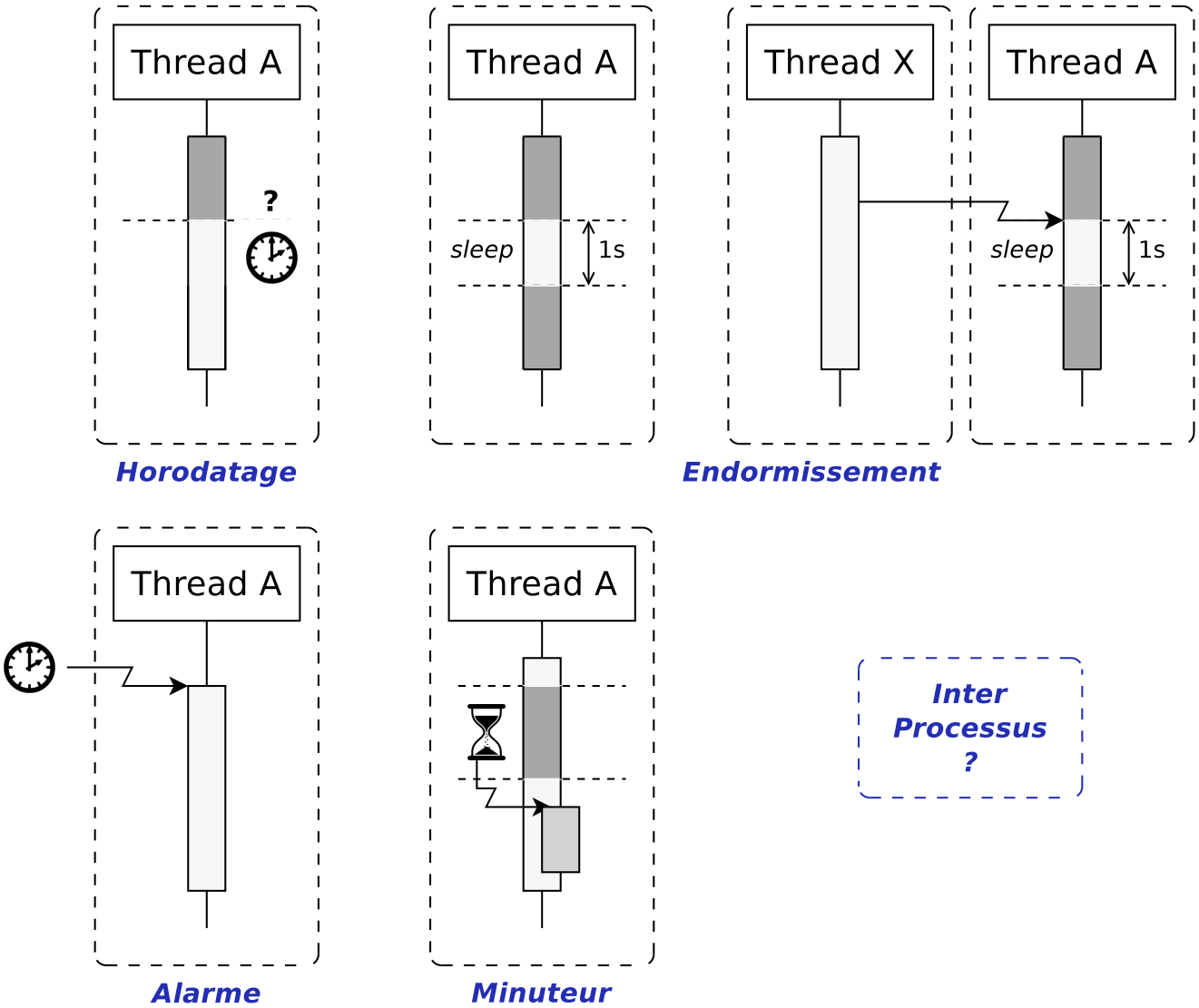


Alarme



Minuteur

Besoins en temporisation



Mécanismes de temporisation POSIX

Threads :

