Programmation Parallèle et Distribuée

PERACHE Marc
marc.perache@cea.fr

Le fonctionnement du cours

- Le cours
  - Présentation des bibliothèques de threads.
  - API POSIX.
  - Les entrailles des bibliothèques.

- Les TD:
  - Découvertes de différentes bibliothèques.
  - Implémentation de fonctionnalités dans la bibliothèque mthread.
  - Tous les TD sont notés.

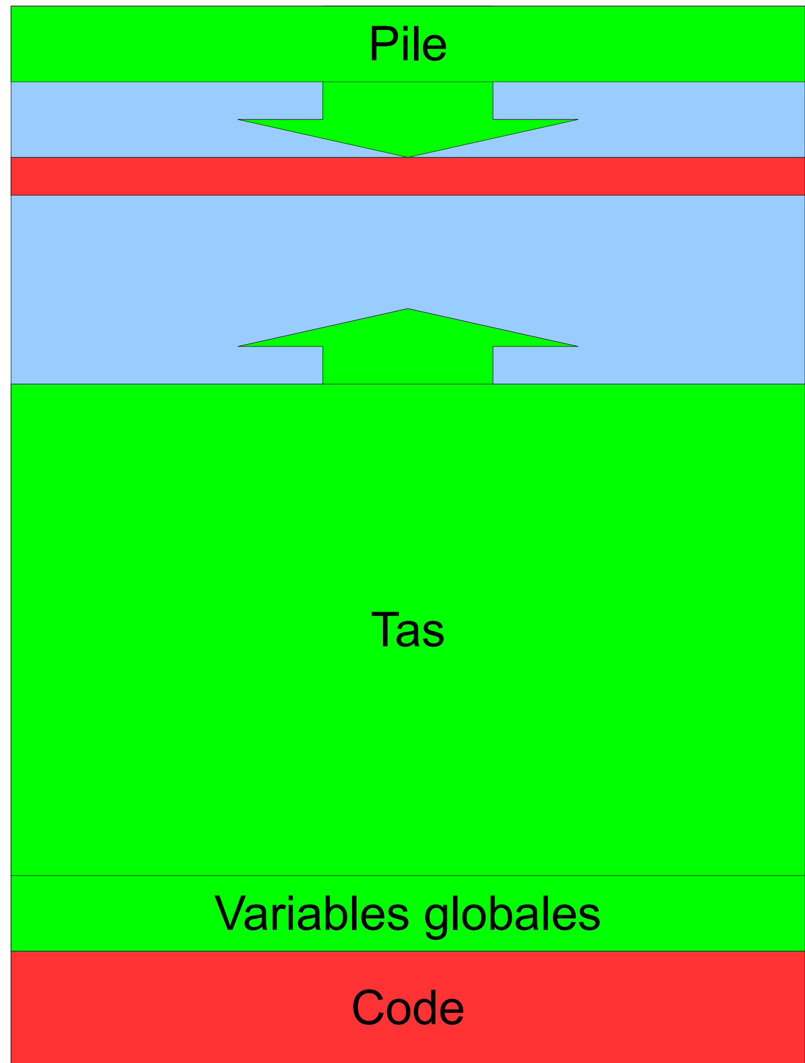
Pourquoi les threads?

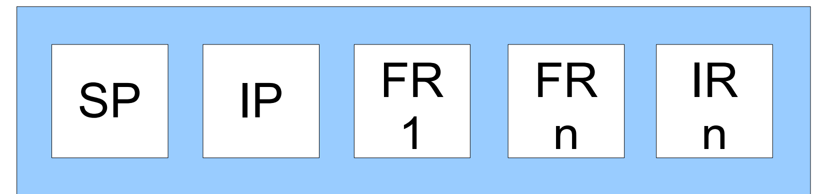
- Permet de profiter de communication instantanées.
- Profite de l'aspect mémoire partagée des noeuds de calcul.
- Permet de faire du recouvrement.
- Très adapté au multicoeur.
- Très adapté aux SMP et NUMA.

Qu'est-ce qu'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.

