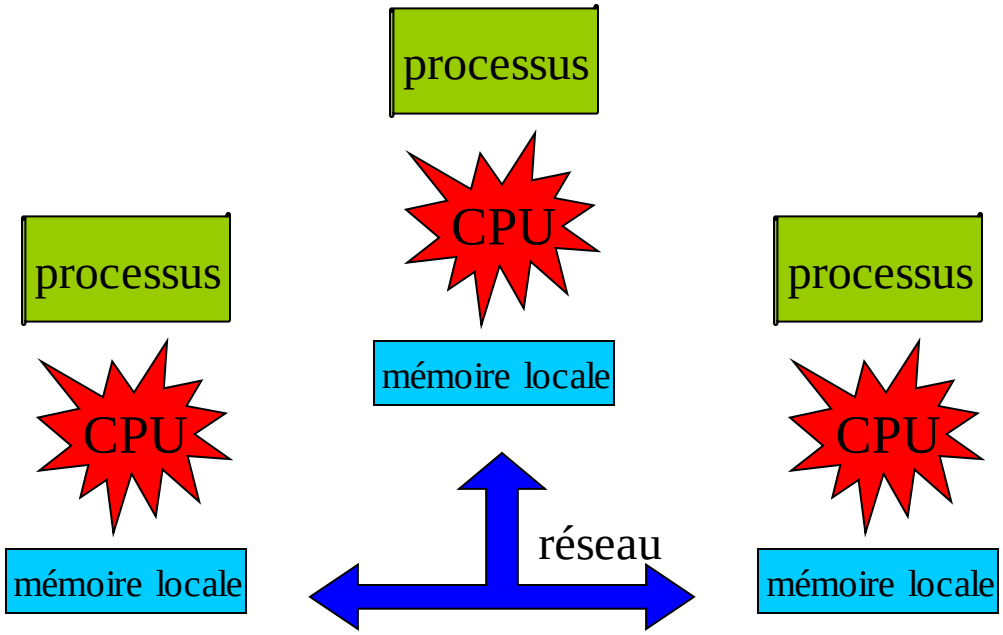
PROGRAMMATION A BASE DE THREADS

Marc Pérache marc.perache@cea.fr

Arthur Loussert

Rappel : Système à mémoire distribuée

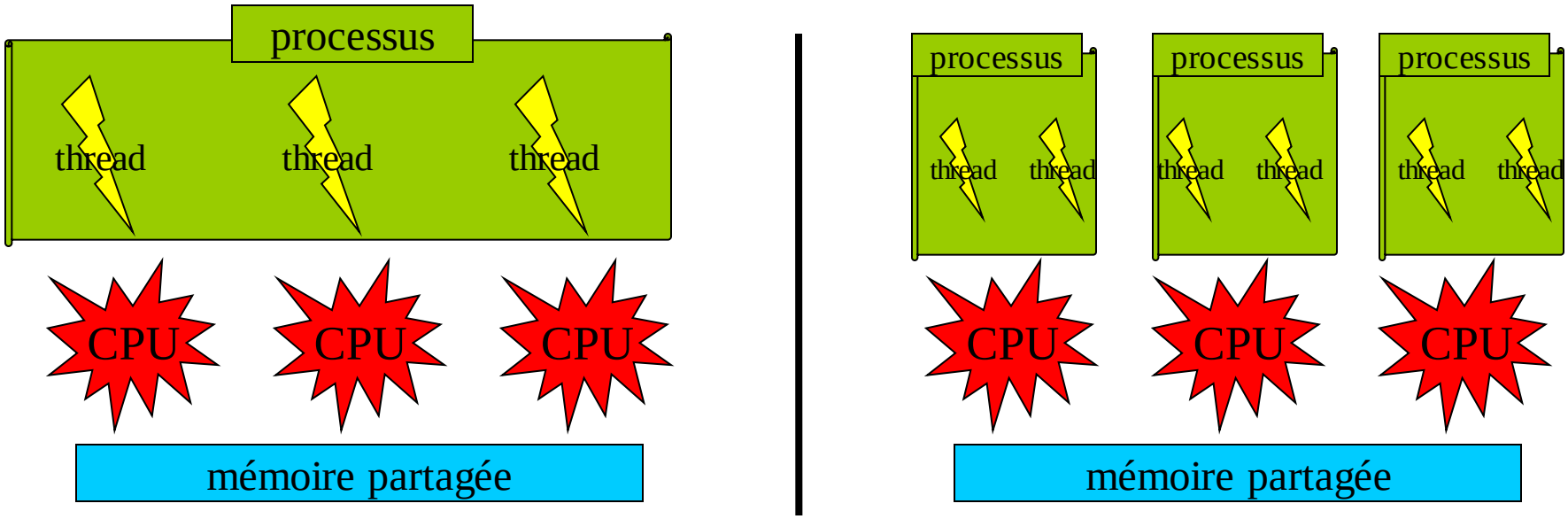
- Système à mémoire distribuée
  - Ressources de calcul qui n'ont pas de mémoire partagée, que ce soit de manière physique ou logicielle



Rappel : Système à mémoire partagée

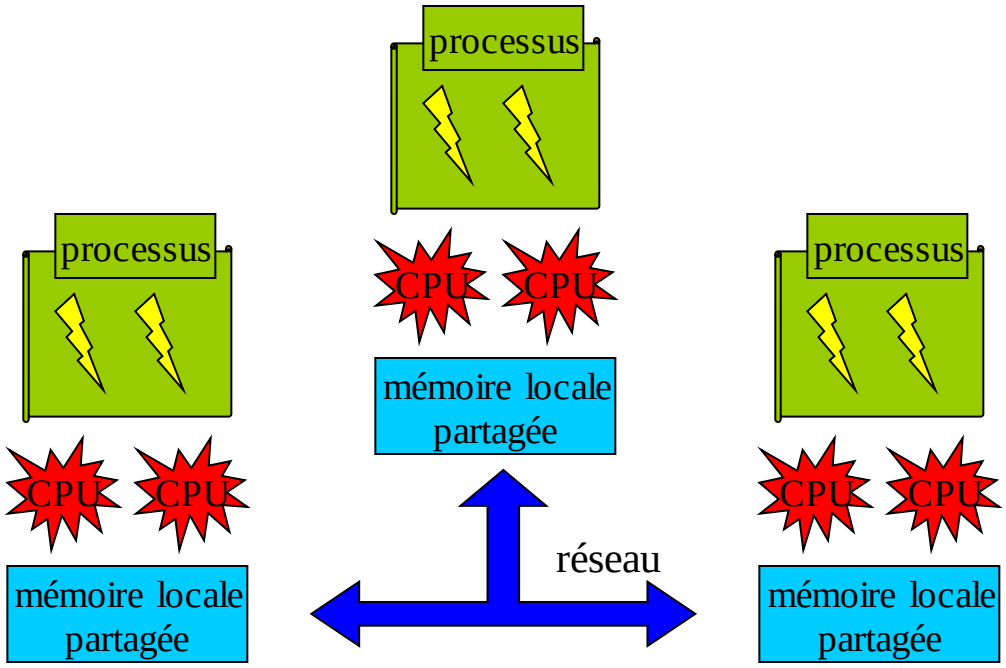
- Système à mémoire partagée
  - Système mettant en jeu plusieurs ressources de calcul qui ont de la mémoire partagée (physiquement ou de manière logicielle)

- Nœud
  - Plus grand ensemble de processeurs partageant matériellement de la mémoire. On parle de nœud SMP ou de nœud NUMA



Rappel : Système hybrides

- Hybrides : mémoire partagée/distribuée
- Grappe ou *Cluster*
  - Ensemble de nœuds interconnectés par un réseau



Programmation mémoire partagée

- Programmation parallèle mémoire partagée
  - Exploitation des architectures mémoire partagée
  - Utilisation de la programmation par threads
- Avantages
  - Permet de profiter de communication instantanées.
  - Profite de l'aspect mémoire partagée des noeuds de calcul.
  - Permet de faire du recouvrement.
- Profil des architectures adaptées
  - Processeur multicoeurs.
  - Nœuds de calcul SMP et NUMA.

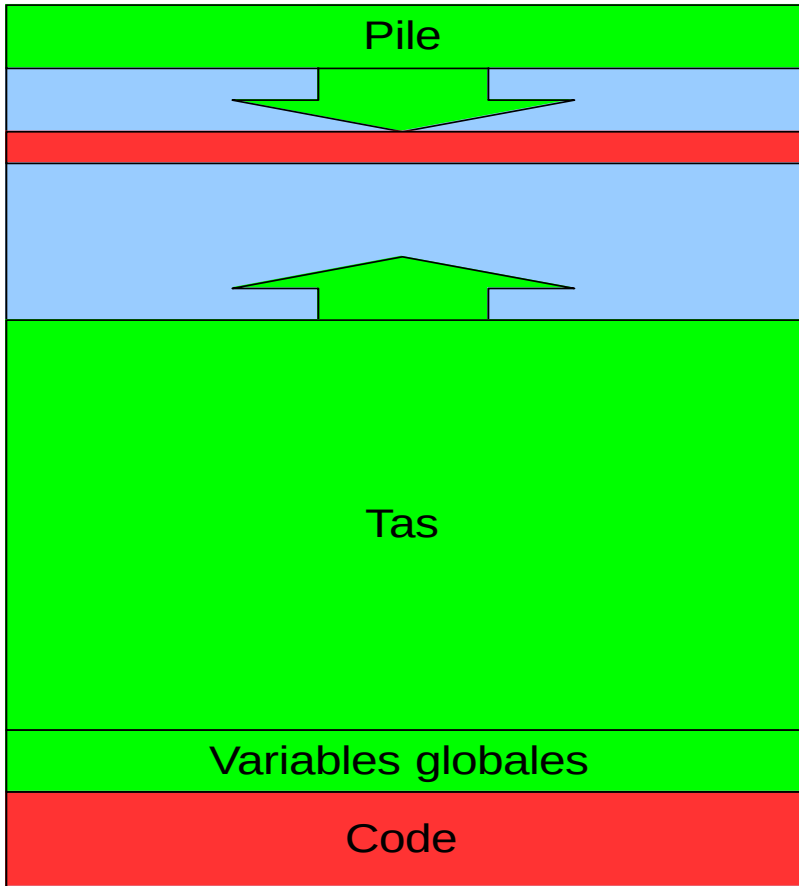
Définition d’un thread

- Thread = processus léger

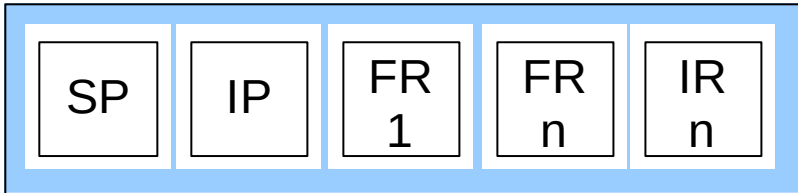
- Éléments d'un thread
  - Une pile
  - Un contexte : ensembles de registres

- Éléments d'un processus multithread
  - Une table de pages
  - Un ensemble de threads

Processus

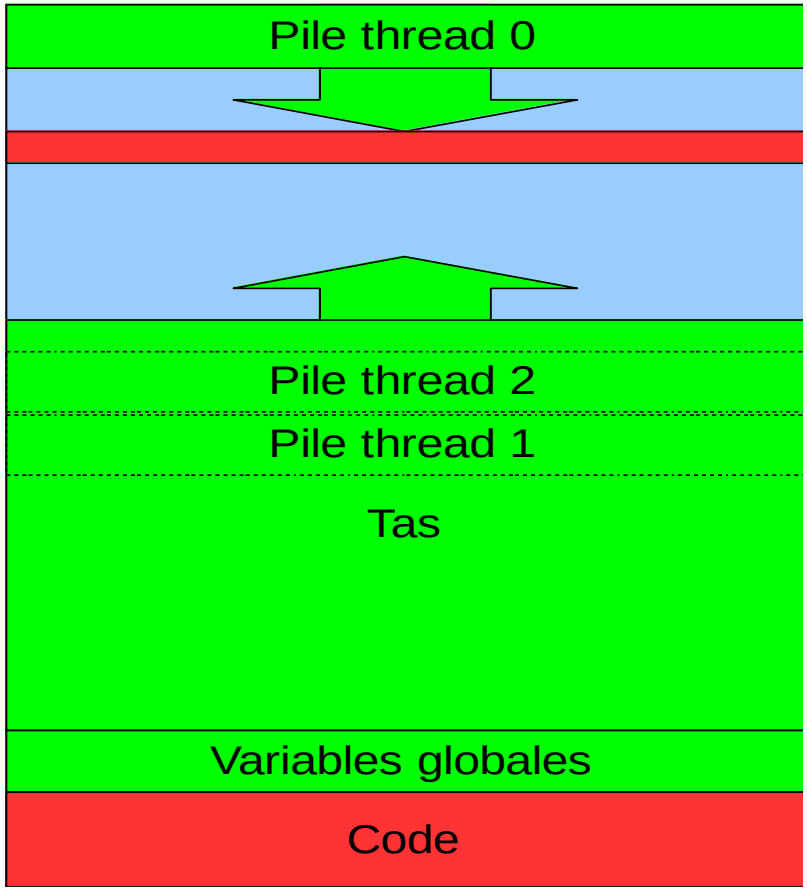


Mémoire

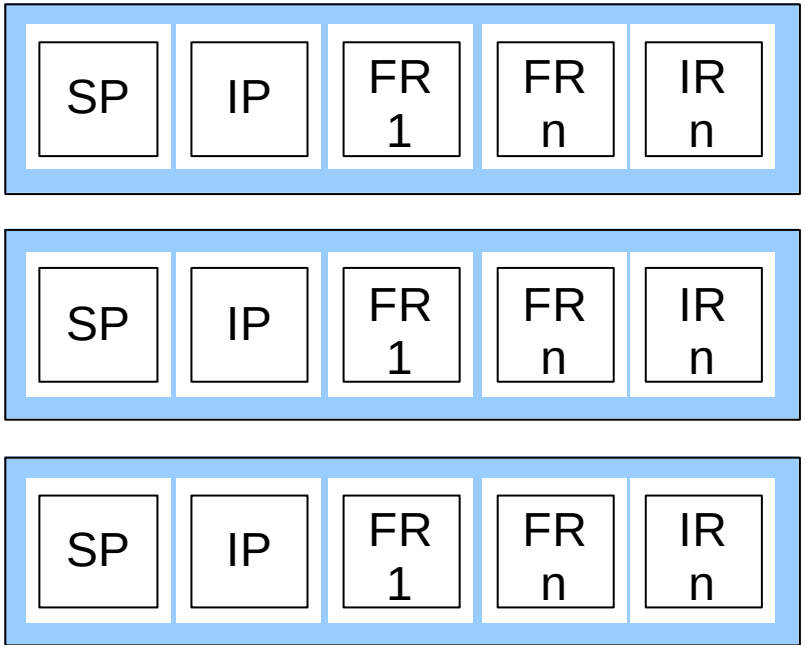


Structures

Processus multithread



Mémoire



Structures

Caractéristiques

- Modèle
  - Type *fork/join*
- Espace d'adressage
  - Les threads partagent tous le même espace d'adressage
  - Les variables globales sont toutes partagées
- Pile
  - La pile des threads (hormis le 0) ne grossit pas
  - La pile d'un thread est localisée dans le tas
- Mode de communication
  - Les threads communiquent directement par la mémoire

Quelques définitions

- **Définition 1 : SMP** – Un système SMP est constitué de plusieurs processeurs identiques connecté à une unique mémoire physique.
- **Définition 2 : Non Uniform Memory Access** – Un système NUMA est constitué de plusieurs processeurs connecté à plusieurs mémoires distinctes reliées entre-elles par des mécanisme matériels. Le temps d'accès à des données en mémoire locale au processeur est donc réduit par rapport au temps d'accès à des données présente dans une mémoire distante.
- **Définition 3 : Processus** – Un processus est une "coquille" dans laquelle le système exécute chaque programme.
- **Définition 4 : Thread** – Un thread est une suite logique d'actions résultat de l'exécution d'un programme.

Quelques définitions

- **Définition 5 : Préemption** – La préemption est le fait de passer régulièrement la main d'un flot d'exécution à l'autre sans indication particulière dans les flots eux-mêmes.
- **Définition 6 : Section critique** – Région du programme où l'on souhaite limiter (généralement à un seul) le nombre de flots d'exécution. Ces régions correspondent généralement à des zones où des invariants sur des données peuvent ne pas être respectés.
- **Définition 7 : Attente active** – Méthode de synchronisation où l'entité, attendant de passer cette synchronisation, garde le contrôle du processeur.

Quelques définitions

- **Définition 8 : Réentrance** – Fait, pour une fonction, de pouvoir être exécutée pendant son exécution. Une fonction sans effet de bord et travaillant uniquement avec des variables locales est de fait automatiquement réentrante.
- **Définition 9 : Mutex** – Un mutex permet l'exclusion mutuelle et la synchronisation entre threads.
- **Définition 10 : Sémaphore** – Les sémaphores sont purement et simplement des compteurs pour des ressources partagées par plusieurs threads.
- **Définition 11 : Condition** – Les conditions variables (condvar) permettent de réveiller un thread endormi en fonction de la valeur d'une variable.