- horodatage (*time*)
- temps processeur (*clock*)
- endormissement (*sleep*)
- alarmes (*alarm*)
- minuteurs (*timer*)

Horodatage

Système POSIX :

- 3 types d'horadatage :

→ **`time_t`** : type simple, précision à la seconde

→ **`timeval`** : type structuré, précision à la microseconde

→ **`timespec`** : type structuré, précision à la nanoseconde

- temps initial de référence : Epoch

→ Unix : 01/01/1970 0h00 TU (ex : Linux, MacOS)

→ limite sur 32 bits : 19/01/2038 3h14 :7s (retour au vendredi 13/12/1901 20h45 :52s)

→ variation de Epoch sur des systèmes non 100% compatibles POSIX

- cf. `man 7 time`

Horodatage

Système POSIX :

- 3 types d'horadatage :

→ **`time_t`** : type simple, précision à la seconde

→ **`timeval`** : type structuré, précision à la microseconde

→ **`timespec`** : type structuré, précision à la nanoseconde

- temps initial de référence : Epoch

→ Unix : 01/01/1970 0h00 TU (ex : Linux, MacOS)

→ limite sur 32 bits : 19/01/2038 3h14 :7s (retour au vendredi 13/12/1901 20h45 :52s)

→ variation de Epoch sur des systèmes non 100% compatibles POSIX

- cf. `man 7 time`

Horodatage

Système POSIX :

- 3 types d'horadatage :

→ **`time_t`** : type simple, précision à la seconde

→ **`timeval`** : type structuré, précision à la microseconde

→ **`timespec`** : type structuré, précision à la nanoseconde

- temps initial de référence : Epoch

→ Unix : 01/01/1970 0h00 TU (ex : Linux, MacOS)

→ limite sur 32 bits : 19/01/2038 3h14 :7s (retour au vendredi 13/12/1901 20h45 :52s)

→ variation de Epoch sur des systèmes non 100% compatibles POSIX

- cf. `man 7 time`

Horodatage

Programmation avec **time_t** :

- définition : `time.h`

→ type signé (pour distinction code erreur en retour de fonction)

→ opérations arithmétiques (`difftime` recommandée pour évolutivité)

- conversions calendaires (`struct tm`) :

→ `time_t` ↔ `struct tm` : `localtime`, `gmtime`, `mktime`

→ affichage : `ctime` (pour `time_t`), `asctime` ou `strftime` (pour `struct tm`)

- obtenir un horodatage : `time_t time(time_t * tloc)`

→ retourne le nombre de secondes depuis Epoch

→ `tloc` : stoquage de la valeur retour (si ≠ `NULL`)

→ cf. `man 3 time`

Horodatage

Programmation avec **time_t** :

- définition : `time.h`

→ type signé (pour distinction code erreur en retour de fonction)

→ opérations arithmétiques (`difftime` recommandée pour évolutivité)

- conversions calendaires (`struct tm`) :

→ `time_t` ↔ `struct tm` : `localtime`, `gmtime`, `mktime`

→ affichage : `ctime` (pour `time_t`), `asctime` ou `strftime` (pour `struct tm`)

- obtenir un horodatage : `time_t time(time_t * tloc)`

→ retourne le nombre de secondes depuis Epoch

→ `tloc` : stoquage de la valeur retour (si ≠ `NULL`)

→ cf. `man 3 time`

Horodatage

Programmation avec **time_t** :

- définition : `time.h`

→ type signé (pour distinction code erreur en retour de fonction)

→ opérations arithmétiques (`difftime` recommandée pour évolutivité)

- conversions calendaires (`struct tm`) :

→ `time_t` ↔ `struct tm` : `localtime`, `gmtime`, `mktime`

→ affichage : `ctime` (pour `time_t`), `asctime` ou `strftime` (pour `struct tm`)