API POSIX

- Interface de programmation pour la création et gestion des threads
  - Modèle fork/join
- Création des threads

```
pthread_create( pthread_t * thread,
      pthread_attr_t * attribut,
      void *(*routine) (void *),
      void * argument )
```

  - Le nouveau flot d'exécution démarre en se branchant à la routine spécifiée.
  - Cette routine reçoit l'argument prévu.
- Attente de la terminaison d'un thread

```
pthread_join( pthread_t thread, void ** resultat )
```

- Fin du thread courant

```
pthread_exit( void * resultat ).
```

API POSIX (suite)

- Envoi d'un signal à un thread

  `pthread_kill( pthread_t thread, int nu_du_signal )`

  - Moyen dur pour tuer un thread. Il existe des méthodes plus conviviales.

- Abandonner le CPU pour le donner à un autre thread/processus

  `sched_yield()` ou `pthread_yield()`

- Identifiant d'un thread

  `pthread_self()`

Premier code multithread

```c
#include <pthread.h>

int NB_THREADS;
void* run(void* arg){
    long rank;
    rank = (long)arg;
    printf("Hello, I'am %ld (%p)\n",rank,pthread_self());
    return arg;
}
int main(int argc, char** argv){
    pthread_t* pids;
    long i;
    NB_THREADS=atoi(argv[1]);
    pids = (pthread_t*)malloc(NB_THREADS*sizeof(pthread_t));
    for(i = 0; i < NB_THREADS; i++){
        pthread_create(&(pids[i]),NULL,run,(void*)i);
    }
    for(i = 0; i < NB_THREADS; i++){
        void* res;
        pthread_join(pids[i],&res);
        fprintf(stderr,"Thread %ld Joined\n",(long)res);
        assert(res == (void*)i);
    }
return 0;
}
```

```
$ gcc -o test test.c -pthread

$ ./test 4
Hello, I'am 0 (0xb785db70)
Thread 0 Joined
Hello, I'am 1 (0xb705cb70)
Thread 1 Joined
Hello, I'am 2 (0xb685bb70)
Thread 2 Joined
Hello, I'am 3 (0xb605ab70)
Thread 3 Joined

$ ./test 4
Hello, I'am 2 (0xb6860b70)
Hello, I'am 3 (0xb605fb70)
Hello, I'am 1 (0xb7061b70)
Hello, I'am 0 (0xb7862b70)
Thread 0 Joined
Thread 1 Joined
Thread 2 Joined
Thread 3 Joined
```

Premier code multithread

```
#include <pthread.h>

int NB_THREADS;
void* run(void* arg){
  long rank;
  rank = (long)arg;
  printf("Address of rank (%p) and NB_THREADS (%p)\n",
    &rank,&NB_THREADS );
  return arg;
}
int main(int argc, char** argv){
  pthread_t* pids;
  long i;
  NB_THREADS=atoi(argv[1]);
  pids = (pthread_t*)malloc(NB_THREADS*sizeof(pthread_t));
  for(i = 0; i < NB_THREADS; i++){
      pthread_create(&(pids[i]),NULL,run,(void*)i);
  }
  for(i = 0; i < NB_THREADS; i++){
      void* res;
      pthread_join(pids[i],&res);
      assert(res == (void*)i);
  }
return 0;
}
```

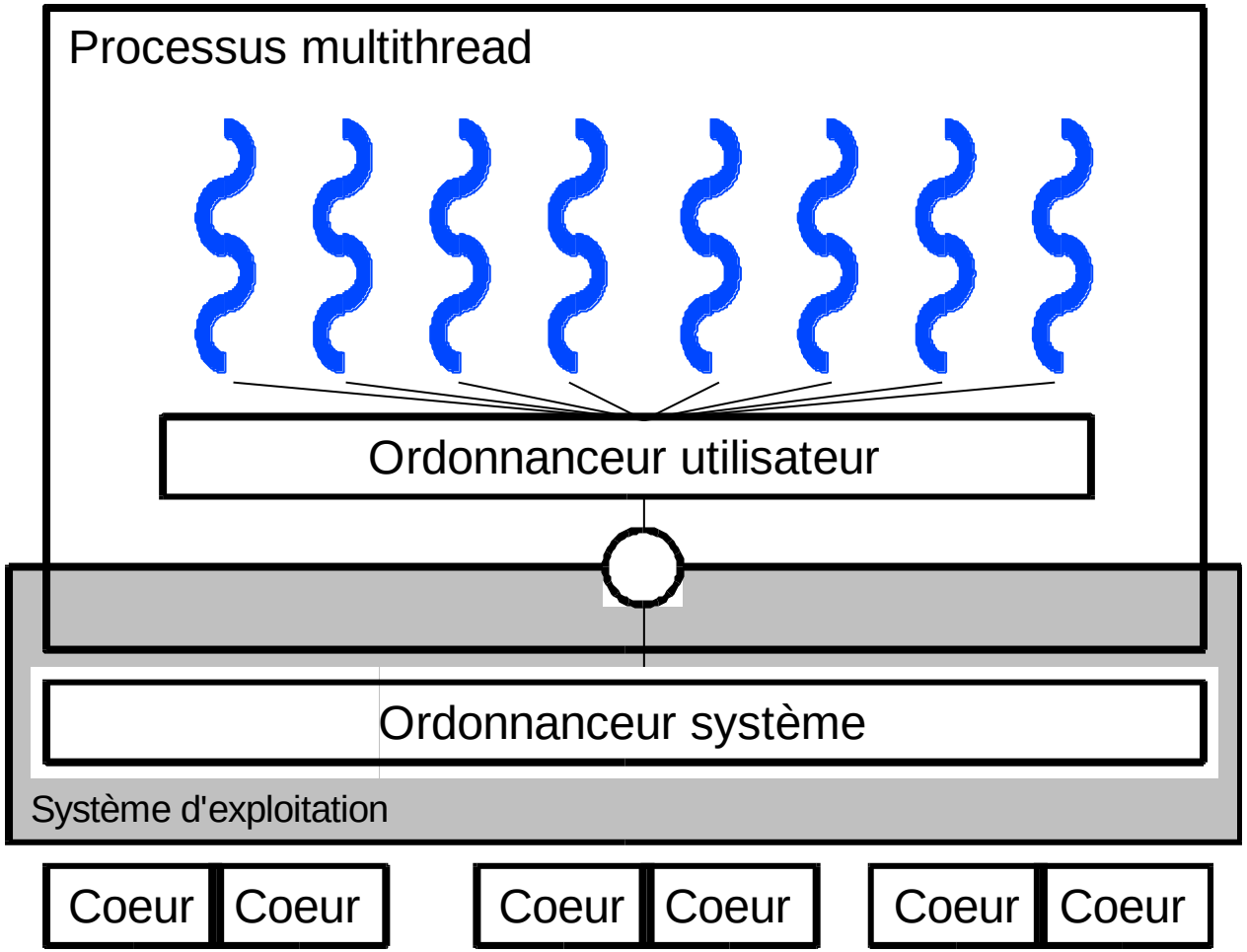
```
$ gcc -o test2 test2.c -pthread

$ ./test2 4
Address of rank (0xb788e38c) and
NB_THREADS (0x804a06c)
Address of rank (0xb708d38c) and
NB_THREADS (0x804a06c)
Address of rank (0xb688c38c) and
NB_THREADS (0x804a06c)
Address of rank (0xb608b38c) and
NB_THREADS (0x804a06c)
```

MODÈLE D'EXÉCUTION

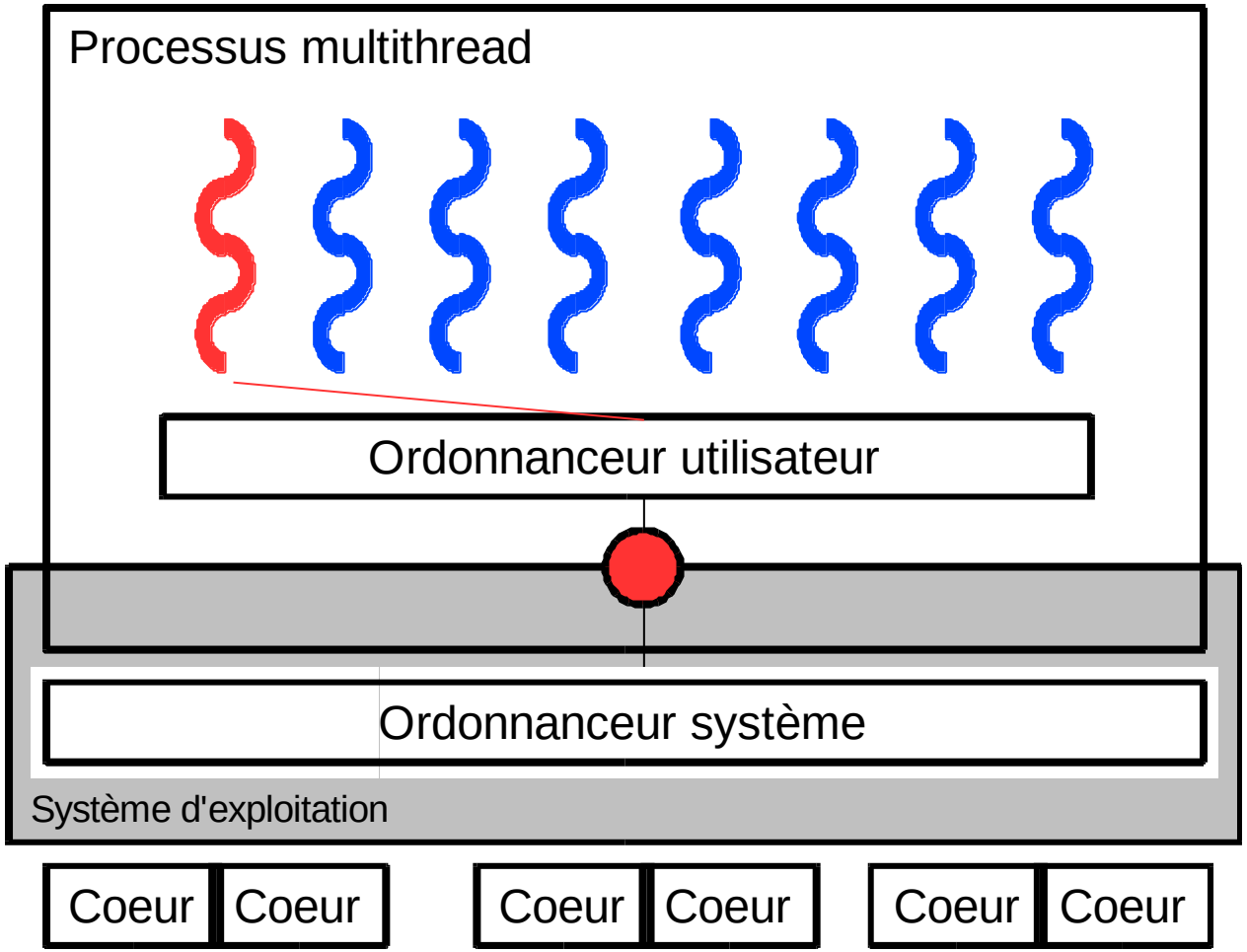
Bibliothèques de threads

- Gestion des threads
  - Dans une bibliothèque extérieure
  - Par exemple : `libpthread`
- Mode de gestion des threads
  - Bibliothèque utilisateur
  - Bibliothèque système
  - Bibliothèque mixte (ou MxN)