- obtenir un horodatage : `time_t time(time_t * tloc)`

→ retourne le nombre de secondes depuis Epoch

→ `tloc` : stoquage de la valeur retour (si ≠ `NULL`)

→ cf. `man 3 time`

Horodatage

Programmation avec **timeval** :

- définition : `sys/time.h`
- type `struct timeval` :

→ `tv_sec` : en secondes (type `time_t`)

→ `tv_usec` : en microsecondes (type `time_t`)

- fonctions arithmétiques :

→ `timeradd`, `timersub`, `timercmp`, `timerclear`, `timerisset`

- obtenir un horodatage :

```
int gettimeofday(
        struct timeval * tv,
        struct timezone * tz)
```

→ `tv` : nombre de secondes + microsecondes depuis Epoch (si $\neq$ `NULL`)

→ `tz` : donne des informations sur la zone horaire (si $\neq$ `NULL`) (obsolète)

Horodatage

Programmation avec **timeval** :

- définition : `sys/time.h`
- type `struct timeval` :

→ `tv_sec` : en secondes (type `time_t`)

→ `tv_usec` : en microsecondes (type `time_t`)

- fonctions arithmétiques :

→ `timeradd`, `timersub`, `timercmp`, `timerclear`, `timerisset`

- obtenir un horodatage :

```
int gettimeofday(
        struct timeval * tv,
        struct timezone * tz)
```

→ `tv` : nombre de secondes + microsecondes depuis Epoch (si $\neq$ `NULL`)

→ `tz` : donne des informations sur la zone horaire (si $\neq$ `NULL`) (obsolète)

Horodatage

Programmation avec **timeval** :

- définition : `sys/time.h`
- type `struct timeval` :

→ `tv_sec` : en secondes (type `time_t`)

→ `tv_usec` : en microsecondes (type `time_t`)

- fonctions arithmétiques :

→ `timeradd`, `timersub`, `timercmp`, `timerclear`, `timerisset`

- obtenir un horodatage :

```
int gettimeofday(
        struct timeval * tv,
        struct timezone * tz)
```

→ `tv` : nombre de secondes + microsecondes depuis Epoch (si $\neq$ `NULL`)

→ `tz` : donne des informations sur la zone horaire (si $\neq$ `NULL`) (obsolète)

Horodatage

Programmation avec **timespec** :

- définition : `time.h`
- type `struct timespec` :

→ `tv_sec` : en secondes (type `time_t`)

→ `tv_nsec` : en nanosecondes (type `long`)

- obtenir un horodatage :

```
int clock_gettime(
        clockid_t clk_id,
        struct timespec * tp)
```

→ `clk_id` : type d'horloge (`CLOCK_REALTIME`, `CLOCK_MONOTONIC`, etc.)

→ `tp` : nombre de secondes + nanosecondes depuis Epoch (si $\neq$ `NULL`)

→ option gcc : `-lrt`

- obtenir la résolution : `clock_getres`

Horodatage

Programmation avec **timespec** :

- définition : `time.h`
- type `struct timespec` :

→ `tv_sec` : en secondes (type `time_t`)

→ `tv_nsec` : en nanosecondes (type `long`)

- obtenir un horodatage :

```
int clock_gettime(
        clockid_t clk_id,
        struct timespec * tp)
```

→ `clk_id` : type d'horloge (`CLOCK_REALTIME`, `CLOCK_MONOTONIC`, etc.)

→ `tp` : nombre de secondes + nanosecondes depuis Epoch (si $\neq$ `NULL`)

→ option gcc : `-lrt`

- obtenir la résolution : `clock_getres`

Horodatage

Programmation avec **timespec** :

- définition : `time.h`
- type `struct timespec` :

→ `tv_sec` : en secondes (type `time_t`)

→ `tv_nsec` : en nanosecondes (type `long`)

- obtenir un horodatage :

```
int clock_gettime(
        clockid_t clk_id,
        struct timespec * tp)
```

→ `clk_id` : type d'horloge (`CLOCK_REALTIME`, `CLOCK_MONOTONIC`, etc.)