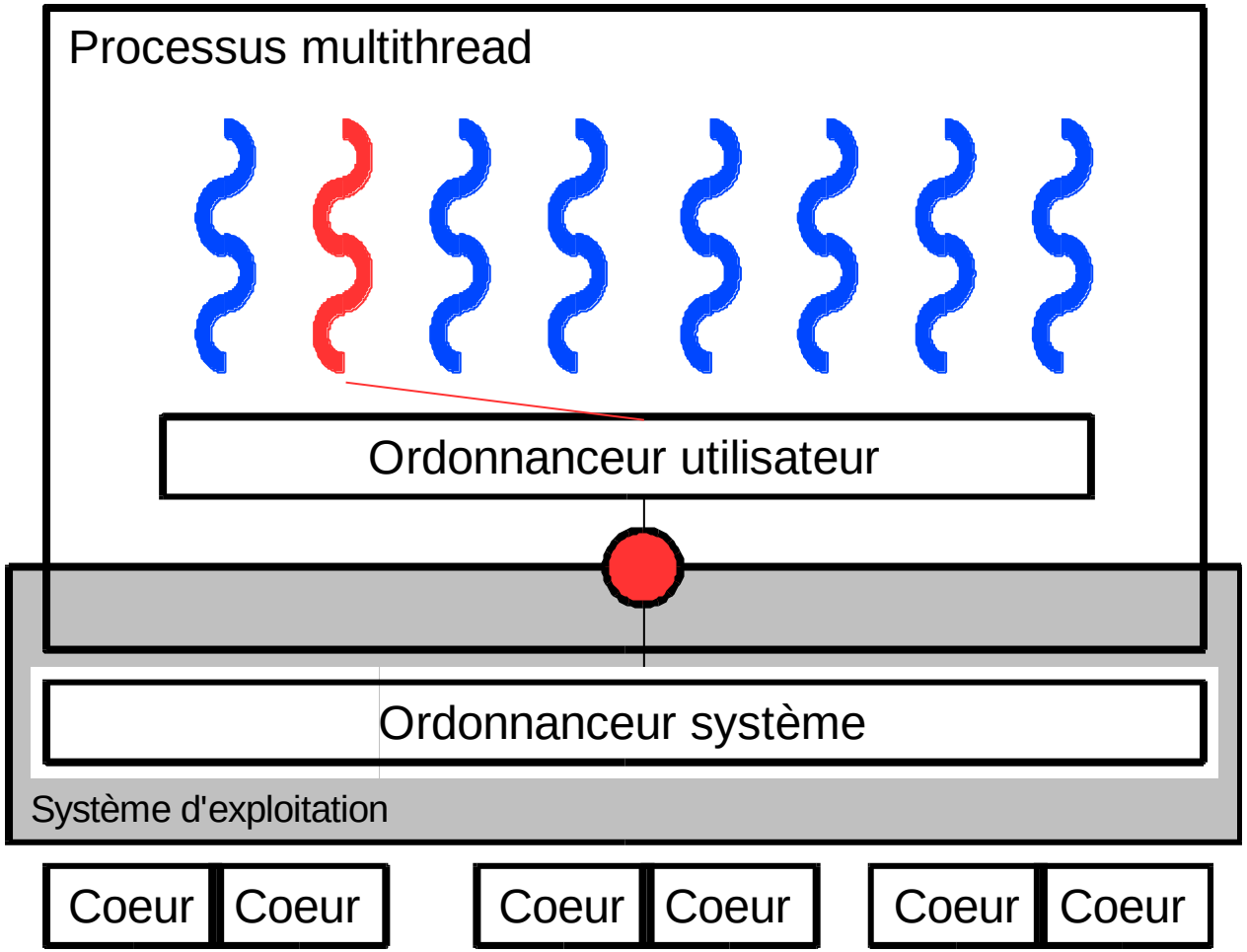
Bibliothèque utilisateur



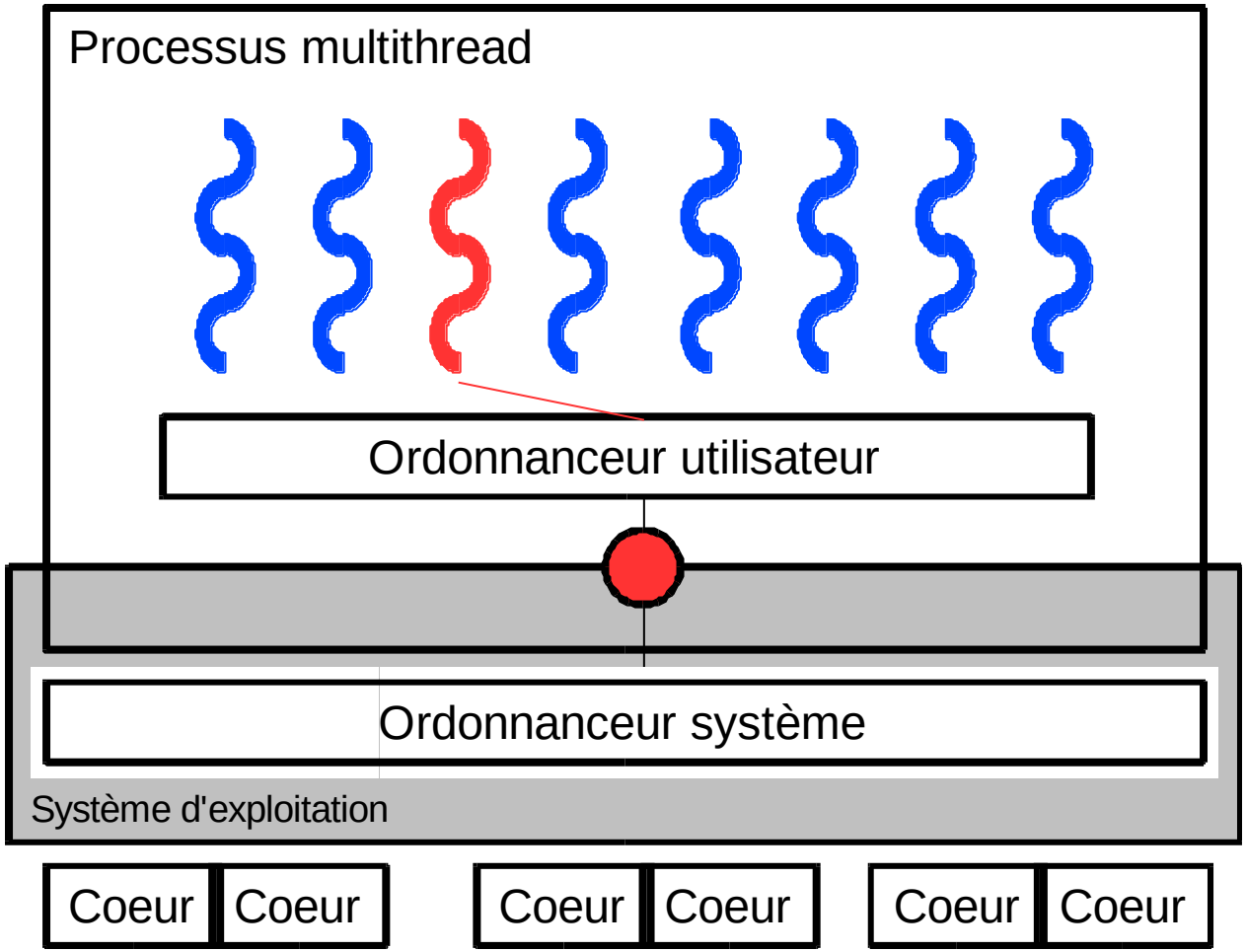
Bibliothèque utilisateur



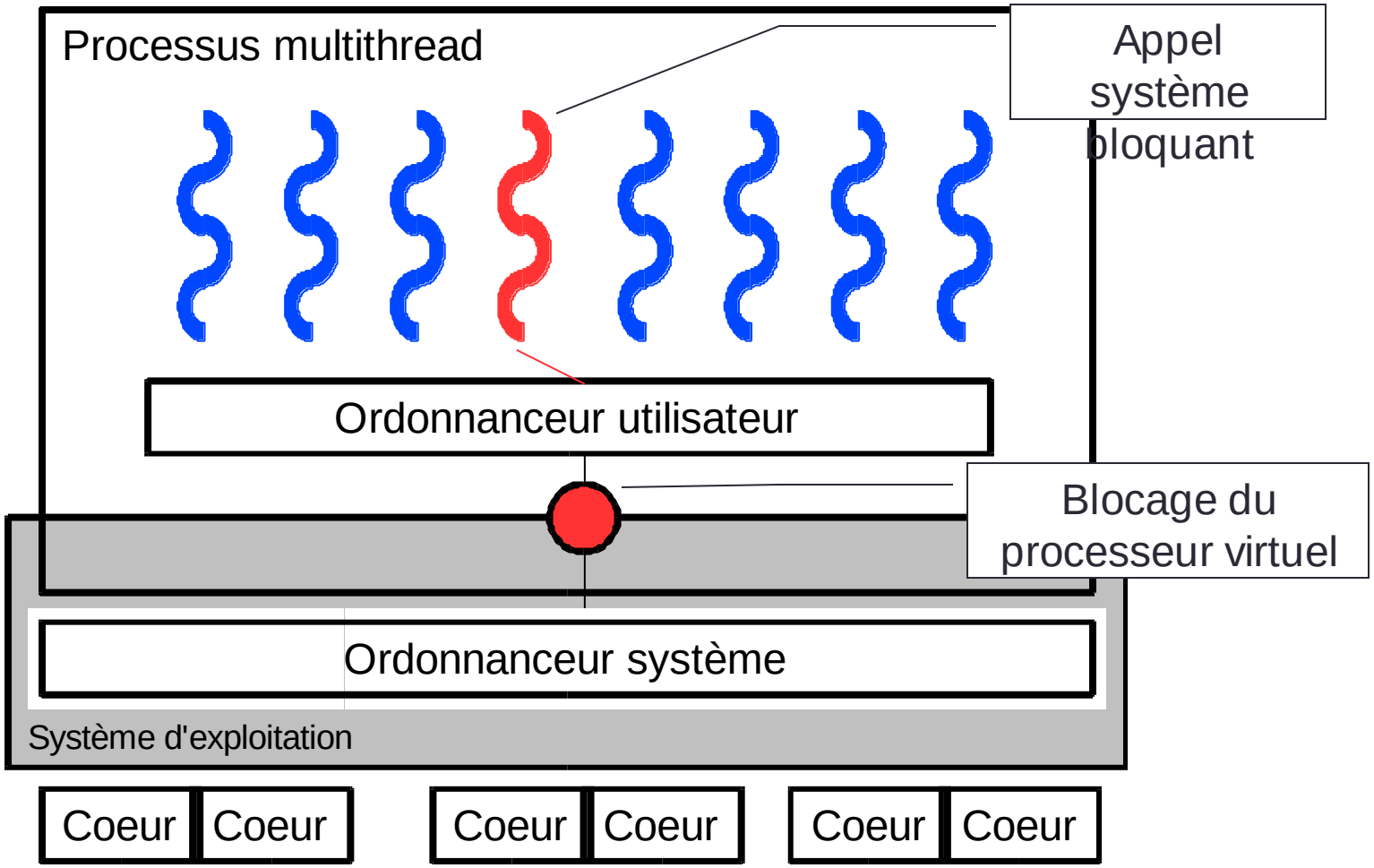
Bibliothèque utilisateur



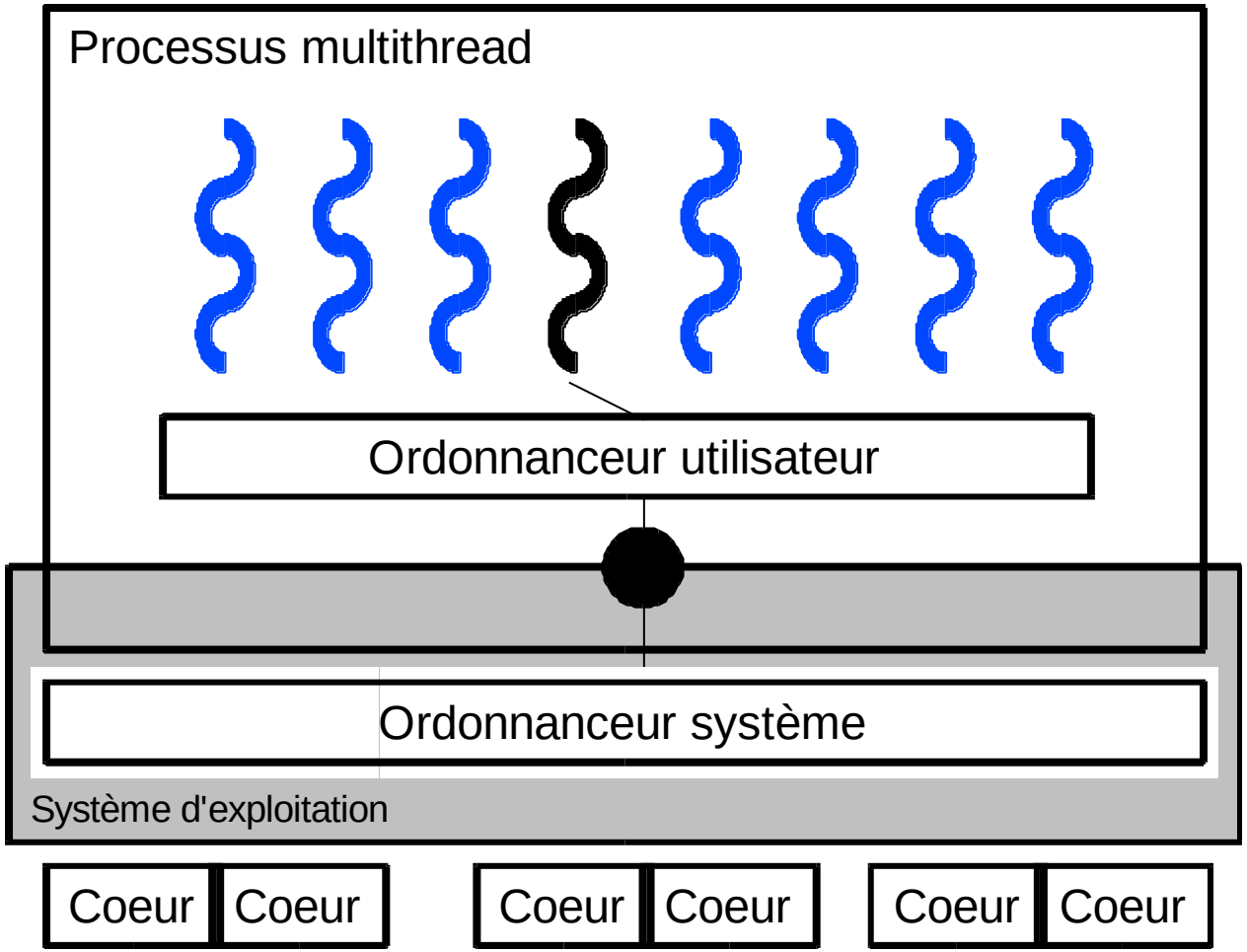
Bibliothèque utilisateur



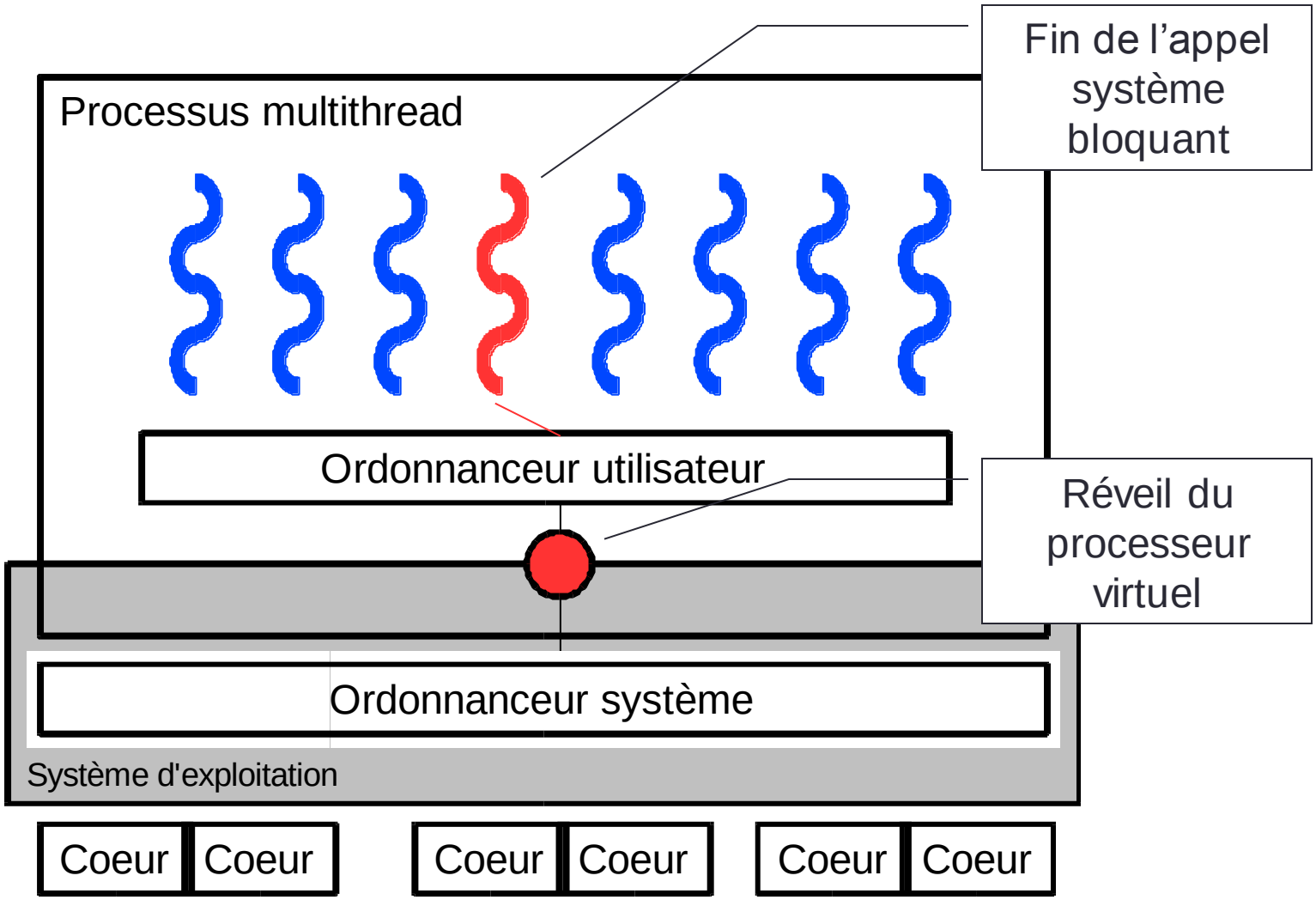
Bibliothèque utilisateur



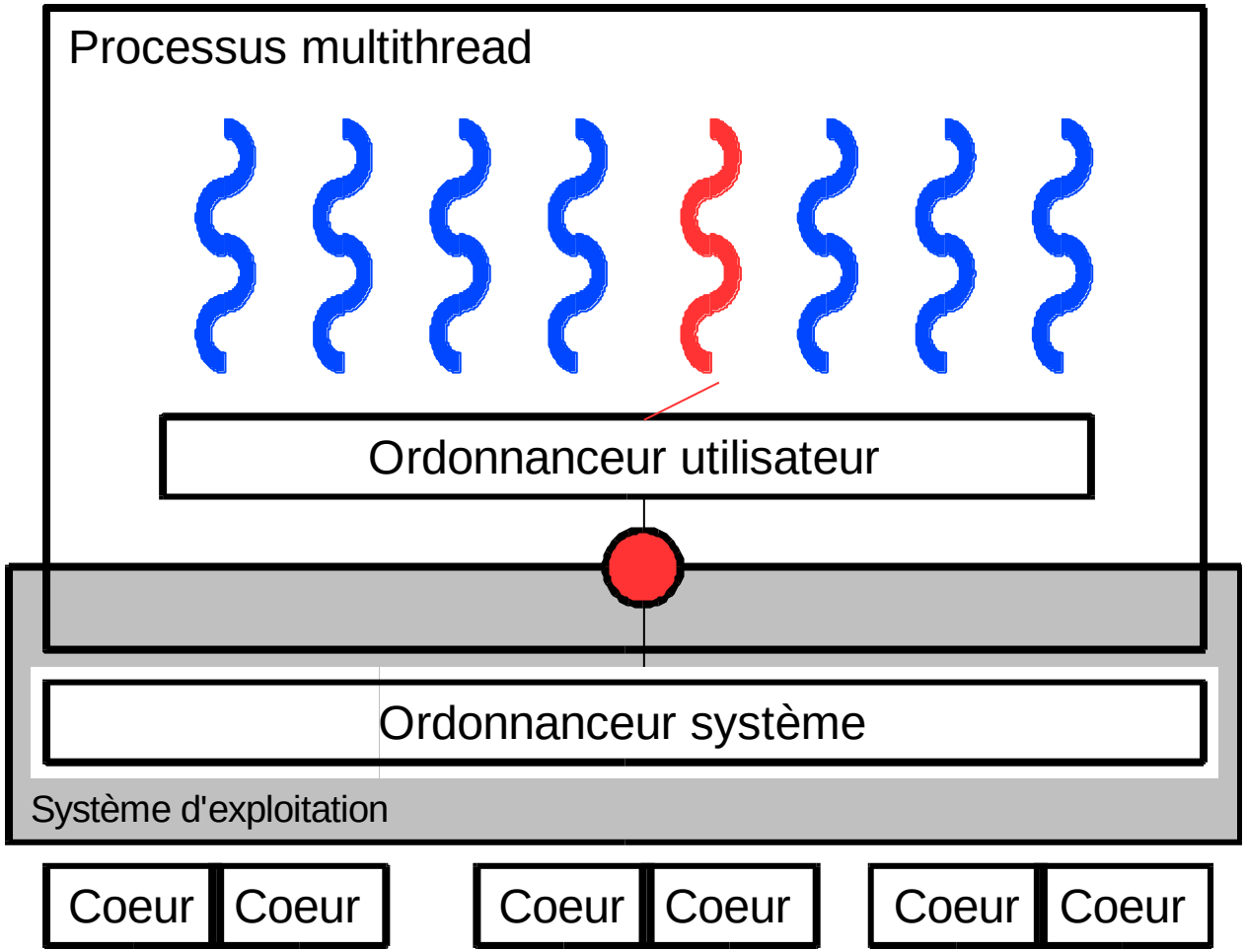
Bibliothèque utilisateur