→ `tp` : nombre de secondes + nanosecondes depuis Epoch (si $\neq$ `NULL`)

→ option gcc : `-lrt`

- obtenir la résolution : `clock_getres`

Temps processeur

Système POSIX :

- temps calculés sur base de *ticks* d'horloge (*clock*)

→ principe : t_d et t_f pris sur une horloge, durée $= t_f - t_d$

Programmation :

- `clock_t clock(void)`

→ retourne un temps processeur écoulé (impulsions d'horloge théorique)

→ temps en secondes : / `CLOCKS_PER_SEC`

- `clock_t times(struct tms * buffer)`

→ retourne nb *jiffies* (nb cycles horloge exécutés depuis démarrage système)

→ temps en secondes : / `sysconf(_SC_CLK_TCK)`

→ `buffer` : renseigné au retour (si $\neq$ `NULL`)

→ `tms_utime` : temps processeur passé en mode utilisateur (type `clock_t`)

→ `tms_stime` : temps processeur passé en mode noyau (type `clock_t`)

→ `tms_cutime` : `tms_utime` + `tms_cutime` des processus fils

→ `tms_cstime` : `tms_stime` + `tms_cstime` des processus fils

Temps processeur

Système POSIX :

- temps calculés sur base de *ticks* d'horloge (*clock*)

→ principe : t_d et t_f pris sur une horloge, durée $= t_f - t_d$

Programmation :

- `clock_t clock(void)`

→ retourne un temps processeur écoulé (impulsions d'horloge théorique)

→ temps en secondes : / `CLOCKS_PER_SEC`

- `clock_t times(struct tms * buffer)`

→ retourne nb *jiffies* (nb cycles horloge exécutés depuis démarrage système)

→ temps en secondes : / `sysconf(_SC_CLK_TCK)`

→ `buffer` : renseigné au retour (si $\neq$ `NULL`)

→ `tms_utime` : temps processeur passé en mode utilisateur (type `clock_t`)

→ `tms_stime` : temps processeur passé en mode noyau (type `clock_t`)

→ `tms_cutime` : `tms_utime` + `tms_cutime` des processus fils

→ `tms_cstime` : `tms_stime` + `tms_cstime` des processus fils

Temps processeur

Système POSIX :

- temps calculés sur base de *ticks* d'horloge (*clock*)

→ principe : t_d et t_f pris sur une horloge, durée $= t_f - t_d$

Programmation :

- `clock_t clock(void)`

→ retourne un temps processeur écoulé (impulsions d'horloge théorique)

→ temps en secondes : / `CLOCKS_PER_SEC`

- `clock_t times(struct tms * buffer)`

→ retourne nb *jiffies* (nb cycles horloge exécutés depuis démarrage système)

→ temps en secondes : / `sysconf(_SC_CLK_TCK)`

→ `buffer` : renseigné au retour (si $\neq$ `NULL`)

→ `tms_utime` : temps processeur passé en mode utilisateur (type `clock_t`)

→ `tms_stime` : temps processeur passé en mode noyau (type `clock_t`)

→ `tms_cutime` : `tms_utime` + `tms_cutime` des processus fils

→ `tms_cstime` : `tms_stime` + `tms_cstime` des processus fils

Temps processeur

Système POSIX :

- temps calculés sur base de *ticks* d'horloge (*clock*)

→ principe : t_d et t_f pris sur une horloge, durée $= t_f - t_d$

Programmation :

- `clock_t clock(void)`

→ retourne un temps processeur écoulé (impulsions d'horloge théorique)

→ temps en secondes : / `CLOCKS_PER_SEC`

- `clock_t times(struct tms * buffer)`

→ retourne nb *jiffies* (nb cycles horloge exécutés depuis démarrage système)