Bibliothèque utilisateur



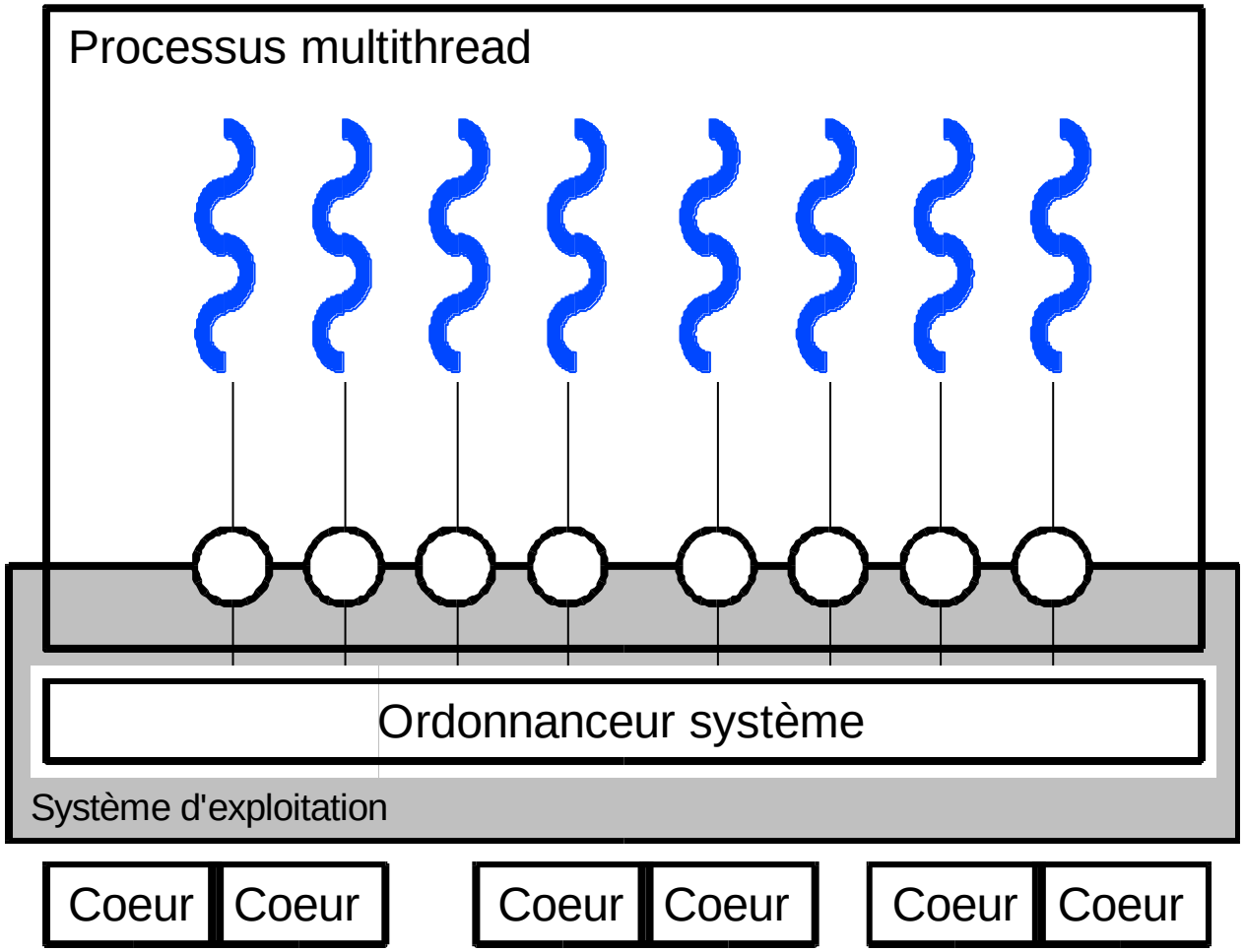
Bibliothèque utilisateur



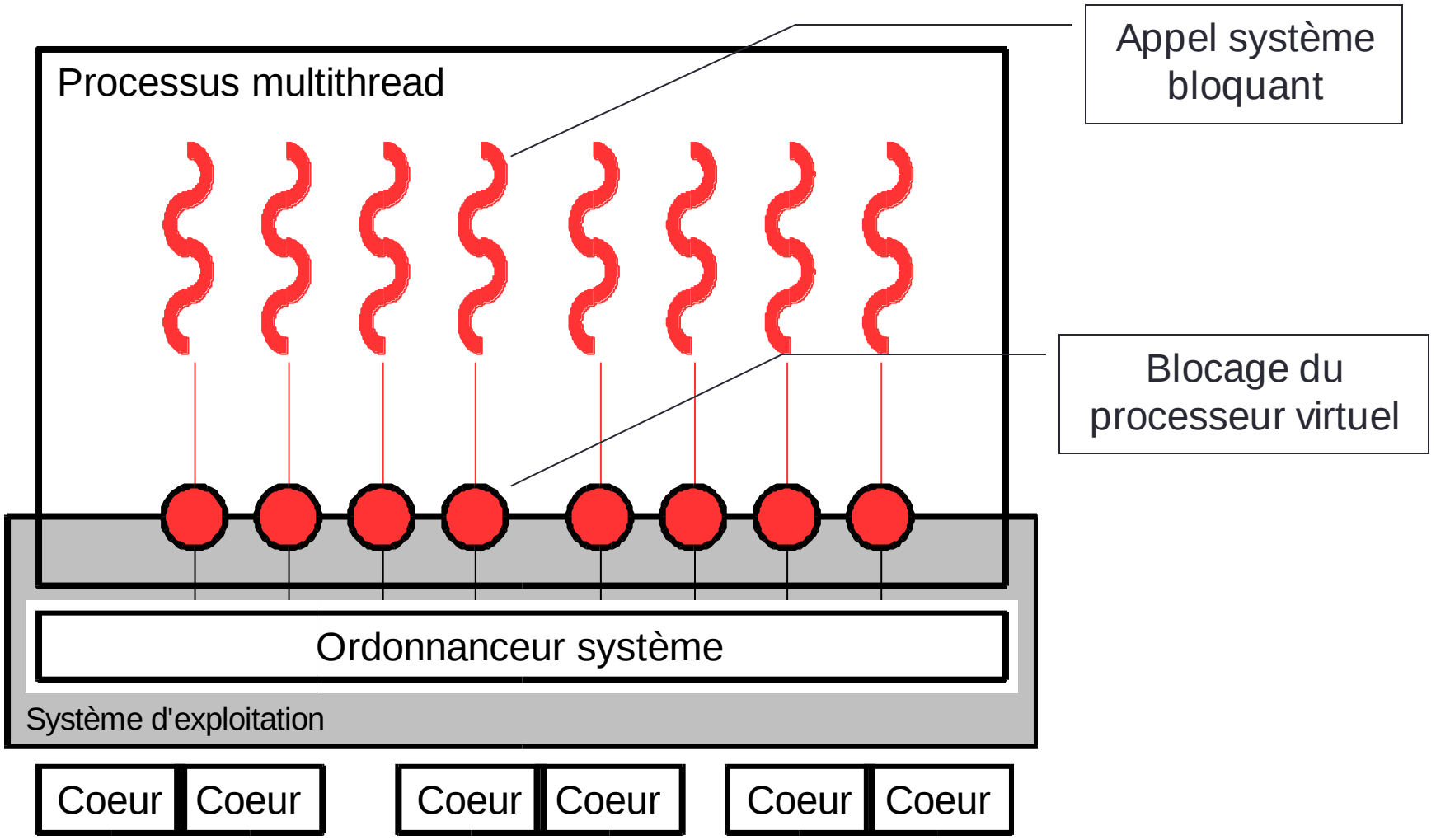
Bibliothèque utilisateur

- Première bibliothèque de threads.
  - 1 thread noyau par processus
- Avantages
  - Simple à implémenter.
  - Performante.
  - Entièrement en espace utilisateur.
- Inconvénients
  - Pas adaptée au SMP et multicoeur.
  - Problèmes avec les appels systèmes.
- Exemple : GNUPth