→ temps en secondes : / `sysconf(_SC_CLK_TCK)`

→ `buffer` : renseigné au retour (si $\neq$ `NULL`)

→ `tms_utime` : temps processeur passé en mode utilisateur (type `clock_t`)

→ `tms_stime` : temps processeur passé en mode noyau (type `clock_t`)

→ `tms_cutime` : `tms_utime` + `tms_cutime` des processus fils

→ `tms_cstime` : `tms_stime` + `tms_cstime` des processus fils

Endormissement

Système POSIX :

- 3 granularités de sommeil (*sleep*) : *sec*, μ*sec*, η*sec*

→ précision effective $\sim$ dizaines de msec. sur systèmes et matériels courants

→ pas de mécanisme *ad hoc* pour endormir un thread depuis un autre thread

Programmation :

- `unsigned int sleep(unsigned int sec)`

→ retournent temps restant si interruption par un signal

- `int usleep(useconds_t microsec)` (obsolete)
- `int nanosleep(const struct timespec * req, struct timespec * rem)`

→ `req` : durée de sommeil souhaitée

→ `rem` : durée de sommeil restante si interruption par un signal (et si $\neq$ `NULL`)

→ pour choix horloge et délai relatif ou absolu : `clock_nanosleep`

Endormissement

Système POSIX :

- 3 granularités de sommeil (*sleep*) : *sec*, μ*sec*, η*sec*

→ précision effective $\sim$ dizaines de msec. sur systèmes et matériels courants

→ pas de mécanisme *ad hoc* pour endormir un thread depuis un autre thread

Programmation :

- `unsigned int sleep(unsigned int sec)`

→ retournent temps restant si interruption par un signal

- `int usleep(useconds_t microsec)` (obsolete)
- `int nanosleep(const struct timespec * req, struct timespec * rem)`

→ `req` : durée de sommeil souhaitée

→ `rem` : durée de sommeil restante si interruption par un signal (et si $\neq$ `NULL`)

→ pour choix horloge et délai relatif ou absolu : `clock_nanosleep`

Endormissement

Système POSIX :

- 3 granularités de sommeil (*sleep*) : *sec*, μ*sec*, η*sec*

→ précision effective $\sim$ dizaines de msec. sur systèmes et matériels courants

→ pas de mécanisme *ad hoc* pour endormir un thread depuis un autre thread

Programmation :

- `unsigned int sleep(unsigned int sec)`

→ retournent temps restant si interruption par un signal

- `int usleep(useconds_t microsec)` (obsolete)
- `int nanosleep(const struct timespec * req, struct timespec * rem)`

→ `req` : durée de sommeil souhaitée

→ `rem` : durée de sommeil restante si interruption par un signal (et si $\neq$ `NULL`)

→ pour choix horloge et délai relatif ou absolu : `clock_nanosleep`

Endormissement

Système POSIX :

- 3 granularités de sommeil (*sleep*) : *sec*, μ*sec*, η*sec*

→ précision effective $\sim$ dizaines de msec. sur systèmes et matériels courants

→ pas de mécanisme *ad hoc* pour endormir un thread depuis un autre thread

Programmation :

- `unsigned int sleep(unsigned int sec)`

→ retournent temps restant si interruption par un signal

- `int usleep(useconds_t microsec)` (obsolete)
- `int nanosleep(const struct timespec * req, struct timespec * rem)`

→ `req` : durée de sommeil souhaitée

→ `rem` : durée de sommeil restante si interruption par un signal (et si $\neq$ `NULL`)

→ pour choix horloge et délai relatif ou absolu : `clock_nanosleep`

Endormissement

Système POSIX :

- 3 granularités de sommeil (*sleep*) : *sec*, μ*sec*, η*sec*

→ précision effective $\sim$ dizaines de msec. sur systèmes et matériels courants

→ pas de mécanisme *ad hoc* pour endormir un thread depuis un autre thread

Programmation :