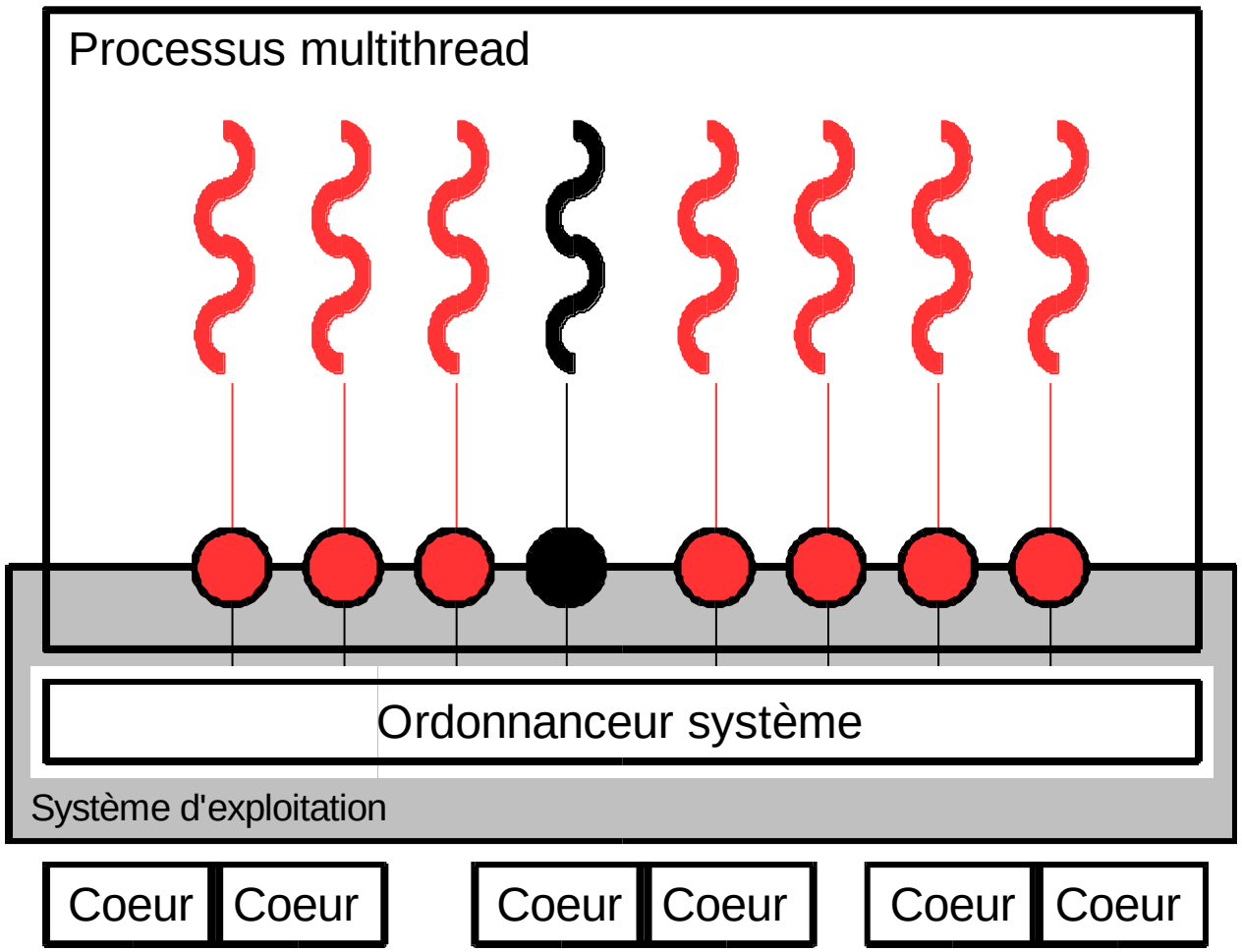
Bibliothèque système



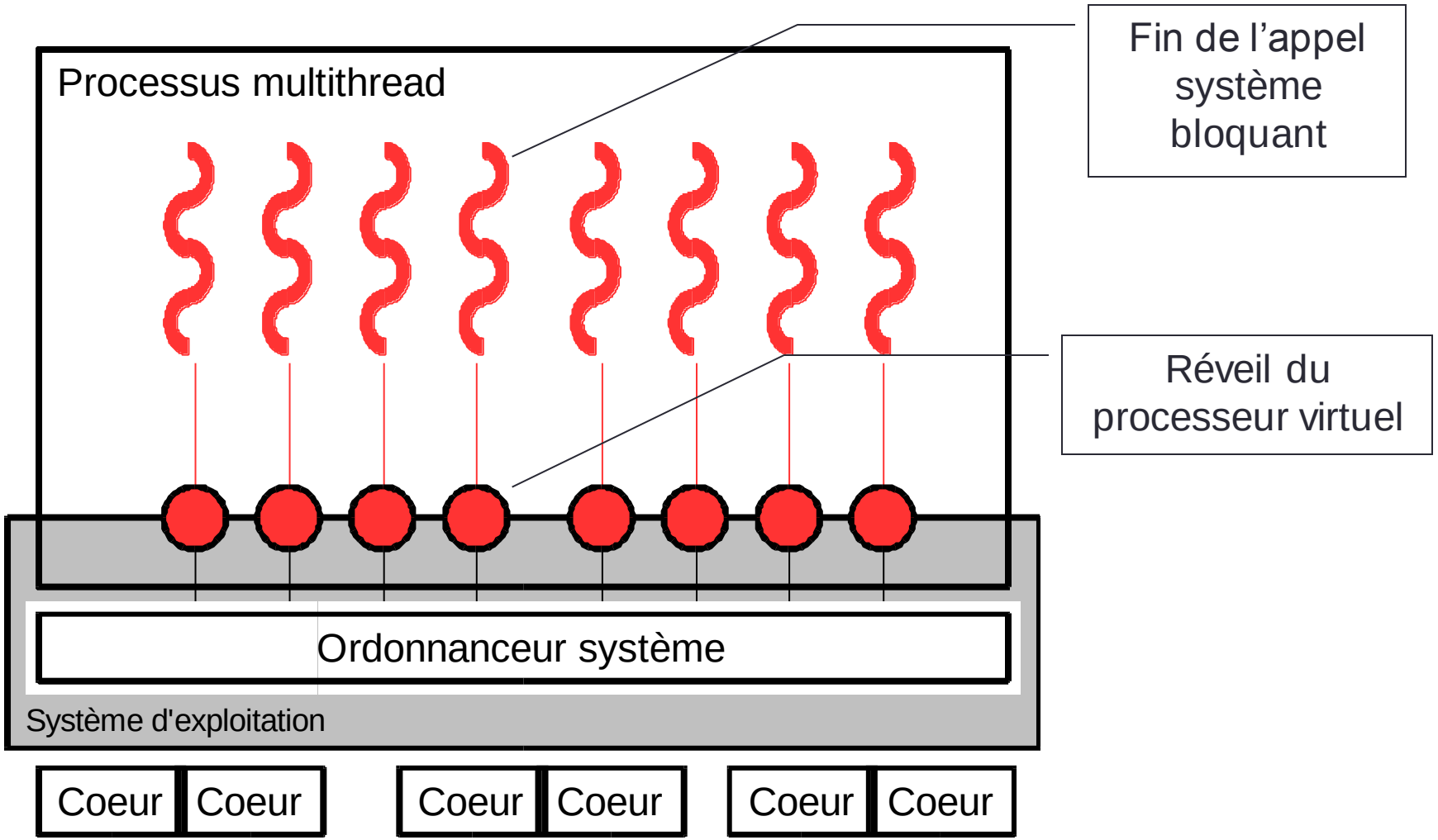
Bibliothèque système



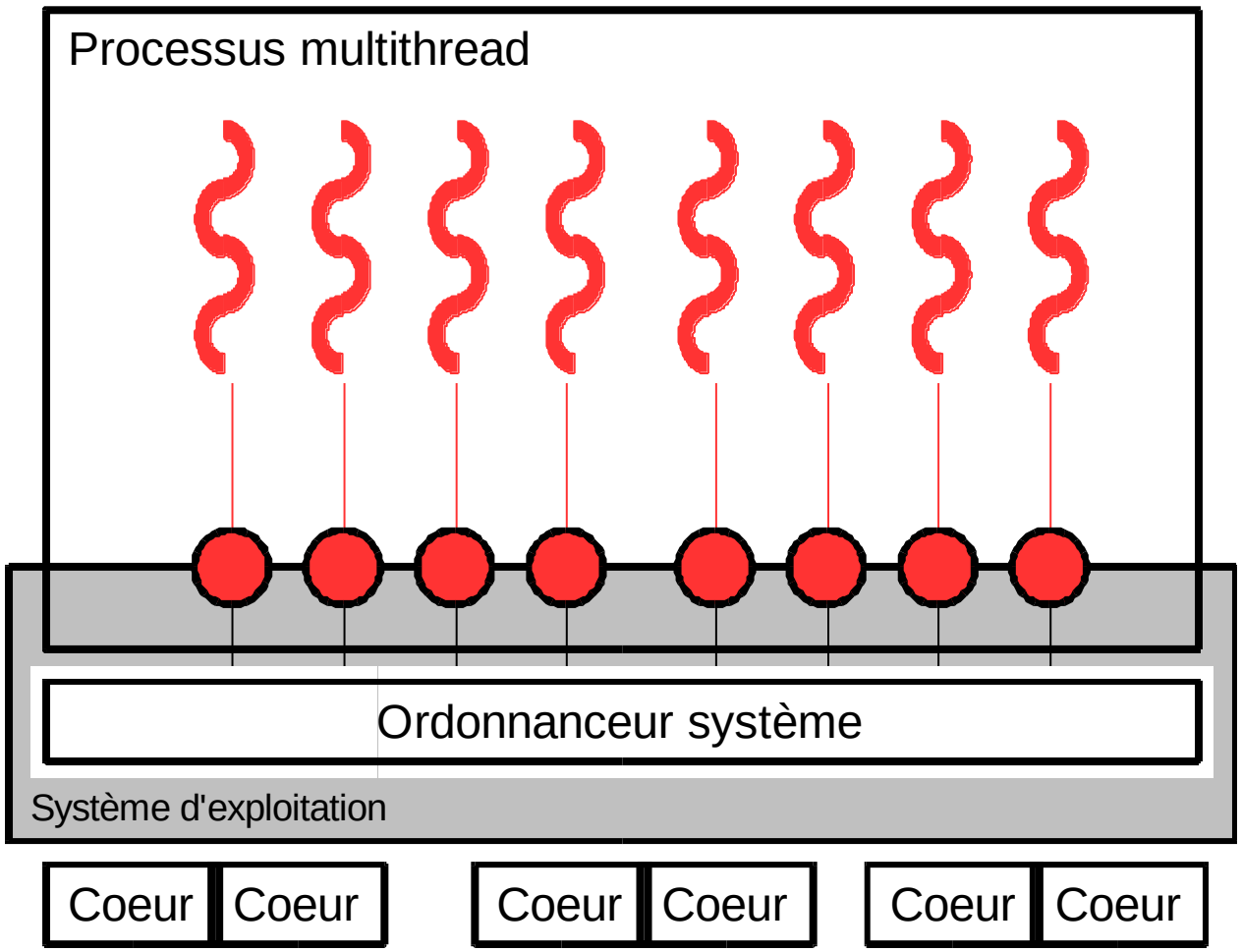
Bibliothèque système



Bibliothèque système



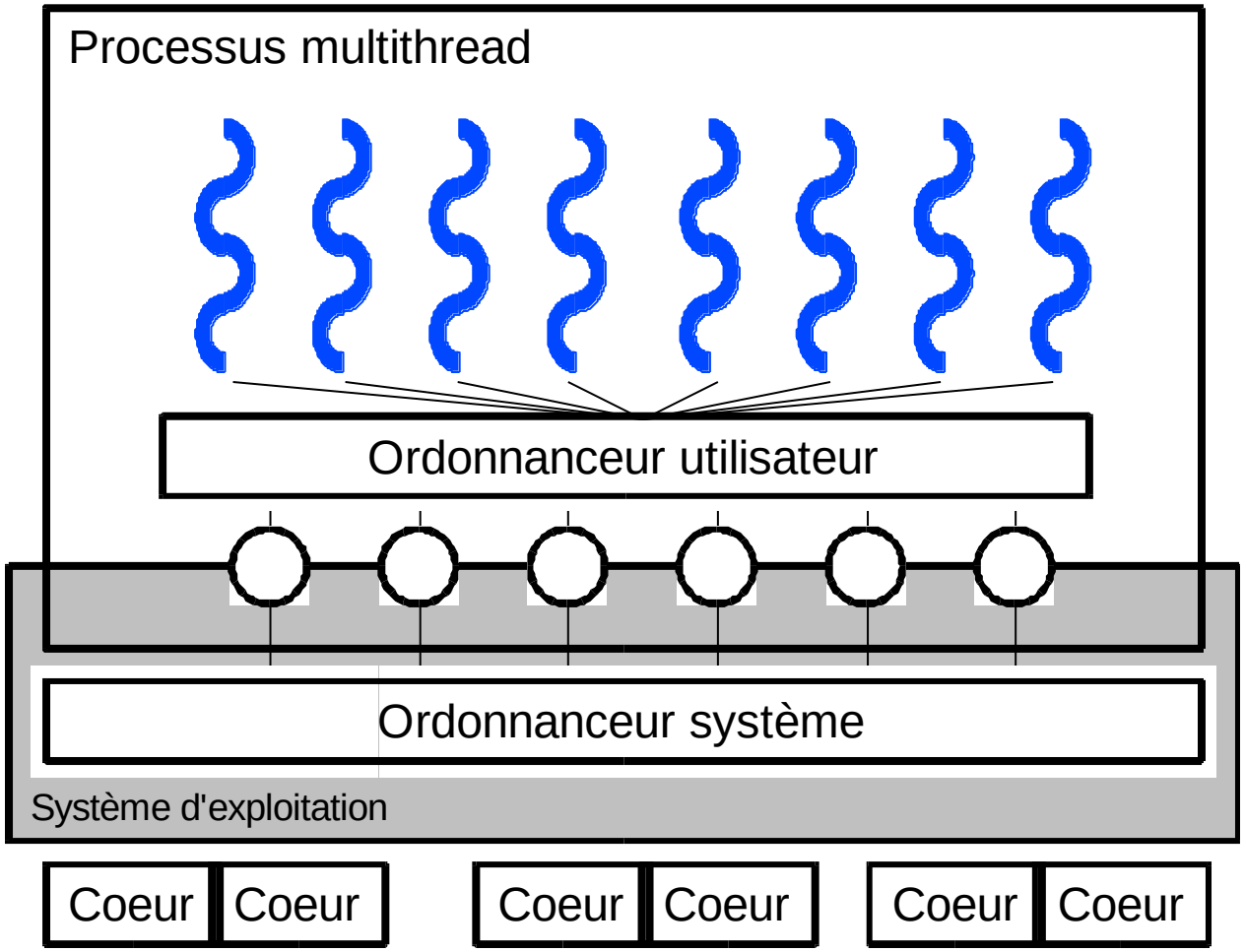
Bibliothèque système



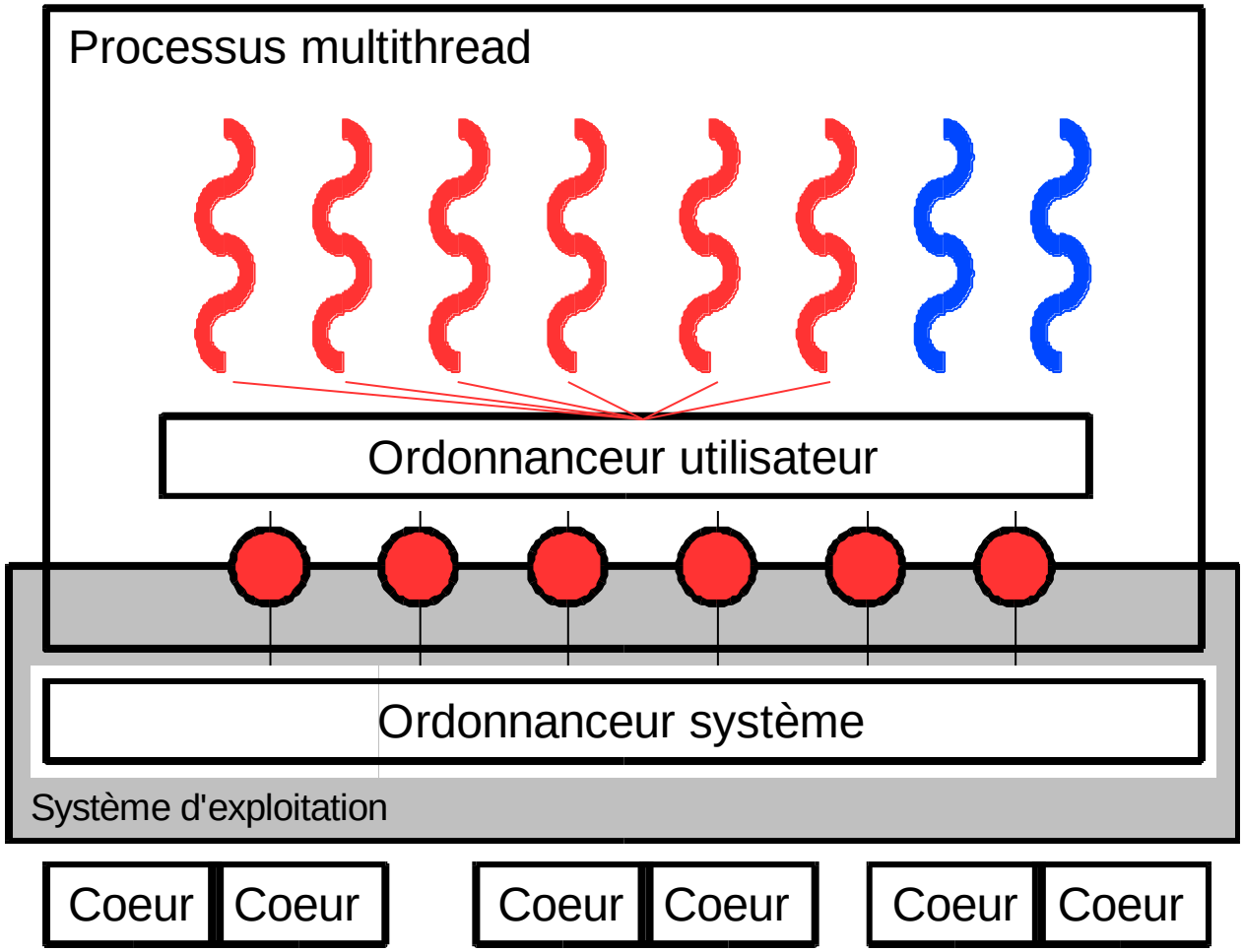
Bibliothèque système

- N threads noyau.
  - Entièrement au niveau système.
- Avantages
  - Adaptée au SMP et multicoeur.
  - Gère bien les appels systèmes.
- Inconvénients
  - Complexe à mettre en oeuvre.
  - Coût plus élevé.
- Exemple : Linuxthread, NPTL

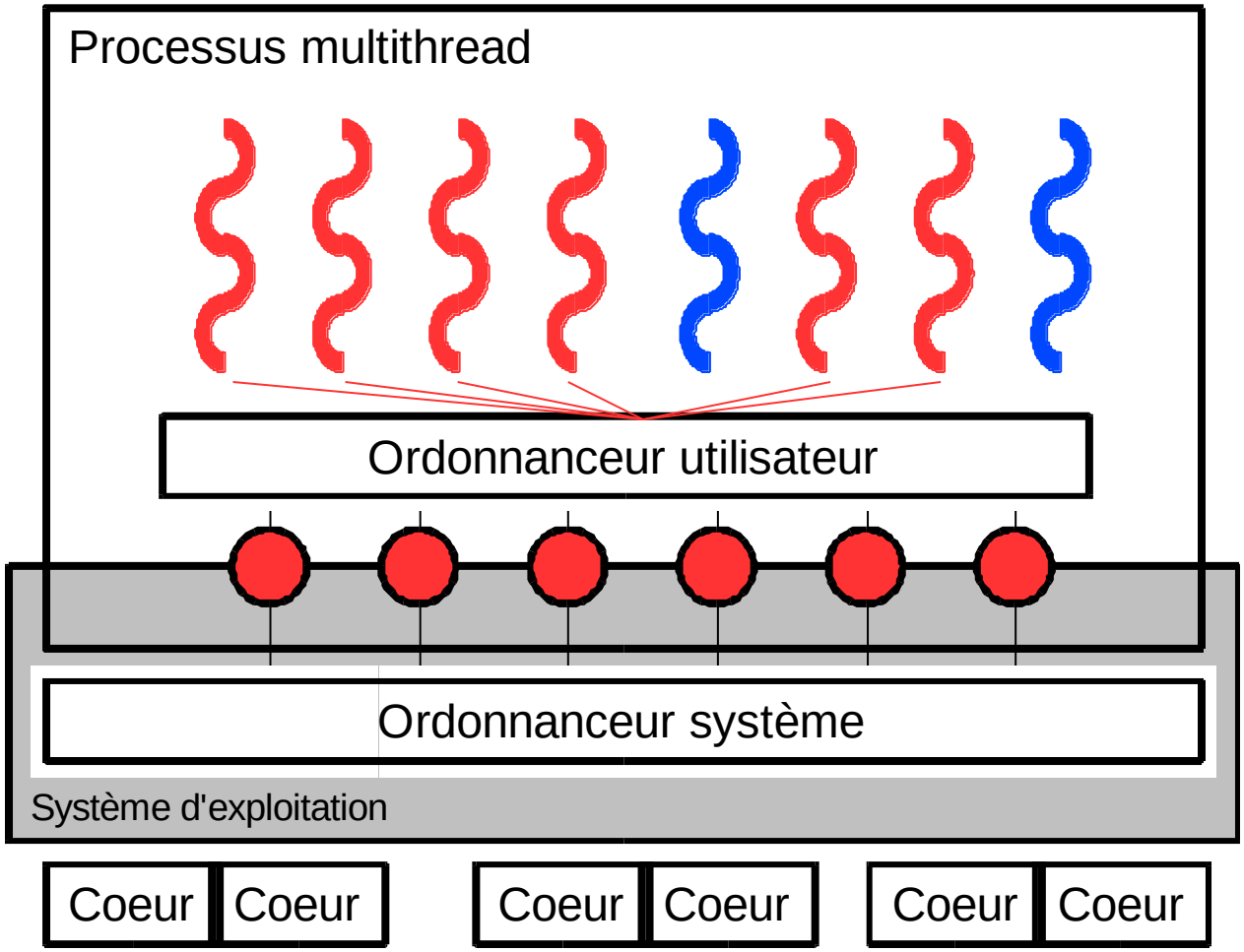
Bibliothèque mixte (ou MxN)



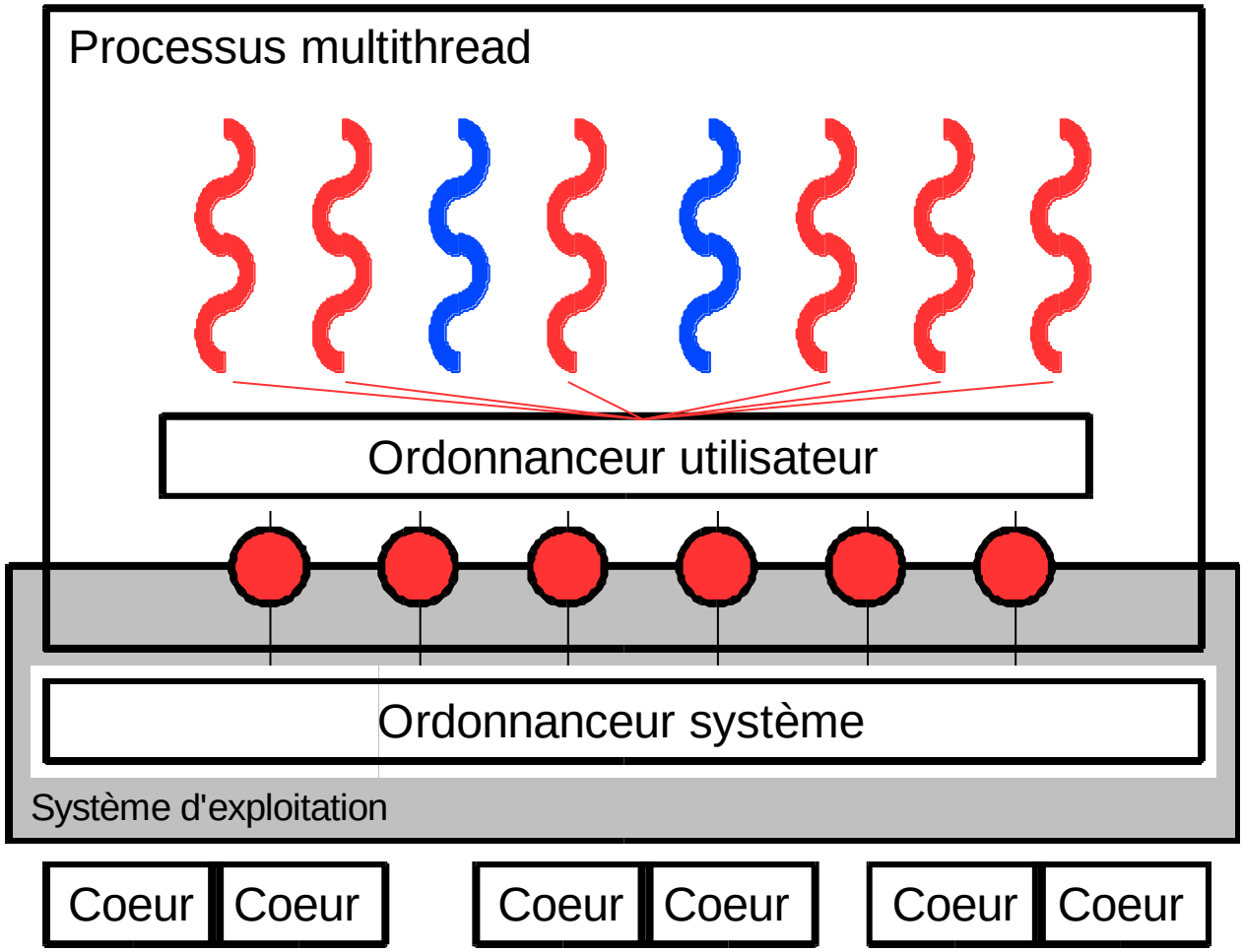
Bibliothèque mixte (ou MxN)



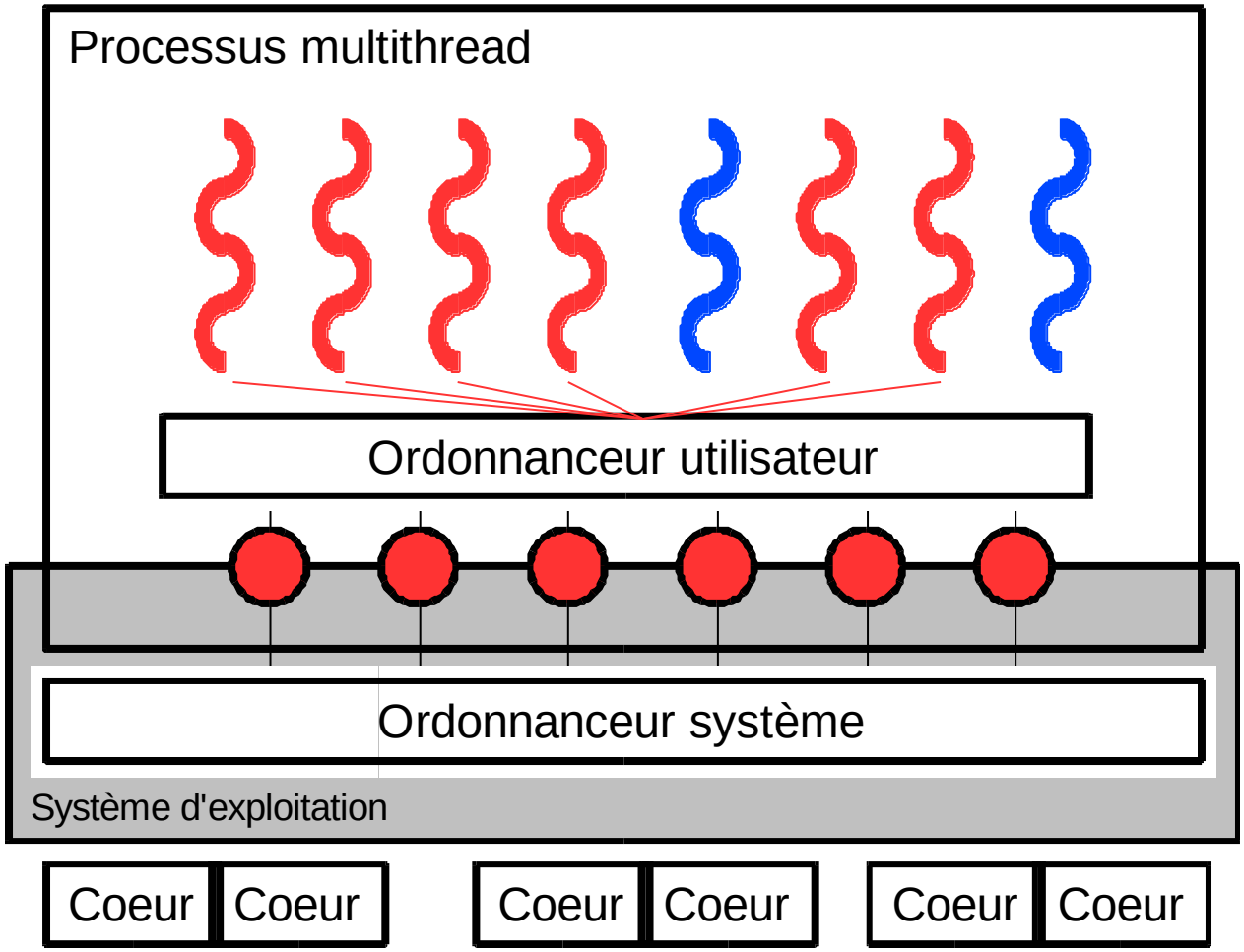
Bibliothèque mixte (ou MxN)



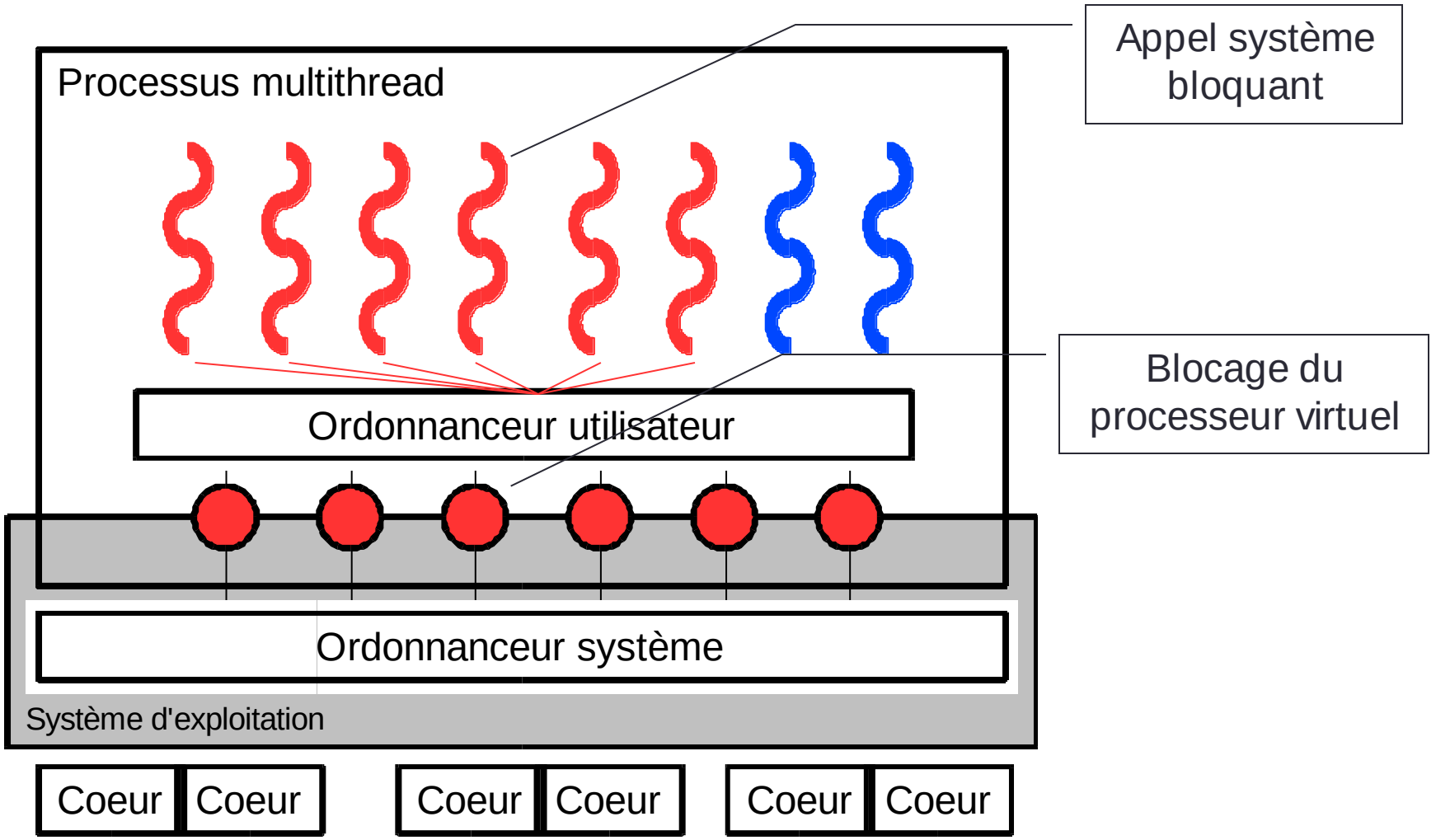
Bibliothèque mixte (ou MxN)



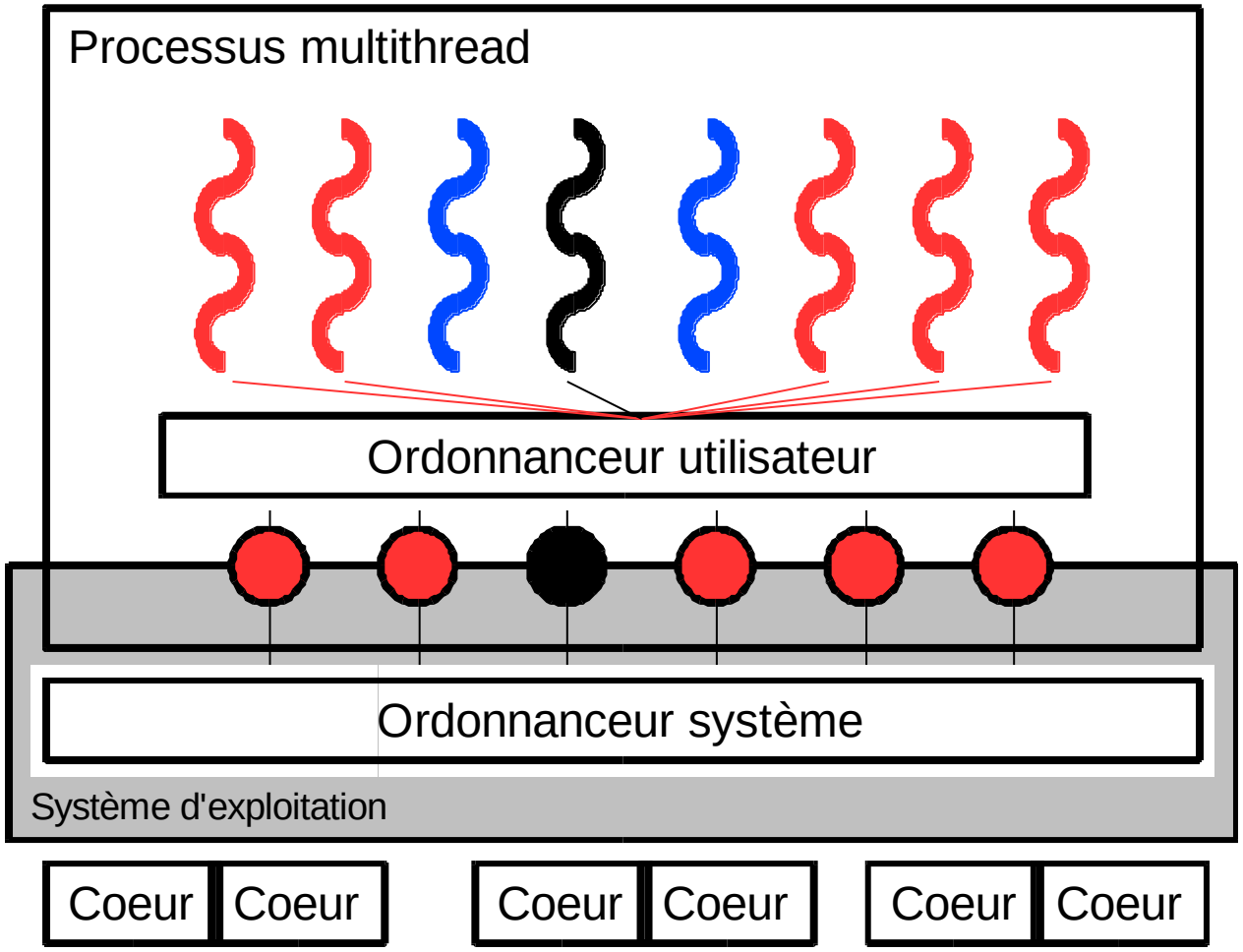
Bibliothèque mixte (ou MxN)



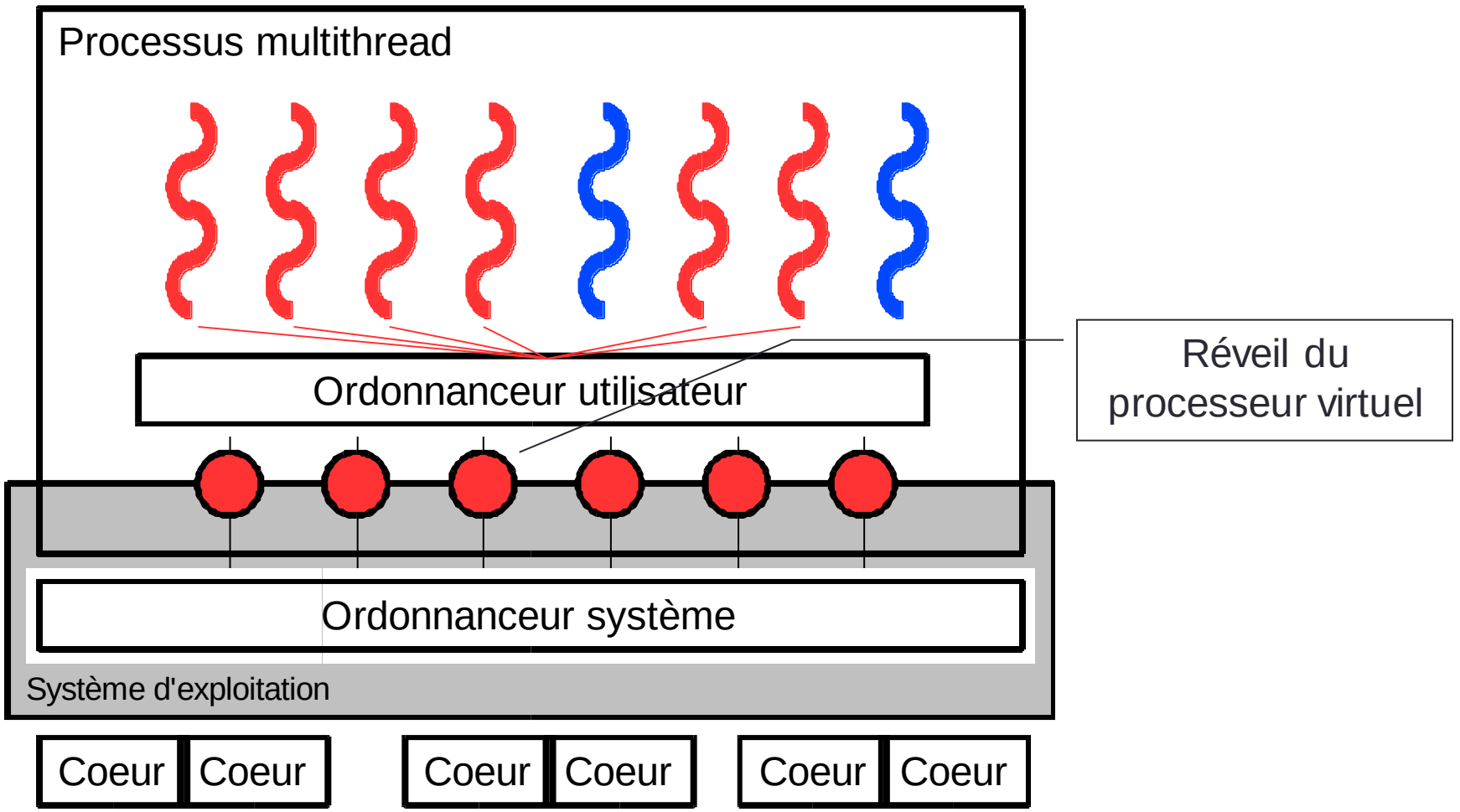
Bibliothèque mixte (ou MxN)