- `unsigned int sleep(unsigned int sec)`

→ retournent temps restant si interruption par un signal

- `int usleep(useconds_t microsec)` (obsolete)
- `int nanosleep(const struct timespec * req, struct timespec * rem)`

→ `req` : durée de sommeil souhaitée

→ `rem` : durée de sommeil restante si interruption par un signal (et si $\neq$ `NULL`)

→ pour choix horloge et délai relatif ou absolu : `clock_nanosleep`

Endormissement

Système POSIX :

- 3 granularités de sommeil (*sleep*) : *sec*, μ*sec*, η*sec*

→ précision effective $\sim$ dizaines de msec. sur systèmes et matériels courants

→ pas de mécanisme *ad hoc* pour endormir un thread depuis un autre thread

Programmation :

- `unsigned int sleep(unsigned int sec)`

→ retournent temps restant si interruption par un signal

- `int usleep(useconds_t microsec)` (obsolete)
- `int nanosleep(const struct timespec * req, struct timespec * rem)`

→ `req` : durée de sommeil souhaitée

→ `rem` : durée de sommeil restante si interruption par un signal (et si $\neq$ `NULL`)

→ pour choix horloge et délai relatif ou absolu : `clock_nanosleep`

Alarme

Système POSIX :

- Mécanisme de déclenchement retardé, basé sur les signaux

→ cf. Signaux (Mécanismes multi-tâches → Communication)

Programmation :

- `unsigned int alarm(unsigned int seconds)`

→ `seconds` : temps d'attente de l'alarme avant déclenchement

→ annulation de toute alarme en attente

→ `seconds = 0` ⇒ simple annulation de l'alarme en attente

→ retourne le temps restant de l'alarme annulée (0 si aucune alarme en attente)

→ expiration de l'alarme ⇒ déclenchement du signal `SIGALRM`

→ cf. `man 2 alarm` et `man 3 alarm`

Alarme

Système POSIX :

- Mécanisme de déclenchement retardé, basé sur les signaux

→ cf. Signaux (Mécanismes multi-tâches → Communication)

Programmation :

- `unsigned int alarm(unsigned int seconds)`

→ `seconds` : temps d'attente de l'alarme avant déclenchement

→ annulation de toute alarme en attente

→ `seconds = 0` ⇒ simple annulation de l'alarme en attente

→ retourne le temps restant de l'alarme annulée (0 si aucune alarme en attente)

→ expiration de l'alarme ⇒ déclenchement du signal `SIGALRM`

→ cf. `man 2 alarm` et `man 3 alarm`

Minuteurs

Système POSIX :

- **timer** : *timer object*

→ recevoir régulièrement un signal

→ à réception, exécution d'une routine (*handler*)

→ relancement automatiquement possible

Rq. précision effective sur système non temps réel $\sim$ millisecondes

Programmation :

- 2 types de *timers* :

→ `struct itimerval`, `setitimer`, `getitiemer` (obsolète)

→ type `timer_t` (*timer* temps réel, option gcc : `-lrt`)

Minuteurs

Système POSIX :

- **timer** : *timer object*

→ recevoir régulièrement un signal

→ à réception, exécution d'une routine (*handler*)

→ relancement automatiquement possible

Rq. précision effective sur système non temps réel $\sim$ millisecondes

Programmation :

- 2 types de *timers* :

→ **`struct itimerval`**, `setitimer`, `getitiemer` (obsolète)

→ type **`timer_t`** (*timer* temps réel, option gcc : `-lrt`)

Minuteurs

Création d'un *timer* avec **timer_t** :

- `int timer_create(`
  `clockid_t clockid,`
  `struct sigevent * sevp,`
  `timer_t * timer)`

→ `clockid` : type d'horloge utilisée (`CLOCK_REALTIME`, `CLOCK_MONOTONIC`, etc.)