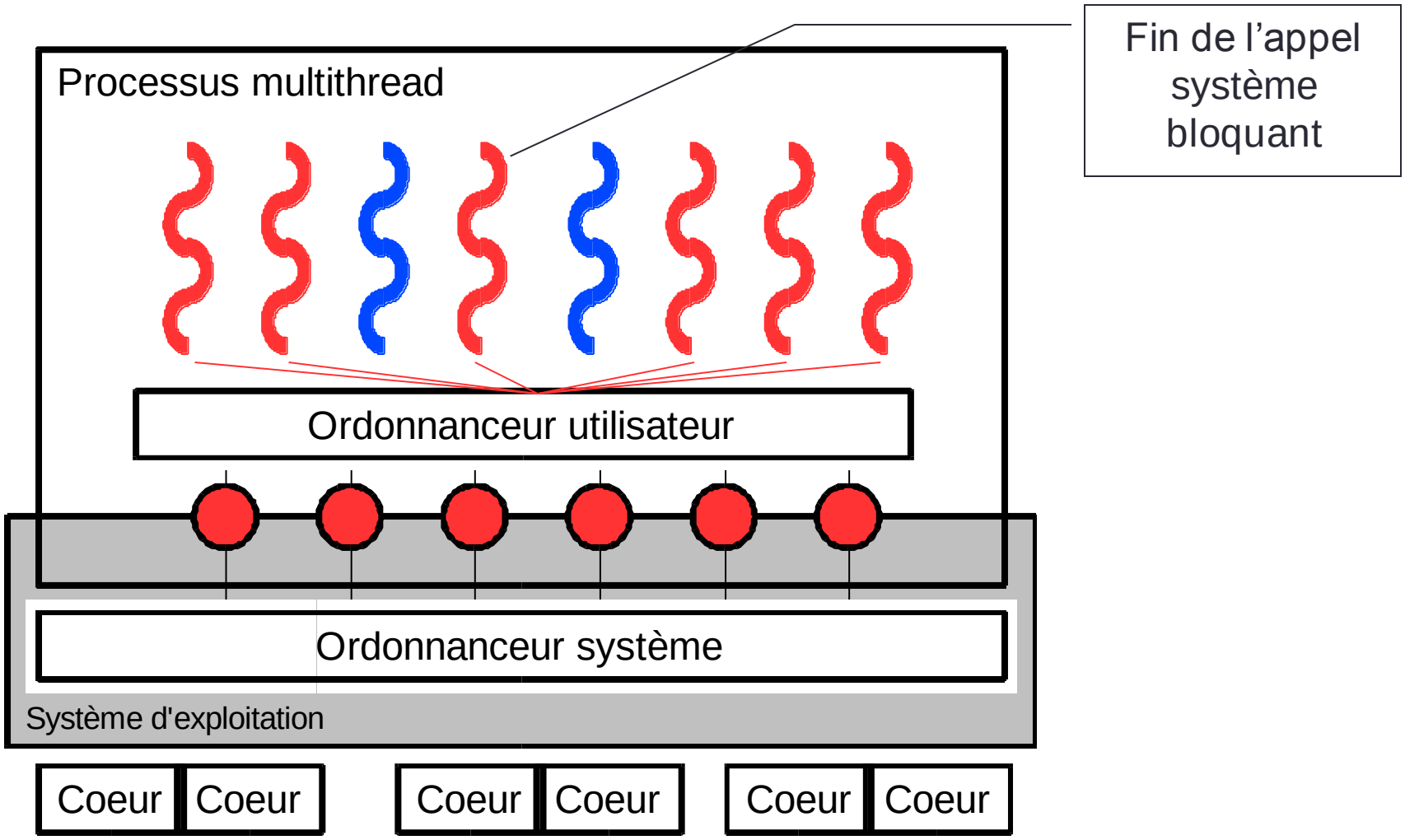
Bibliothèque mixte (ou MxN)



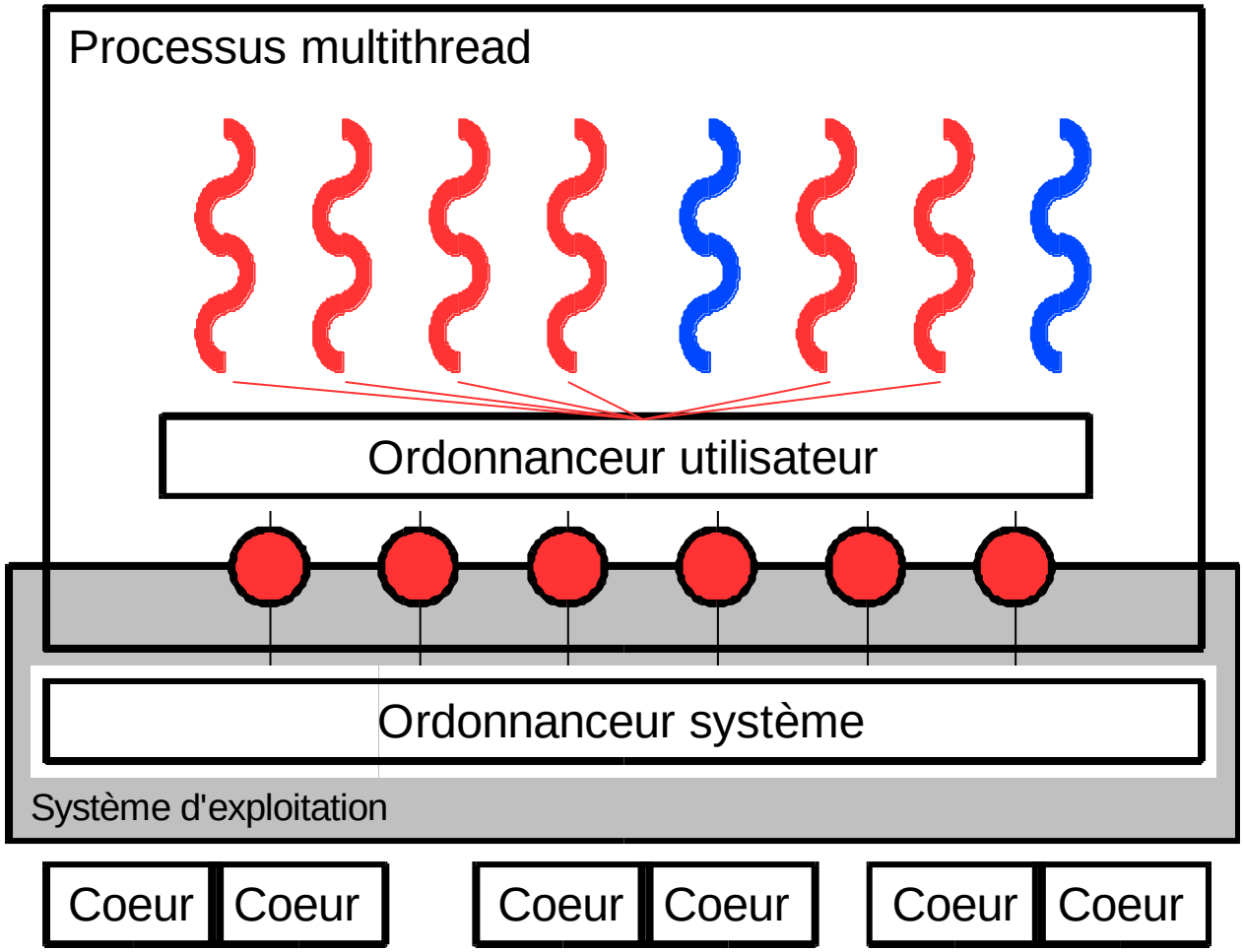
Bibliothèque mixte (ou MxN)



Bibliothèque mixte (ou MxN)



Bibliothèque mixte (ou MxN)



Bibliothèque mixte (ou MxN)

- M threads noyau pour N threads utilisateurs.
  - Deux odonnanceurs : un système et un utilisateur.
- Avantages
  - Adaptée aux SMP et multicoeur.
  - Performante.
- Inconvénients
  - Les plus complexes à implémenter.
  - Quelques problèmes avec les appels systèmes.
- Exemple : Solaris threads, MPC (http://mpc.hpcframework.com).

Tableau récapitulatif

Bibliothèque	Performances	Flexibilité	SMP/NUMA	Appels systèmes bloquants
Utilisateur	+	+	-	-
Système	-	-	+	+
Mixte	+	+	+	Limité

FONCTIONNALITÉS

Réentrance

- Exemple de fonction non réentrante
  - Contenu de buffer indéterminé si invocations simultanées

```
const char * writeHexa(int i)
{
	static char buffer[0x10];
	sprintf(buffer,"0x%.8x",i);
	return(buffer);
}
```

- Version réentrante de cette fonction
  - Chaque invocation utilise des données différentes (transmises)

```
char * writeHexa_r(int i,char * buffer)
{
	sprintf(buffer,"0x%.8x",i);
	return(buffer);
}
```

Atomicité

- **Atomicité** : atomique se dit d'une instruction dont le résultat de l'exécution ne dépend pas de l'entrelacement avec d'autres instructions. Des instructions simples comme une lecture ou une écriture en mémoire sont atomiques. Cependant, les instructions atomiques complexes sont souvent plus intéressantes. Ainsi, l'incrémentation d'une variable n'est généralement pas une instruction atomique : une instruction dans un autre flot d'exécution en parallèle peut modifier la variable entre sa lecture et son écriture incrémentée.

Atomicité

```
int i = 0 ;

int main(int argc,
     char ** argv ) {
i += argc ;
printf( "%d\n", i ) ;
}
```

```
movl %esp, %ebp
movl i, %eax
addl 8(%ebp), %eax
```

Volatilité

- **Volatilité :** le compilateur va créer un problème qui s'ajoute à celui de la non-atomicité des opérations. Il effectue des optimisations en plaçant temporairement les valeurs de variables partagées dans les registres du processeur pour effectuer des calculs. Les threads accédant à la variable à cet instant ne peuvent pas se rendre compte des changements effectués sur celle-ci car sa copie en mémoire n'a pas encore été modifiée. Pour lui éviter d'effectuer ces optimisations, il faut ajouter le qualificatif *volatile* à la déclaration des objets qui seront en mémoire partagée.

Volatilité

```
int i = 0 ;

int main(int argc,
     char ** argv ) {
i += argc ;
i++ ;
printf( "%d\n", i ) ;
}
```

```
movl %esp, %ebp
movl i, %eax
addl 8(%ebp), %eax
```

```
incl %eax
```

Volatilité

```
volatile int i = 0 ;

int main(int argc,
     char ** argv ) {
i += argc ;
i++ ;
printf( "%d\n", i ) ;
}
```

```
movl %esp, %ebp
movl i, %eax
addl 8(%ebp), %eax
movl %eax, i
```

```
movl i, %eax
incl %eax
movl %eax, i
```

Synchronisation

- Problème du producteur/consommateur :
  - le programme est constitué de deux threads ; le premier lit les caractères au clavier et le second se charge de les afficher.

- Notes
  - Le thread principal (le père) se charge de la création de ses fils et de l'attente de leur mort.
  - Cette disparition est programmée à l'arrivée du caractère "F".

Synchronisation

- Comment faire des synchronisations entre threads ?
  - Il suffit de mettre en place une politique d'attente active.

- Quel est le principal défaut de la méthode choisie précédemment ?
  - Elle est très gourmande en temps CPU et peut donc perturber les autres applications.

Synchronisation

```
volatile char theChar = '\0';
volatile char afficher = 0;

void *lire (void *name)
{
    do {
        while (afficher == 1); /* attendre mon tour */
        theChar = getchar ();
        afficher = 1; /* donner le tour */
    } while (theChar != 'F');
    return NULL;
}

void *affichage (void *name)
{
    do {
        while (afficher == 0); /* attendre */
        printf ("car = %c\n", theChar);
        afficher = 0; /* donner le tour */
    } while (theChar != 'F');
    return NULL;
}
```

Synchronisation

```
int main (void)
{
    pthread_t filsA, filsB;
    if (pthread_create (&filsA, NULL, affichage, "AA")) {
        perror ("pthread create");
        exit (EXIT_FAILURE);
    }
    if (pthread_create (&filsB, NULL, lire, "BB")) {
        perror ("pthread create");
        exit (EXIT_FAILURE);
    }
    if (pthread_join (filsA, NULL))
        perror ("pthread join");
    if (pthread_join (filsB, NULL))
        perror ("pthread join");
    printf ("Fin du pere\n");
    return (EXIT_SUCCESS);
}
```

Synchronisation

- Que peut-on observer concernant l'utilisation CPU?
  - Un processeur est chargé à 100% de temps user. Comme on peut le voir avec la commande ps -AeLf

```
$ ps -AeLf
UID PID PPID LWP C NLWP STIME TTY TIME CMD
perache 10857 10666 10857 0   3 08:31 pts/4 00:00:00 ./a.out
perache 10857 10666 10858 99 3 08:31 pts/4 00:01:45 ./a.out
perache 10857 10666 10859 0   3 08:31 pts/4 00:00:00 ./a.out
```

- Sur architecture mono-processeur, cela peut complètement bloquer la réactivité du système!

Synchronisation

- Pourquoi ?
  - La boucle *while* du thread qui affiche consomme tout le temps CPU car le thread ne passe jamais la main aux autres threads.
- Comment y remédier ?
  - Un moyen simple d'y remédier, est de mettre un sched_yield dans le while.
  - Il faut remplacer

```
while (afficher == 0) ;
```

  - par

```
while (afficher == 0) sched_yield();
```

Synchronisation

- Conséquence
  - Le système redevient réactif
  - Mais
    - Perturbations par notre application car la priorité de la tâche est très haute

- Solution
  - Modification en utilisant la fonction usleep

```
while (afficher == 0) usleep(5);
```

Synchronisation

- La commande ps -AeLf donne

```
$ ps -AeLf
UID PID PPID LWP C NLWP STIME TTY TIME CMD
perache 11103 10666 11103 0 3 08:47 pts/4 00:00:00 ./a.out
perache 11103 10666 11104 0 3 08:47 pts/4 00:00:00 ./a.out
perache 11103 10666 11105 0 3 08:47 pts/4 00:00:00 ./a.out
```

- Problème :
  - le code manque de réactivité.

Résumé

- Programmation mémoire partagée
  - Utilisation de threads
  - Les threads d'un même processus partage l'espace mémoire
- Synchronisation
  - Nécessité de synchroniser l'accès aux données partagées
  - Mode de communication des threads
  - Garder en tête la notion d'atomicité et de volatilité pour éviter/comprendre les bugs !

Producteur/consommateur

- Problème du producteur/consommateur :
  - le programme est constitué de deux threads ; le premier lit les caractères au clavier et le second se charge de les afficher.

- Notes
  - Le thread principal (le père) se charge de la création de ses fils et de l'attente de leur mort.
  - Cette disparition est programmée à l'arrivée du caractère "F".

Solution précédente

```
volatile char theChar = '\0';
volatile char afficher = 0;

void *lire (void *name)
{
    do {
        while (afficher == 1); /* attendre mon tour */
        theChar = getchar ();
        afficher = 1; /* donner le tour */
    } while (theChar != 'F');
    return NULL;
}

void *affichage (void *name)
{
    do {
        while (afficher == 0); /* attendre */
        printf ("car = %c\n", theChar);
        afficher = 0; /* donner le tour */
    } while (theChar != 'F');
    return NULL;
}
```

Solution précédente

```
int main (void)
{
    pthread_t filsA, filsB;
    if (pthread_create (&filsA, NULL, affichage, "AA")) {
        perror ("pthread create");
        exit (EXIT_FAILURE);
    }
    if (pthread_create (&filsB, NULL, lire, "BB")) {
        perror ("pthread create");
        exit (EXIT_FAILURE);
    }
    if (pthread_join (filsA, NULL))
        perror ("pthread join");
    if (pthread_join (filsB, NULL))
        perror ("pthread join");
    printf ("Fin du pere\n");
    return (EXIT_SUCCESS);
}
```

EXCLUSION MUTUELLE (MUTEX)

Mutex - Définition

- Définition:
  - un **mutex** est un objet d'exclusion mutuelle. Il sert à protéger des données partagées de modification concurrentes et permet d'implémenter des section critiques.

- Un mutex à deux états:
  - Déverrouillé (pris par aucun thread).
  - Verrouillé (appartenant à un thread).

- Un thread qui tente de verrouiller un mutex déjà verrouillé, est suspendu jusqu'à ce que le mutex soit déverrouillé

Mutex - Définition

- Intérêt :
  - Rendre la main à l'ordonnanceur.
  - Assurer l'équité
- Limitations :
  - Un mutex ne peut être pris que par un seul thread à la fois.
  - Il ne peut y avoir qu'un seul thread dans la section critique.
  - Seul le thread qui a verrouillé le mutex peut le déverrouiller

Mutex - Primitives

- Initialisation d'un mutex
  - `int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *mutexattr)`
- Initialisation statique d'un mutex
  - `pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;`
- Destruction d'un mutex
  - `int pthread_mutex_destroy(pthread_mutex_t *mutex)`

Mutex - Primitives

- Verrouillage d'un mutex
  - `int pthread_mutex_lock(pthread_mutex_t *mutex)`
- Déverrouillage d'un mutex
  - `int pthread_mutex_unlock(pthread_mutex_t *mutex)`
- Tentative de verrouillage
  - `int pthread_mutex_trylock(pthread_mutex_t *mutex)`
  - La valeur de retour permet de déterminer si le mutex a été acquis ou non.

Mutex - Exemple

- Une variable globale partagée `x` peut être protégée par un mutex
  - Tous les accès et modifications de x doivent être entourés de paires d'appels à `pthread_mutex_lock()` et `pthread_mutex_unlock()`

- Exemple :

```
int x;
pthread_mutex_t mut = PTHREAD_MUTEX_INITIALIZER;

/* … */

pthread_mutex_lock(&mut);
/* opération sur x */
pthread_mutex_unlock(&mut);
```

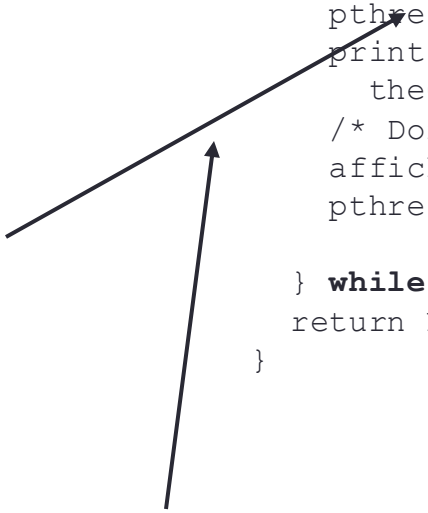
Producteur/consommateur (rappel)

```
volatile char theChar = '\0';
volatile char afficher = 0;

void *lire (void *name)
{
    do {
        while (afficher == 1); /* attendre mon tour */
        theChar = getchar ();
        afficher = 1; /* donner le tour */
    } while (theChar != 'F');
    return NULL;
}

void *affichage (void *name)
{
    do {
        while (afficher == 0); /* attendre */
        printf ("car = %c\n", theChar);
        afficher = 0; /* donner le tour */
    } while (theChar != 'F');
    return NULL;
}
```

Producteur/consommateur

```
volatile char theChar = '\0';
volatile char afficher = 0;
pthread_mutex_t lock =
  PTHREAD_MUTEX_INITIALIZER;

void *lire (void *name)
{
  do {
    /* Attendre mon tour */
    while (afficher == 1);

    pthread_mutex_lock(&lock);
    /* Donner le tour */
    afficher = 1;
    theChar = getchar ();
    pthread_mutex_unlock(&lock);

  } while (theChar != 'F');
  return NULL;
}
```

```
void *affichage (void *name)
{
  do {
    /* Attendre mon tour */
    while (afficher == 0);

    pthread_mutex_lock(&lock);
    printf ("car = %c\n",
      theChar);
    /* Donner le tour */
    afficher = 0;
    pthread_mutex_unlock(&lock);

  } while (theChar != 'F');
  return NULL;
}
```



Déplacement de l'affectation de `afficher` pour éviter l'attente active sur l'écriture

Anatomie d'un mutex

- Structure d'un mutex ainsi que d'un slot pour gérer la liste des threads en attente

```
typedef struct slot_s {
  thread_t *thread;
  struct slot_s *next;
} slot_t;

typedef struct {
  /* Compteur de threads dans ou en attente de section critique */
  volatile int nb_threads;
  /* Liste des thread bloqués */
  volatile slot_t *list_first;
  volatile slot_t *list_last;
  /* Spinlock pour verrouiller les accès aux données du mutex */
  spinlock_t lock;
} mutex_t;
```

Anatomie d'un mutex

- Fonction de verrouillage : lock

```
void mutex_lock (mutex_t * m)
{
    slot_t slot;
    spinlock (&(m→lock));
    if (m→nb_thread == 0) {
        m→nb_thread = 1;
        spinunlock (&(m→lock));
    } else {
        slot.thread = thread_self ();
        enqueue (&slot, m);
        thread_self ()→status = blocked;
        register_spinunlock (&(m→lock));
        yield ();
    }
}
```

Anatomie d'un mutex

- Fonction de déverrouillage : unlock

```
void mutex_unlock (mutex_t * m)
{
    slot_t *slot;
    spinlock (&(m→lock));
    if (m→list_first != NULL) {
        slot = dequeue (m);
        wake (slot→thread);
    } else {
        m→nb_thread = 0;
    }
    spinunlock (&(m→lock));
}
```

Anatomie d'un mutex

- Fonction de test : trylock

```
int mutex_trylock (mutex_t * m)
{
    slot_t slot;
    spinlock (&(m→lock));
    if (m→nb_thread == 0) {
        m→nb_thread = 1;
        spinunlock (&(m→lock));
    return 0;
    }
    spinunlock (&(m→lock));
    return 1;
}
```

Anatomie d'un mutex

- Fonction de test : trylock

```
int mutex_trylock (mutex_t * m)
{
     slot_t slot;
     int res = 1;
     spinlock (&(m→lock));
     if (m→nb_thread == 0) {
          m→nb_thread = 1;
          res = 0;
          goto fin;
     }
fin:
     spinunlock (&(m→lock));
     return res;
}
```

Anatomie d'un mutex récursif

- Fonction de verrouillage : lock

```
void mutex_lock (mutex_t * m) {
    slot_t slot;
    spinlock (&(m→lock));
    if (m→nb_thread == 0) {
    m→nb_thread = 1;
    m→owner = thread_self ();
    spinunlock (&(m→lock));
    } else {
        if (m→owner == thread_self ()) {
            m→step++;
            spinunlock (&(m→lock));
        } else {
            slot.thread = thread_self ();
            enqueue (&slot, m);
            thread_self ()→status = blocked;
            register_spinunlock (&(m→lock));
            yield ();
        }
    }
}
```

Anatomie d'un mutex récursif

- Fonction de verrouillage : unlock

```
void mutex_unlock (mutex_t * m) {
    slot_t *slot;
    spinlock (&(m→lock));
    if (m→step == 1) {
        m→step--;
        if (m→list_first != NULL) {
            slot    = dequeue (m);
            wake (slot→thread);
        } else {
            m→nb_thread = 0;
        }
    } else {
        m→step--;
    }
    spinunlock (&(m→lock));
}
```

Anatomie d'un mutex récursif

- Fonction de test : trylock

```
int mutex_trylock (mutex_t * m){
    slot_t slot;
    spinlock (&(m→lock));
    if (m→nb_thread == 0) {
        m→nb_thread = 1;
        m→owner = thread_self ();
        spinunlock (&(m→lock));
        return 0;
    } else {
        if (m→owner == thread_self ()) {
            m→step++;
            spinunlock (&(m→lock));
            return 0;
        }
    }
    spinunlock (&(m→lock));
    return 1;
}
```

SÉMAPHORES

Sémaphore - Définition

- Définition :
  - les **sémaphores** sont des compteur pour des ressources partagées par plusieurs threads.
- Opérations sur les sémaphores
  - Incrémenter le compteur
  - Décrémenter le compter
  - Si l'on essaye de décrémenter un compteur déjà à zéro, alors le thread est bloqué
- Pour utiliser les sémaphores, il faut utiliser l'include `semaphore.h`

Sémaphore - Fonctions

- Initialisation d'un sémaphore
  - `int sem_init(sem_t *sem, int pshared, unsigned int value);`
  - Sémaphore non nommé
  - L'argument `value` spécifie la valeur initiale du sémaphore.
  - L'argument `pshared` indique si ce sémaphore sera partagé entre les threads d'un processus ou entre processus.
- Destruction d'un sémaphore
  - `int sem_destroy(sem_t *sem);`

Sémaphore - Fonctions

- Décrémentation (verrouillage) un sémaphore
  - `int sem_wait(sem_t *sem);`
  - Si la valeur du sémaphore est plus grande que 0, la décrémentation s'effectue et la fonction revient immédiatement.
  - Si le sémaphore vaut zéro, l'appel bloquera jusqu'à ce que soit il devienne disponible d'effectuer la décrémentation (c'est-à-dire la valeur du sémaphore n'est plus nulle).

Sémaphore - Fonctions

- Test de verrouillage
  - `int sem_trywait(sem_t *sem);`
  - Si la décrémentation ne peut pas être effectuée immédiatement, l'appel renvoie une erreur
  - La variable errno vaut EAGAIN (l'appel n'est alors pas bloquant)
- Incrémentation (déverrouillage) d'un sémaphore
  - `int sem_post(sem_t *sem);`
  - Si, à la suite de cet incrément, la valeur du sémaphore devient supérieure à zéro, un autre processus ou thread bloqué dans un appel sem_wait sera réveillé et procédera au verrouillage du sémaphore.

Producteur/consommateur (rappel)

```
volatile char theChar = '\0';
volatile char afficher = 0;

void *lire (void *name) {
    do {
        while (afficher == 1); /* attendre mon tour */
        theChar = getchar ();
        afficher = 1; /* donner le tour */
    } while (theChar != 'F');
    return NULL;
}
void *affichage (void *name) {
    do {
        while (afficher == 0); /* attendre */
        printf ("car = %c\n", theChar);
        afficher = 0; /* donner le tour */
    } while (theChar != 'F');
    return NULL;
}
```

Producteur/consommateur

```
volatile char theChar = '\0';
sem_t lock_lire;
sem_t lock_affiche;

void *lire (void *name) {
  do {
    sem_wait(&lock_lire);
    theChar = getchar ();
    sem_post(
        &lock_affiche);
  } while (theChar != 'F');
  return NULL;
}
```

```
void *affichage (void *name)
{
  do {
    sem_wait(
        &lock_affiche);
    printf ("car = %c\n",
  theChar);
    sem_post (&lock_lire);
  } while (theChar != 'F');
  return NULL;
}
```

CONDITIONS

Conditions

- Définition :
  - Une **condition** est un mécanisme de synchronisation permettant à un thread de suspendre son exécution jusqu'à ce qu'une certaine condition (un *prédicat*) sur des données partagées soit vérifiée.

- Opérations fondamentales sur les conditions :
  - Signaler la condition (quand le prédicat devient vrai)
  - Attendre la condition en suspendant l'exécution du thread jusqu'à ce qu'un autre thread signale la condition

- Une variable condition doit toujours être associée à un **mutex**, pour éviter une condition concurrente où un thread se prépare à attendre une condition et un autre thread signale la condition juste avant que le premier n'attende réellement.

Conditions

- Initialisation d'une variable condition
  - `int pthread_cond_init(pthread_cond_t *cond, pthread_condattr_t *cond_attr);`
  - Utilisation des attributs par défaut si `cond_attr` vaut `NULL`.
- Initialisation statique
  - Utilisation de la constante `PTHREAD_COND_INITIALIZER`.
- Relancer un des threads attendant une variable condition
  - `int pthread_cond_signal(pthread_cond_t *cond);`
  - S'il n'existe aucun thread répondant à ce critère, rien ne se produit.
  - Si plusieurs threads attendent sur cond, seul l'un d'entre eux sera relancé, mais il est impossible de savoir lequel.

Conditions

- Relancer tous les threads attendant sur une variable condition
  - `int pthread_cond_broadcast(pthread_cond_t *cond);`
  - Rien ne se passe s'il n'y a aucun thread attendant sur cond.

- Déverrouiller atomiquement le mutex et attendre qu'une variable condition soit signalée
  - `int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);`
  - L'exécution du thread est suspendue et ne consomme pas de temps CPU jusqu'à ce que la variable condition soit signalée.
  - Le mutex doit être verrouillé par le thread appelant à l'entrée de `pthread_cond_wait`().
  - Avant de rendre la main au thread appelant, `pthread_cond_wait`() reverrouille mutex.

Conditions

- Le déverrouillage du mutex et la suspension de l'exécution sur la variable condition sont liés atomiquement.
  - Si tous les threads verrouillent le mutex avant de signaler la condition, il est garanti que la condition ne peut être signalée (et donc ignorée) entre le moment où un thread verrouille le mutex et le moment où il attend sur la variable condition.
- Destruction d'une variable condition
  - `int pthread_cond_destroy(pthread_cond_t *cond);`
  - Librération des ressources qu'elle possède.
  - Aucun thread ne doit attendre sur la condition à l'entrée de `pthread_cond_destroy`().

Producteur/consommateur (rappel)

```
volatile char theChar = '\0';
volatile char afficher = 0;
void *lire (void *name) {
    do {
        while (afficher == 1); /* attendre mon tour */
        theChar = getchar ();
        afficher = 1; /* donner le tour */
    } while (theChar != 'F');
    return NULL;
}
void *affichage (void *name) {
    do {
        while (afficher == 0); /* attendre */
        printf ("car = %c\n", theChar);
        afficher = 0; /* donner le tour */
    } while (theChar != 'F');
    return NULL;
}
```

Producteur/consommateur

```
volatile char theChar = '\0';
pthread_mutex_t lock =
  PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond_lire =
  PTHREAD_COND_INITIALIZER;
pthread_cond_t cond =
  PTHREAD_COND_INITIALIZER;

void *lire (void *name){
  do {
   pthread_mutex_lock(&lock);
   theChar = getchar ();
    pthread_cond_signal(&cond);
    pthread_cond_wait(
       &cond_lire, &lock);
    pthread_mutex_unlock(&lock);
  } while (theChar != 'F');
  return NULL;
}
```

```
void *affichage (void *name){
  do {
    pthread_mutex_lock (&lock);
   pthread_cond_wait (
      &cond, &lock);
    printf ("car = %c\n",
      theChar);
    pthread_cond_signal(
      &cond_lire);
    pthread_mutex_unlock(&lock);
  } while (theChar != 'F');
  return NULL;
}
```

Producteur/consommateur

```
volatile char theChar = '\0';
pthread_mutex_t lock =
  PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond_lire =
  PTHREAD_COND_INITIALIZER;
pthread_cond_t cond =
  PTHREAD_COND_INITIALIZER;

void *lire (void *name){
  do {
   pthread_mutex_lock(&lock);
   theChar = getchar ();
    pthread_cond_signal(&cond);
    pthread_cond_wait(
       &cond_lire, &lock);
    pthread_mutex_unlock(&lock);
  } while (theChar != 'F');
  return NULL;
}
```

Bug car si lire commence on a un deadlock

```
void *affichage (void *name){
  do {
    pthread_mutex_lock (&lock);
   pthread_cond_wait (
      &cond, &lock);
    printf ("car = %c\n",
      theChar);
    pthread_cond_signal(
      &cond_lire);
    pthread_mutex_unlock(&lock);
  } while (theChar != 'F');
  return NULL;
}
```

Producteur/consommateur

```
volatile char theChar = '\0';
pthread_mutex_t lock =
  PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond_lire =
  PTHREAD_COND_INITIALIZER;
pthread_cond_t cond =
  PTHREAD_COND_INITIALIZER;
volatile int affiche = 1;

void *lire (void *name){
  do {
   while(affiche == 1);
    pthread_mutex_lock(&lock);
   affiche = 1;
    theChar = getchar ();
    pthread_cond_signal(&cond);
    pthread_cond_wait(
       &cond_lire, &lock);
    pthread_mutex_unlock(&lock);
  } while (theChar != 'F');
  return NULL;
}
```

```
void *affichage (void *name){
  do {
    pthread_mutex_lock (&lock);
   affiche = 0;
    pthread_cond_wait (
      &cond, &lock);
    printf ("car = %c\n",
      theChar);
    pthread_cond_signal(
      &cond_lire);
    pthread_mutex_unlock(&lock);
  } while (theChar != 'F');
  return NULL;
}
```

CLÉS

Clés

- Les programmes ont souvent besoin de variables globales ou statiques ayant différentes valeurs dans des threads différents.
  - Comme les threads partagent le même espace mémoire, cet objectif ne peut être réalisé avec les variables usuelles.
  - Les données spécifiques à un thread POSIX sont la réponse à ce problème.

- Chaque thread possède un segment mémoire privé, le TSD (Thread-Specific Data : Données Spécifiques au Thread).
  - Cette zone mémoire est indexée par des clés TSD.
  - La zone TSD associe des valeurs du type `void *` aux clés TSD.
  - Ces clés sont communes à tous les threads, mais la valeur associée à une clé donnée est différente dans chaque thread.

Clés

- Pour concrétiser ce formalisme :
  - les zones TSD peuvent être vues comme des tableaux de pointeurs void *,
  - les clés TSD comme des indices entiers pour ces tableaux, et
  - les valeurs des clés TSD comme les valeurs des entrées correspondantes dans le tableau du thread appelant.
- Quand un thread est créé, sa zone TSD associe initialement NULL à toutes les clés.

Clés

- Allocation d'une nouvelle clé TSD
  - `int pthread_key_create(pthread_key_t *clé, void (*destr_function) (void *));`
  - Cette clé est enregistrée à l'emplacement pointée par clé.
  - Il ne peut y avoir plus de PTHREAD_KEYS_MAX clés allouées à un instant donné.
  - La valeur initialement associée avec la clé renvoyée est NULL dans tous les threads en cours d'exécution.

- Le paramètre `destr_function`, si différent de NULL, spécifie une fonction de destruction associée à une clé.
  - Quand le thread se termine par `pthread_exit`() ou par une annulation, `destr_function` est appelée avec en argument les valeurs associées aux clés de ce thread.
  - La fonction `destr_function` n'est pas appelée si cette valeur est NULL.
  - L'ordre dans lequel les fonctions de destruction sont appelées lors de la fin du thread n'est pas spécifiée.

Clés

- Avant que la fonction de destruction soit appelée, la valeur NULL est associée à la clé dans le thread courant.
- Une fonction de destruction peut cependant réassocier une valeur différente de NULL à cette clé ou une autre clé.
- Pour gérer ce cas de figure, si après l'appel de tous les destructeurs pour les valeurs différentes de NULL, il existe toujours des valeurs différentes de NULL avec des destructeurs associés, alors la procédure est répétée.

Clés

- Désallocation d'une clé TSD
  - `int pthread_key_delete(pthread_key_t clé);`
  - Pas de vérification si des valeurs différentes de NULL sont associées avec cette clé dans les threads en cours d'exécution,
  - Pas d'appel à la fonction de destruction associée avec cette clé.
- Changement de la valeur associée avec une clé
  - `int pthread_setspecific(pthread_key_t clé, const void *pointer);`
- Accès à la valeur d'une clé
  - `void * pthread_getspecific(pthread_key_t clé);`
  - Valeur correspondant au thread appelant.

TLS

- TLS (thread local storage) :
  - Espace de stockage spécifique à chaque thread.
  - La différence avec les clés présentées auparavant, est que c'est ici le compilateur qui se charge du travail.

- Mot clé du langage : __thread
  - Exemple : __thread int v;
  - Permet de déclarer la variable v comme étant locale à chaque thread.

- Elle s'utilise ensuite (lecture/écriture) comme une variable normale.
  - Le compilateur génère certains appels de fonctions au moment de la génération de codes
  - Optimisation du *linker* pour diminuer le coût d'accès à ces variables

ARTICLE DE RECHERCHE