→ `sevp` : manière de notifier le processus à expiration

→ `timer` : pointeur sur le *timer* à créer

- `struct sigevent` (`man 7 sigevent`) :

→ `sigev_notify` : methode de notification

→ `SIGEV_SIGNAL` : par signal ⇒ numéro du signal : `sigev_signo`

→ `SIGEV_THREAD` : par lancement d'un thread ⇒

point d'entrée : `sigev_notify_function`

attributs : `sigev_notify_attributes`

→ `sigev_value` : données passées avec notification (`sigev_value.sival_ptr = NULL`)

Minuteurs

Création d'un *timer* avec **timer_t** :

- `int timer_create(`
  `clockid_t clockid,`
  `struct sigevent * sevp,`
  `timer_t * timer)`

→ `clockid` : type d'horloge utilisée (`CLOCK_REALTIME`, `CLOCK_MONOTONIC`, etc.)

→ `sevp` : manière de notifier le processus à expiration

→ `timer` : pointeur sur le *timer* à créer

- `struct sigevent` (`man 7 sigevent`) :

→ `sigev_notify` : methode de notification

→ `SIGEV_SIGNAL` : par signal ⇒ numéro du signal : `sigev_signo`

→ `SIGEV_THREAD` : par lancement d'un thread ⇒

point d'entrée : `sigev_notify_function`

attributs : `sigev_notify_attributes`

→ `sigev_value` : données passées avec notification (`sigev_value.sival_ptr = NULL`)

Minuteurs

Amorçage d'un *timer* de type **timer_t**

- `int timer_settime(`
  `timer_t timer,`
  `int flags,`
  `const struct itimerspec * conf,`
  `struct itimerspec * old)`

→ `timer` : *timer* à amorcer

→ `flags` : `0` ⇒ `conf` relative h. actuelle, `TIMER_ABSTIME` ⇒ `conf` absolue

→ `conf` : configuration du *timer*

→ `old` : ancienne configuration (pour sauvegarde, si ≠ `NULL`)

- `struct itimerspec` :

→ `it_interval` : période de répétition (type `struct timespec`, cf. Horodatage)

→ `it_value` : délai avant expiration (type `struct timespec`, cf. Horodatage)

Minuteurs

Amorçage d'un *timer* de type **timer_t**

- `int timer_settime(`
  `timer_t timer,`
  `int flags,`
  `const struct itimerspec * conf,`
  `struct itimerspec * old)`

→ `timer` : *timer* à amorcer

→ `flags` : `0` ⇒ `conf` relative h. actuelle, `TIMER_ABSTIME` ⇒ `conf` absolue

→ `conf` : configuration du *timer*

→ `old` : ancienne configuration (pour sauvegarde, si ≠ `NULL`)

- `struct itimerspec` :

→ `it_interval` : période de répétition (type `struct timespec`, cf. Horodatage)

→ `it_value` : délai avant expiration (type `struct timespec`, cf. Horodatage)

Minuteurs

Destruction d'un *timer* de type **timer_t**

- `int timer_delete(timer_t timer)`

→ `timer` : *timer* à détruire

Plan

Références


Développement Système sous Linux,
édition Eyrolles, ISBN 978-2212142075,
Christophe BLAESS


Programming with Posix Threads,
édition Addison-Wesley, ISBN 978-0201633924,
David BUTENHOF


Posix Programmer's Guide,
édition O'Reilly, ISBN 978-0937175736,
Donald LEWINE


The Art of Unix Programming,
édition Addison-Wesley, ISBN 978-0131429017,
Eric S. RAYMOND


Advanced Programming in the Unix Environment,
édition Addison-Wesley, ISBN 978-0321637734,
W. Richard STEVENS

Références

- *Posix Specifications*,
  `https://pubs.opengroup.org/onlinepubs/9699919799/`
- *Posix FAQ*,
  `http://www.opengroup.org/austin/papers/posix_faq.html`
- *The Open Group*,
  `https://www.opengroup.org`
- *The Open Group Publications*,
  `https://publications.opengroup.org`
- *IEEE Standards*,
  `https://standards.ieee.org`