Qu'est-ce qu'un processus

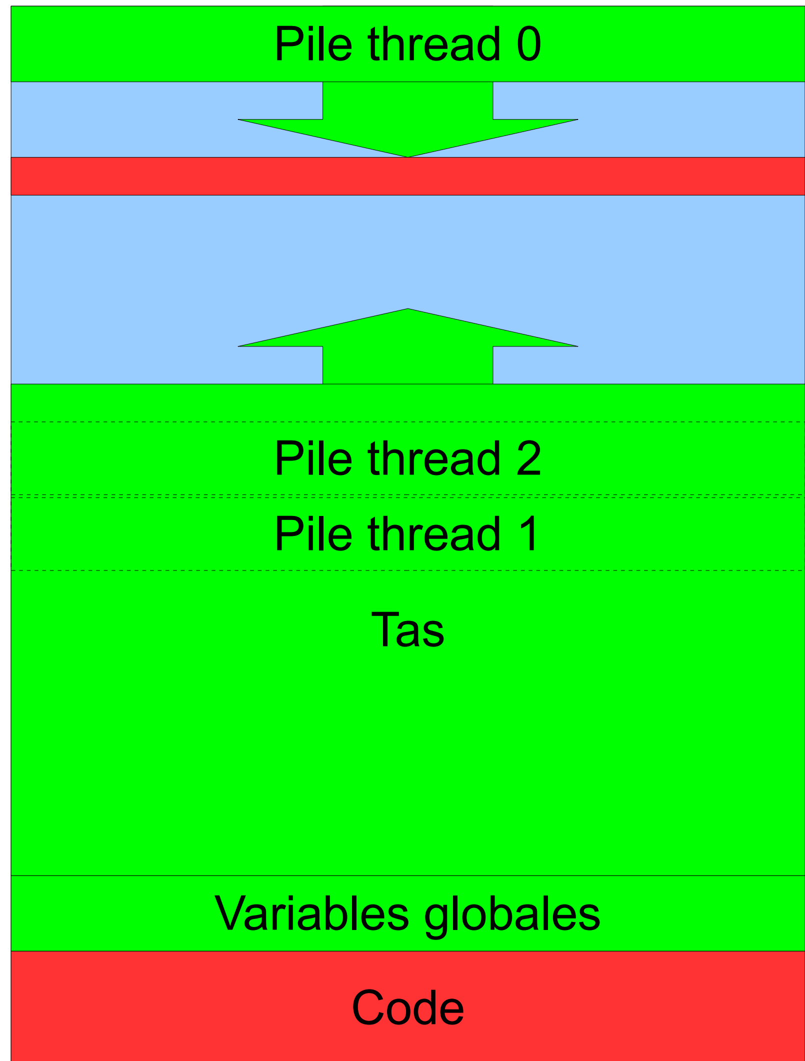


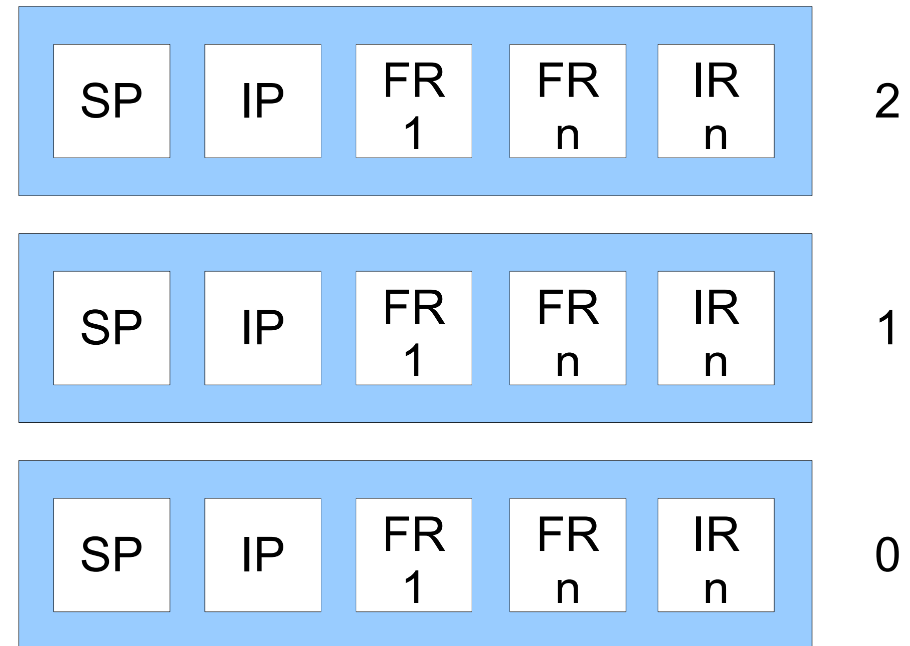


Mémoire

Structures

Qu'est-ce qu'un processus multithread





Mémoire

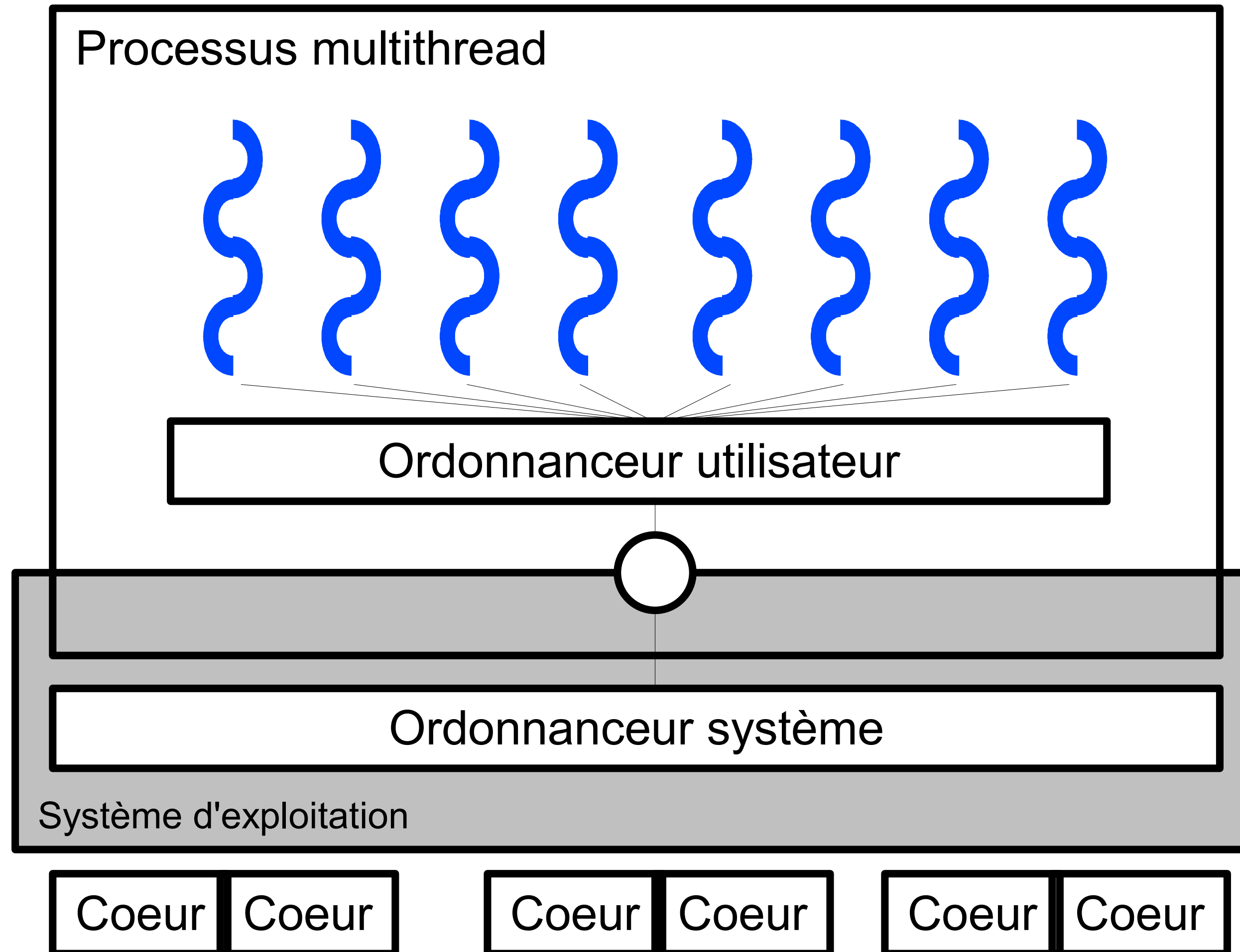
Structures

Qu'est-ce qu'un thread

- Caractéristiques:
  - Les threads partagent tous le même espace d'adressage.
  - La pile des threads (hormis le 0) ne grossit pas.
  - La pile d'un thread est localisée dans le tas.
  - Les variables globales sont toutes partagées.
  - Les threads communiquent directement par la mémoire.

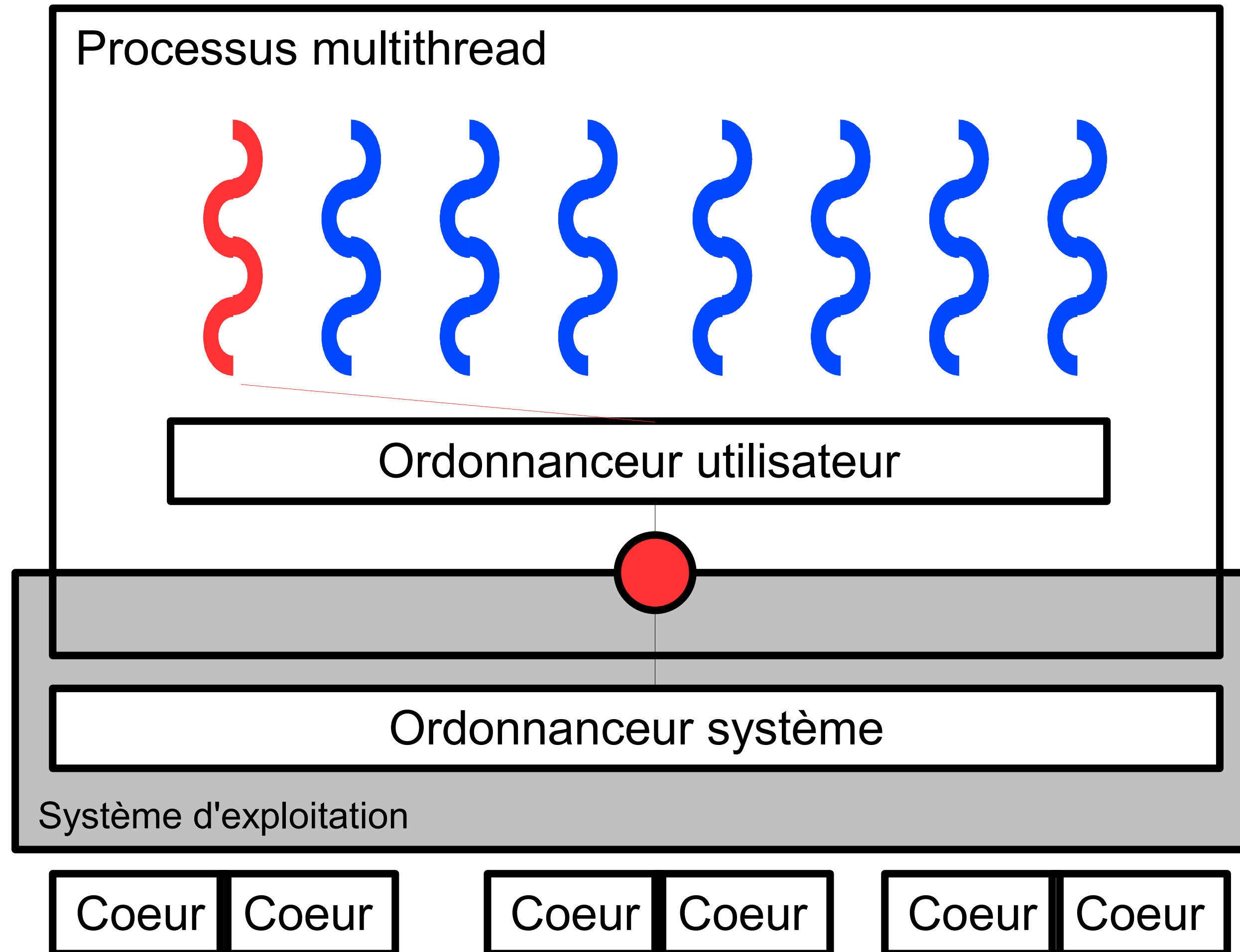
Les bibliothèques de threads

- Bibliothèque utilisateur:



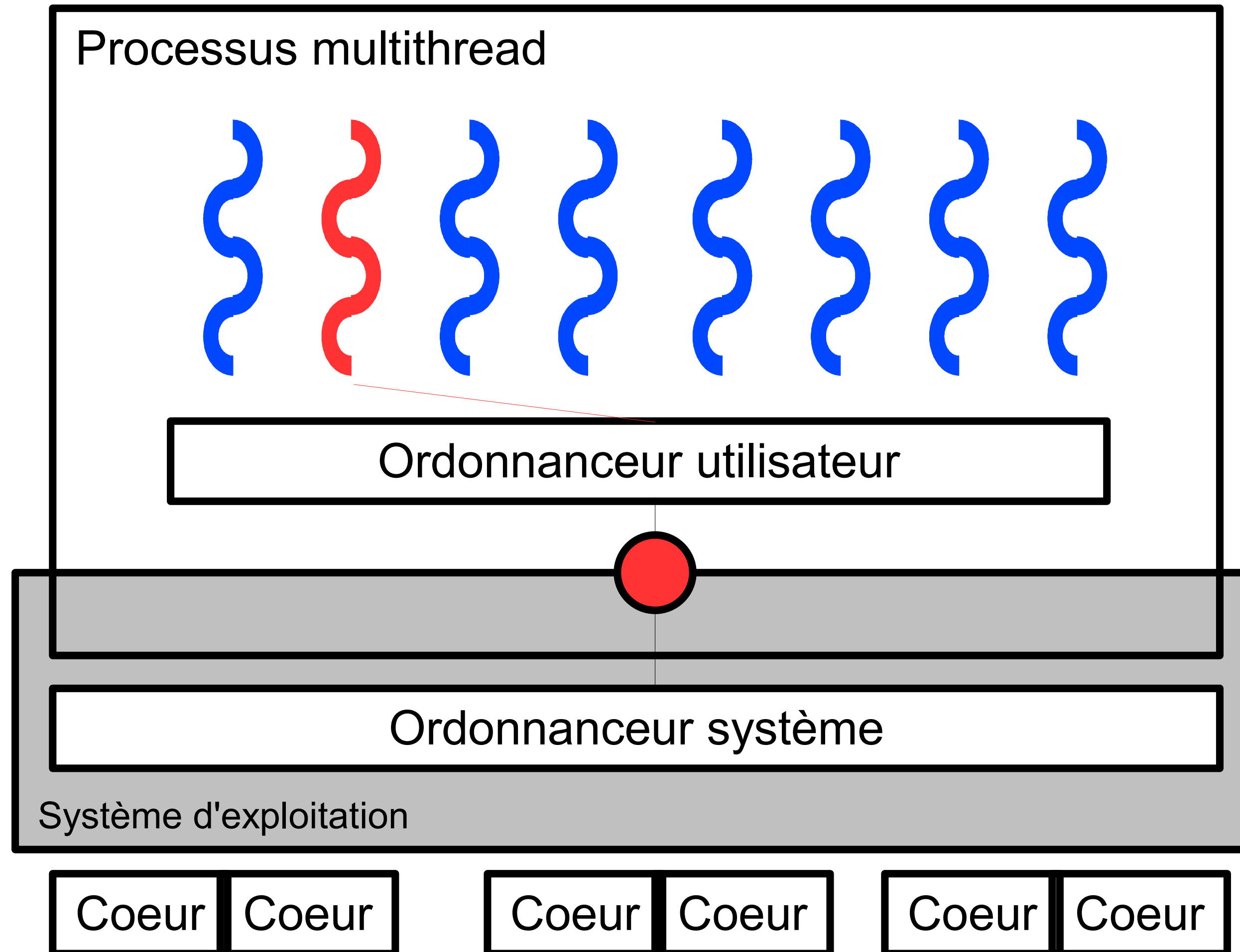
Les bibliothèques de threads

- Bibliothèque utilisateur:



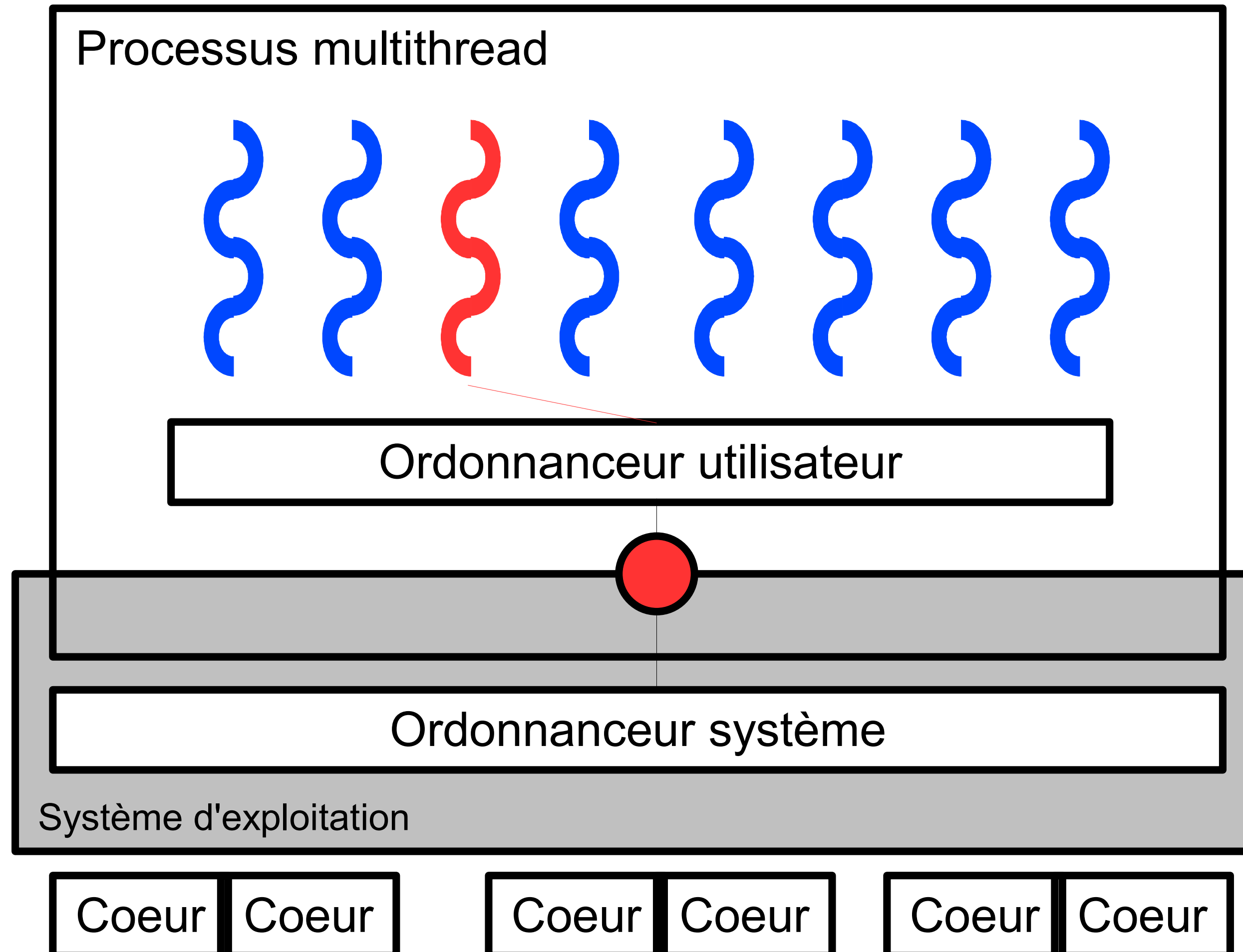
Les bibliothèques de threads

- Bibliothèque utilisateur:



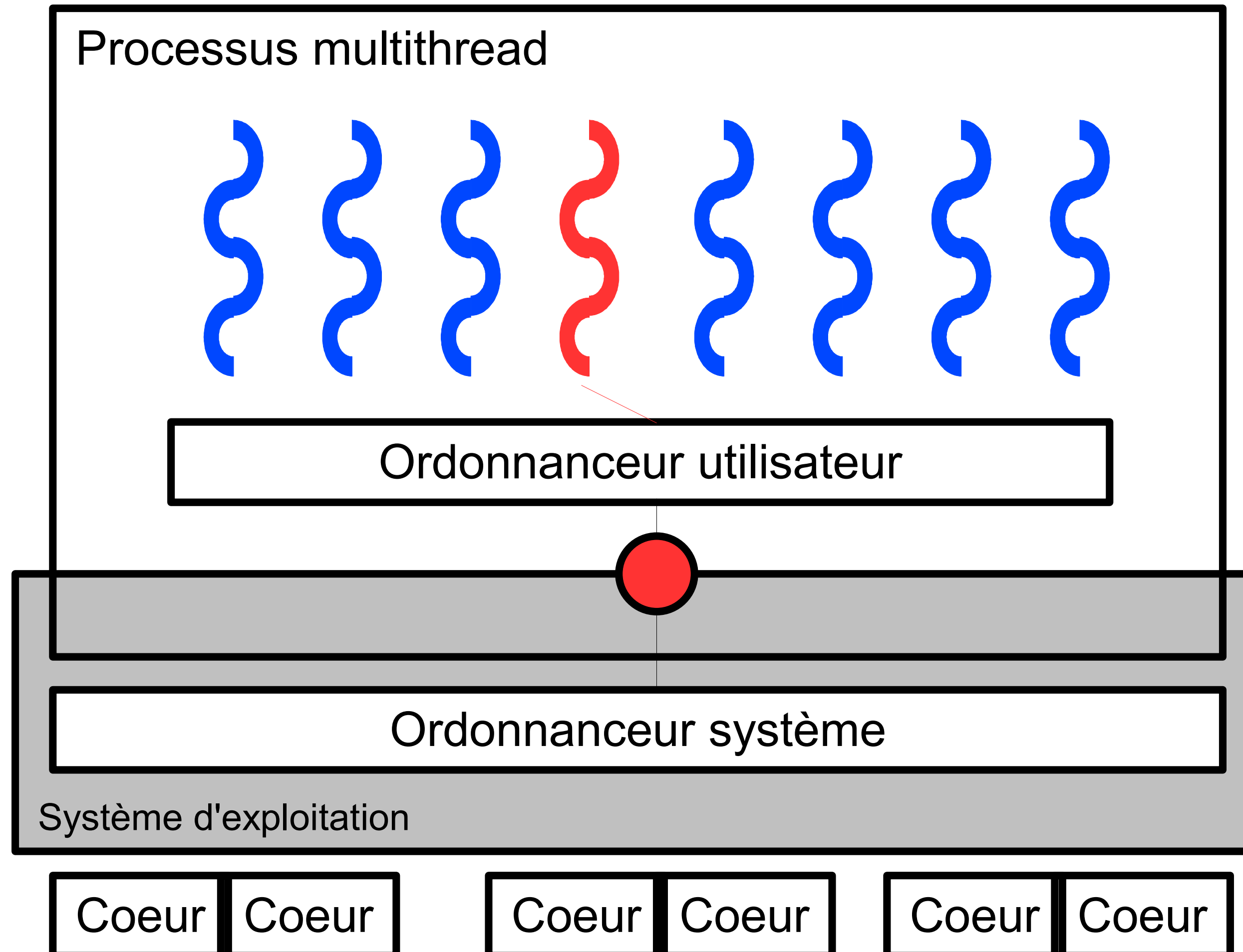
Les bibliothèques de threads

- Bibliothèque utilisateur:



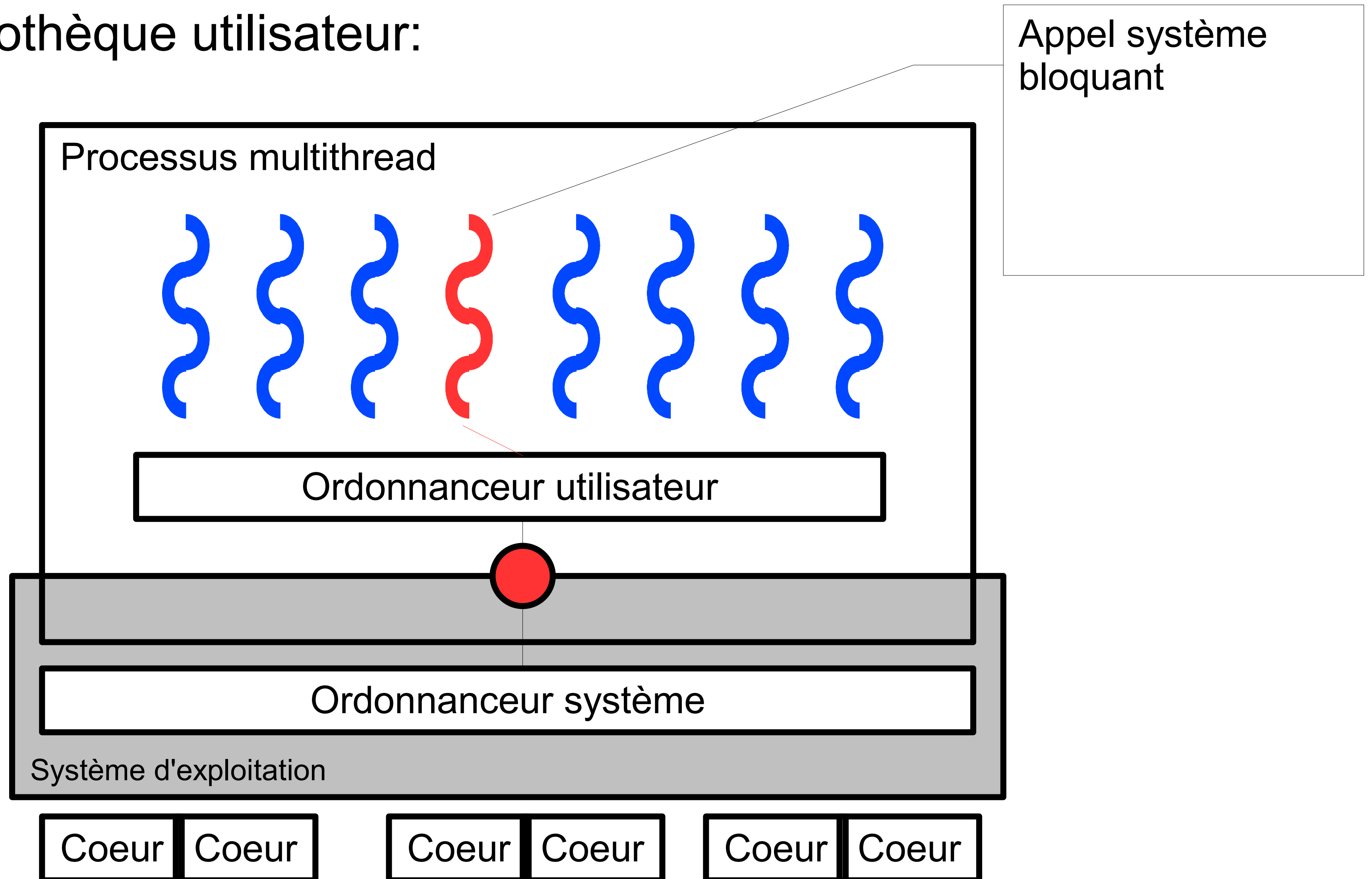
Les bibliothèques de threads

- Bibliothèque utilisateur:



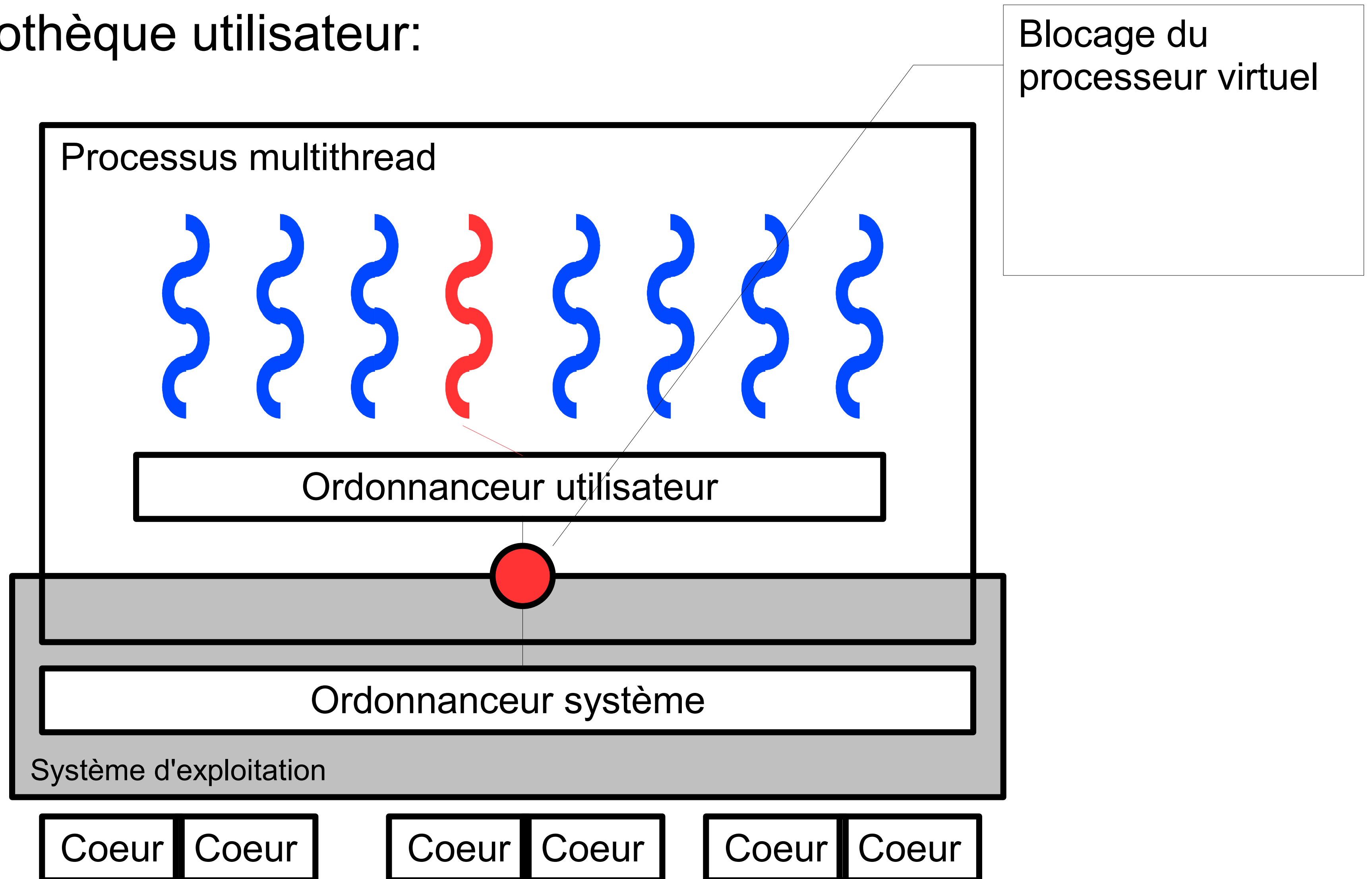
Les bibliothèques de threads

- Bibliothèque utilisateur:



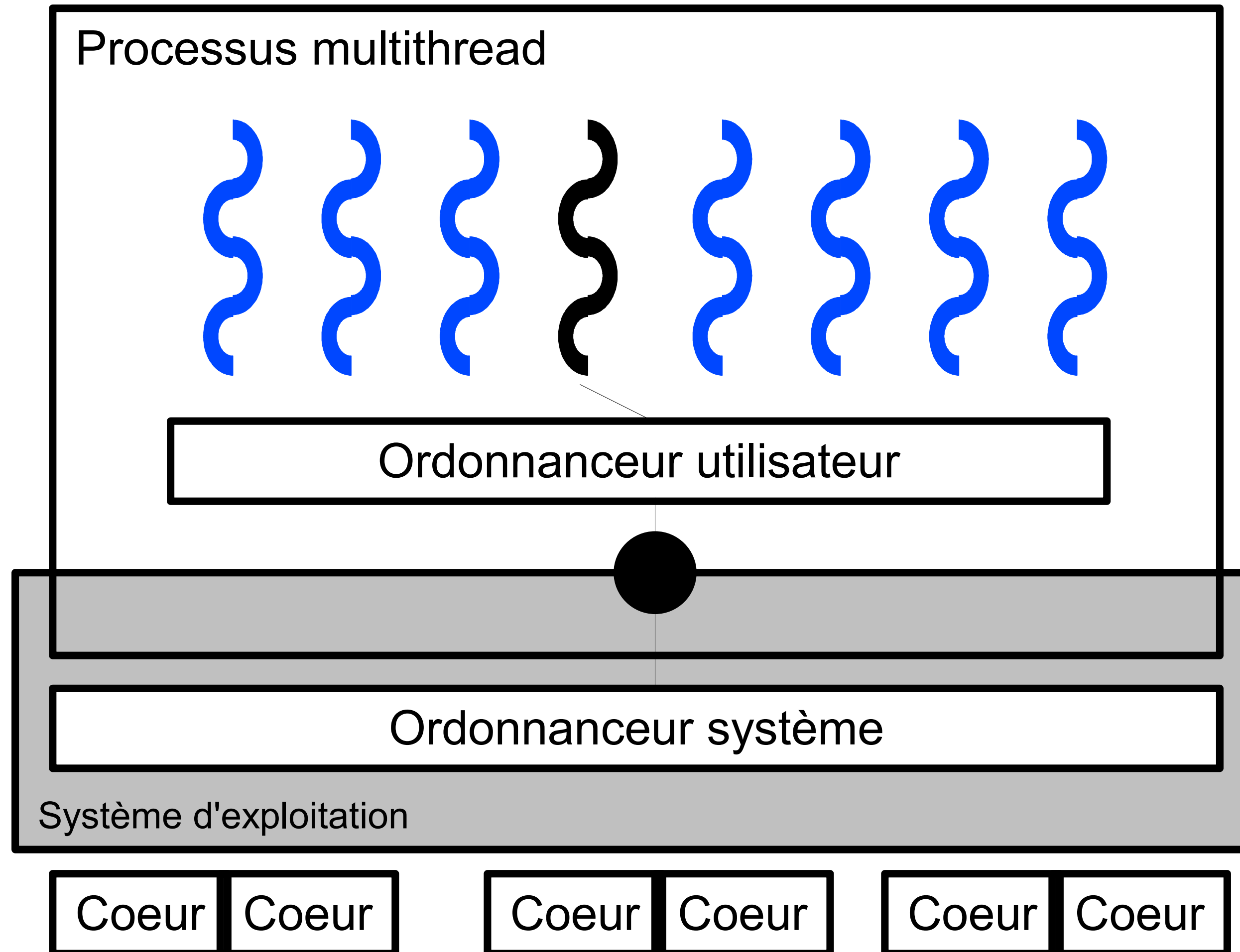
Les bibliothèques de threads

- Bibliothèque utilisateur:



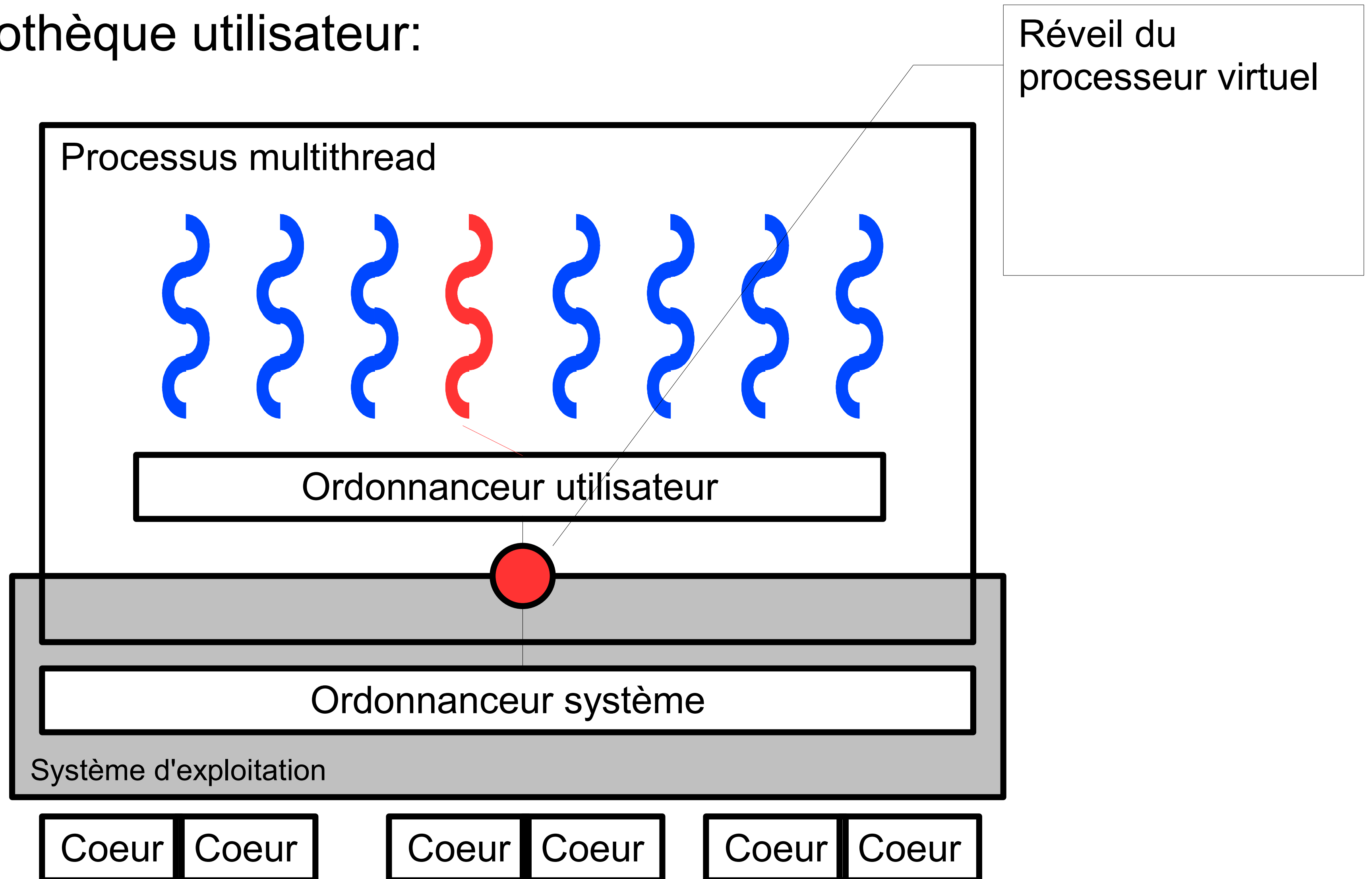
Les bibliothèques de threads

- Bibliothèque utilisateur:



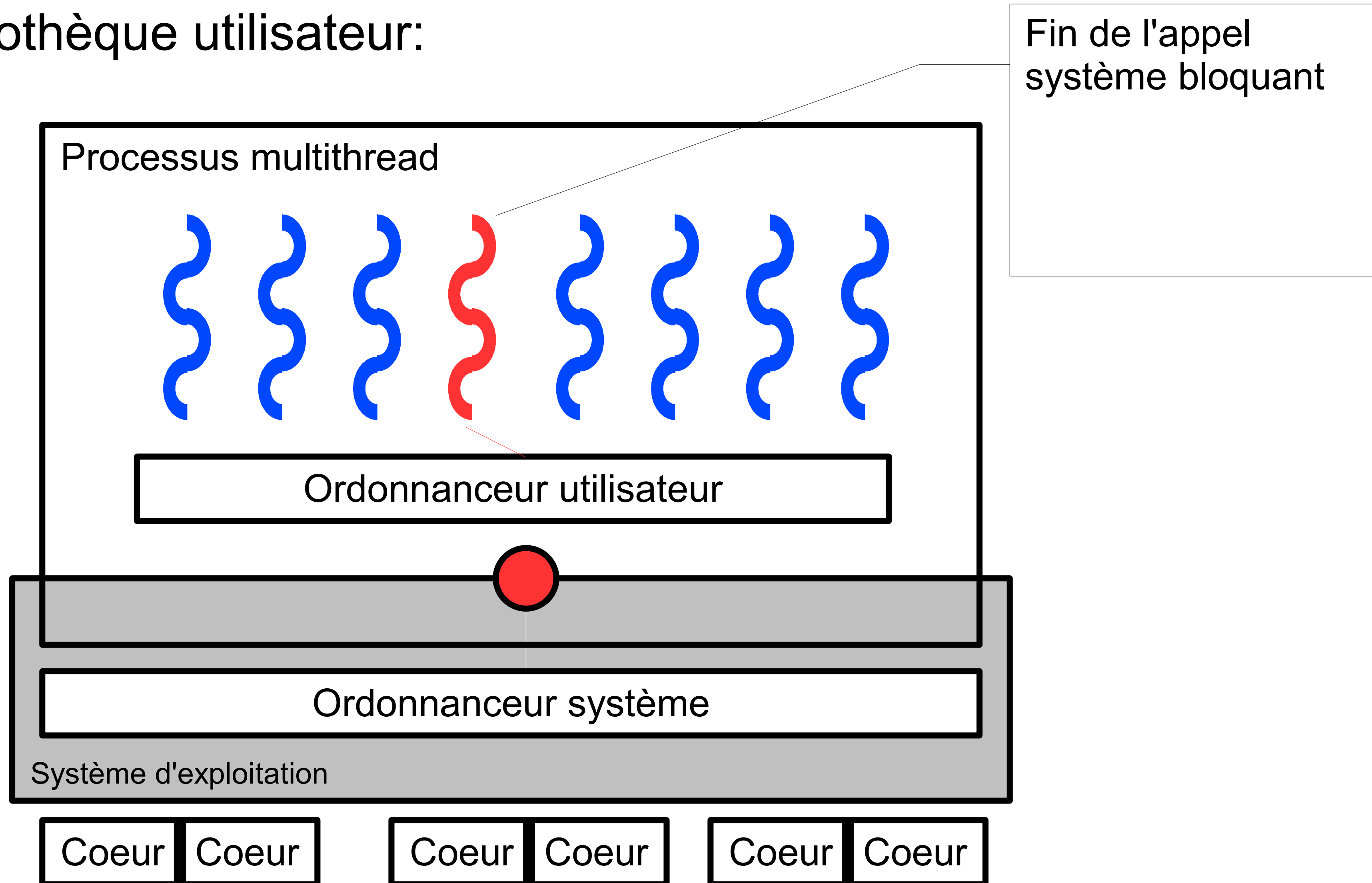
Les bibliothèques de threads

- Bibliothèque utilisateur:



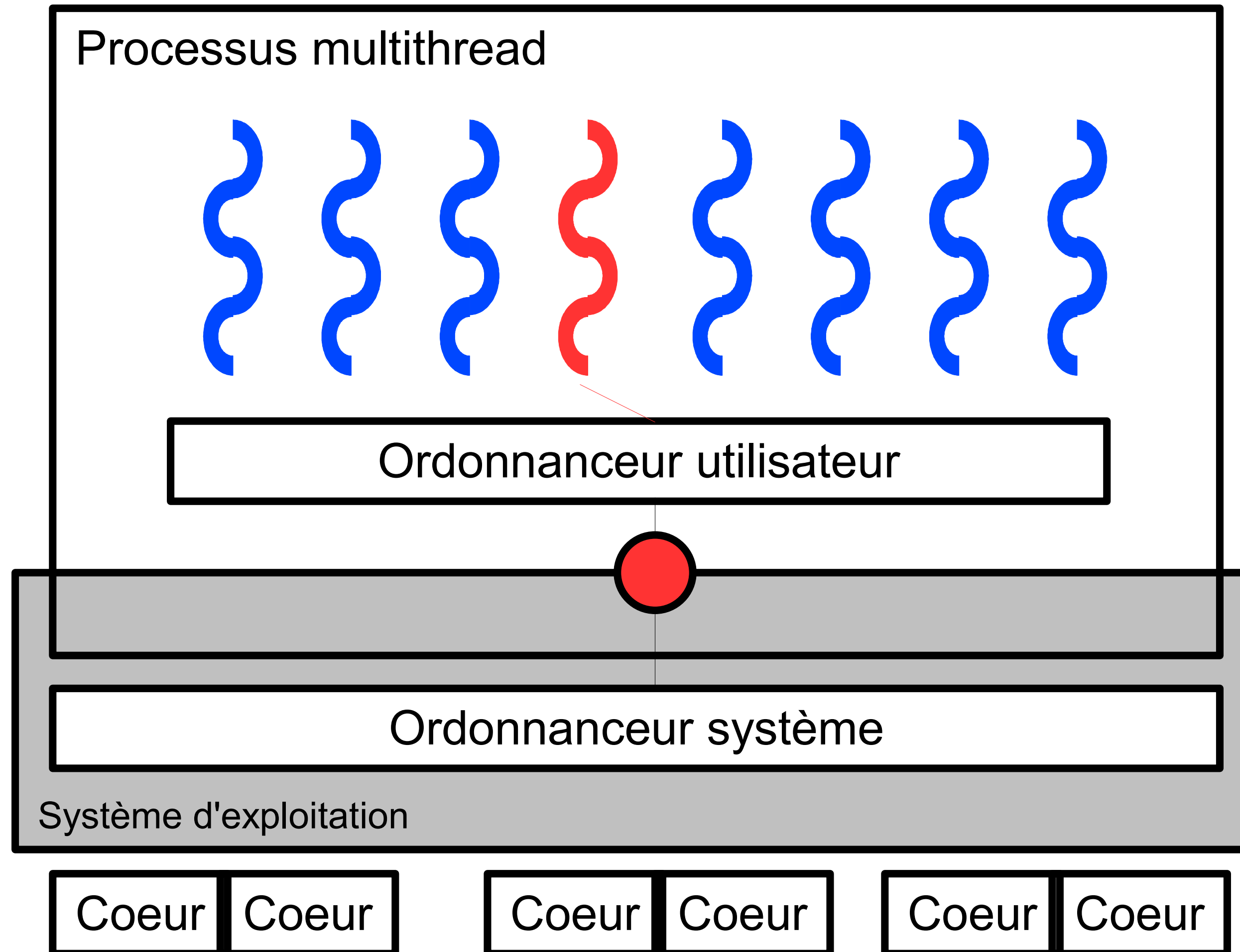
Les bibliothèques de threads

- Bibliothèque utilisateur:



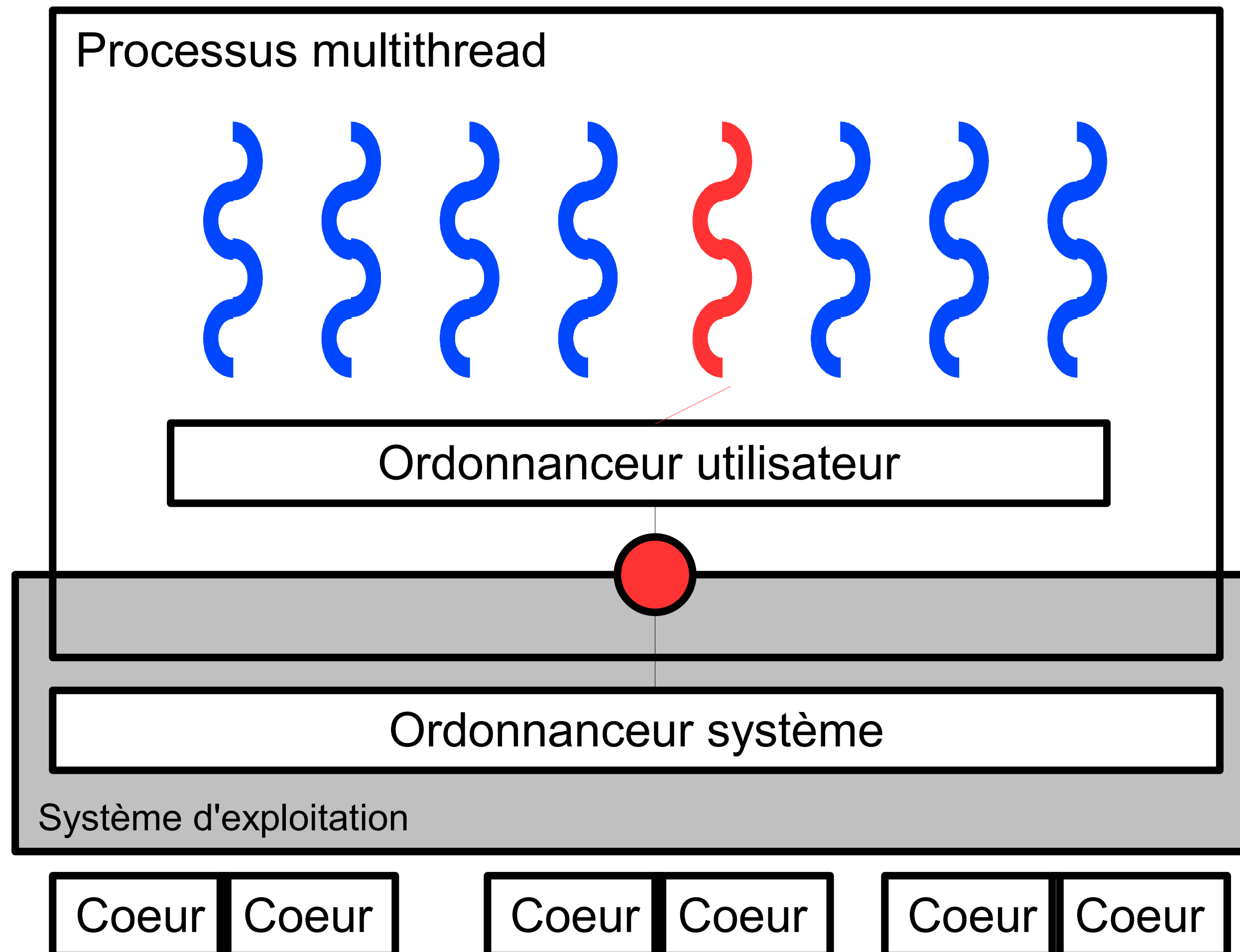
Les bibliothèques de threads

- Bibliothèque utilisateur:



Les bibliothèques de threads

- Bibliothèque utilisateur:

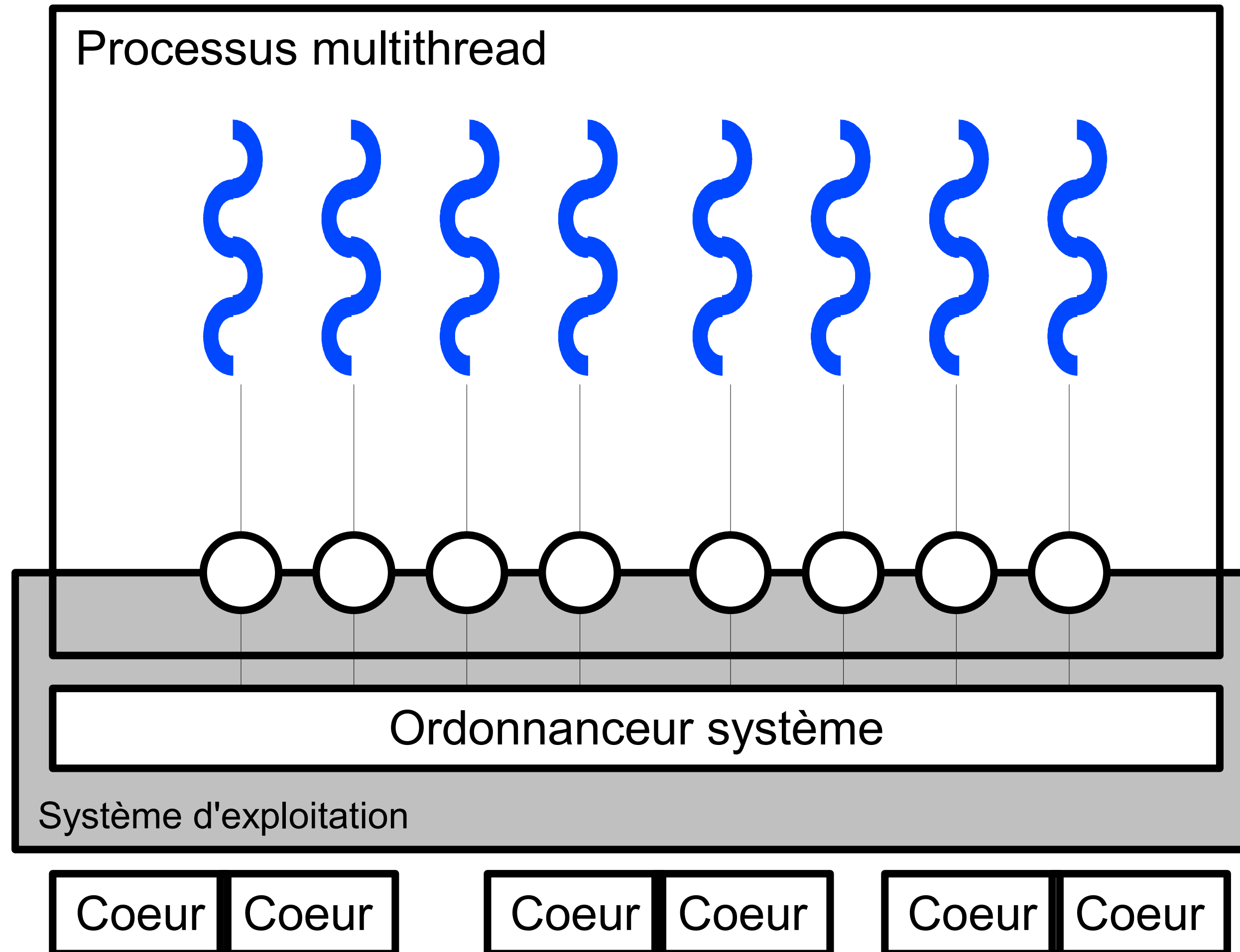


Les bibliothèques de threads

- Bibliothèque utilisateur:
  - 1 thread noyau par processus
  - Simple à implémenter.
  - Première bibliothèque de threads.
  - Pas adapté au SMP et multicoeur.
  - Problèmes avec les appels systèmes.
  - Entièrement en utilisateur.
  - Performante.
  - Ex: GNUPth

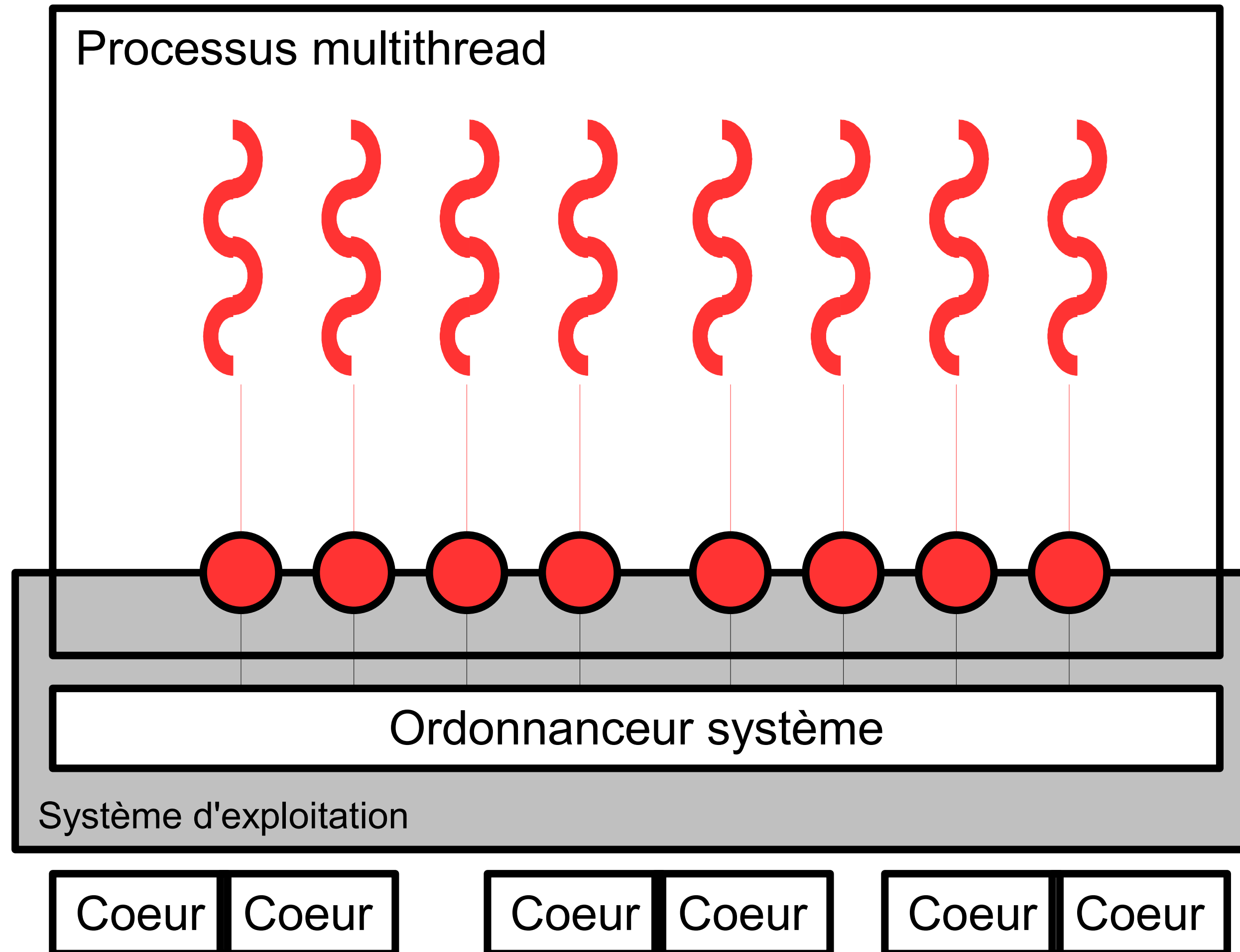
Les bibliothèques de threads

- Bibliothèque système:



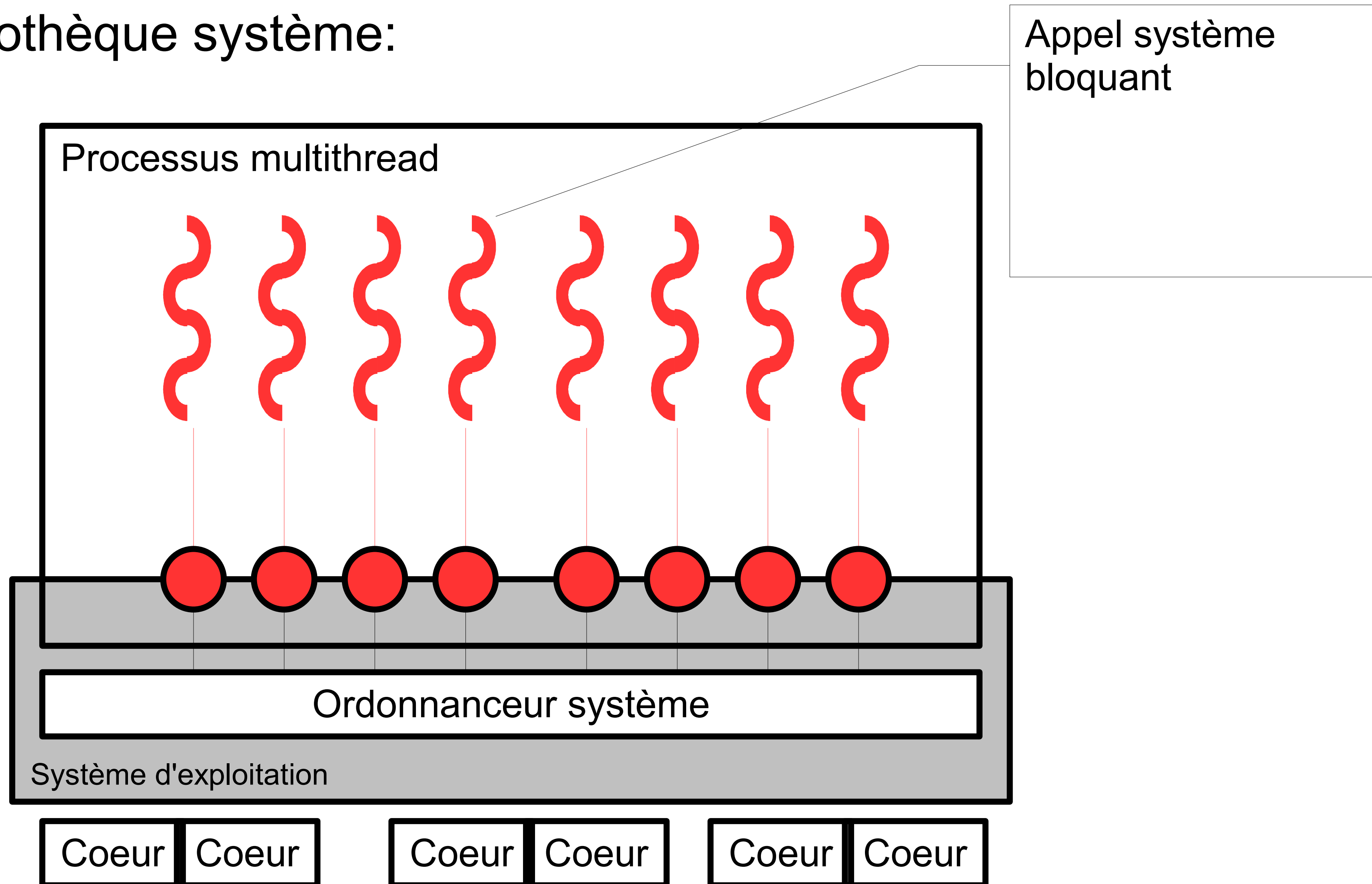
Les bibliothèques de threads

- Bibliothèque système:



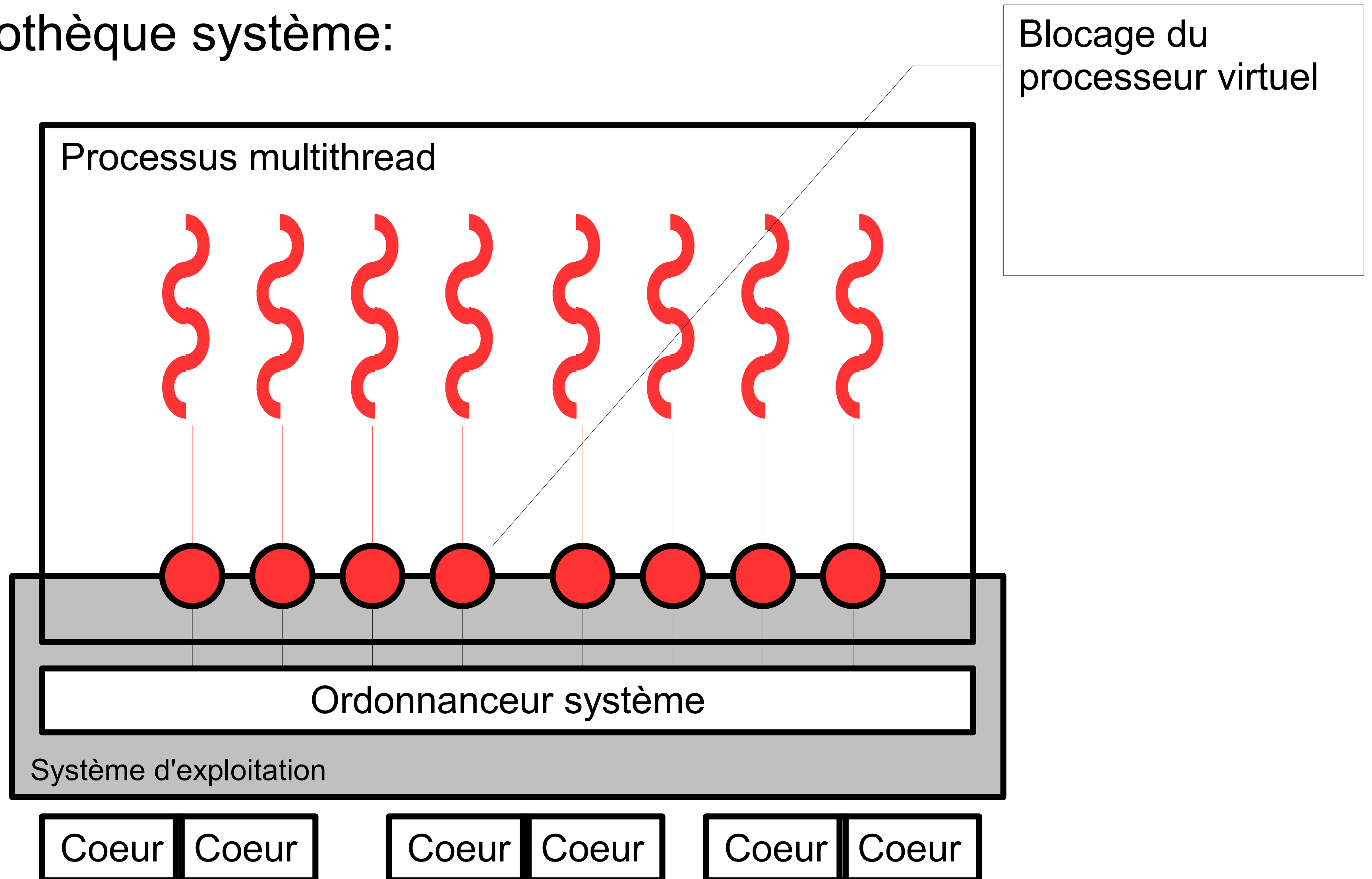
Les bibliothèques de threads

- Bibliothèque système:



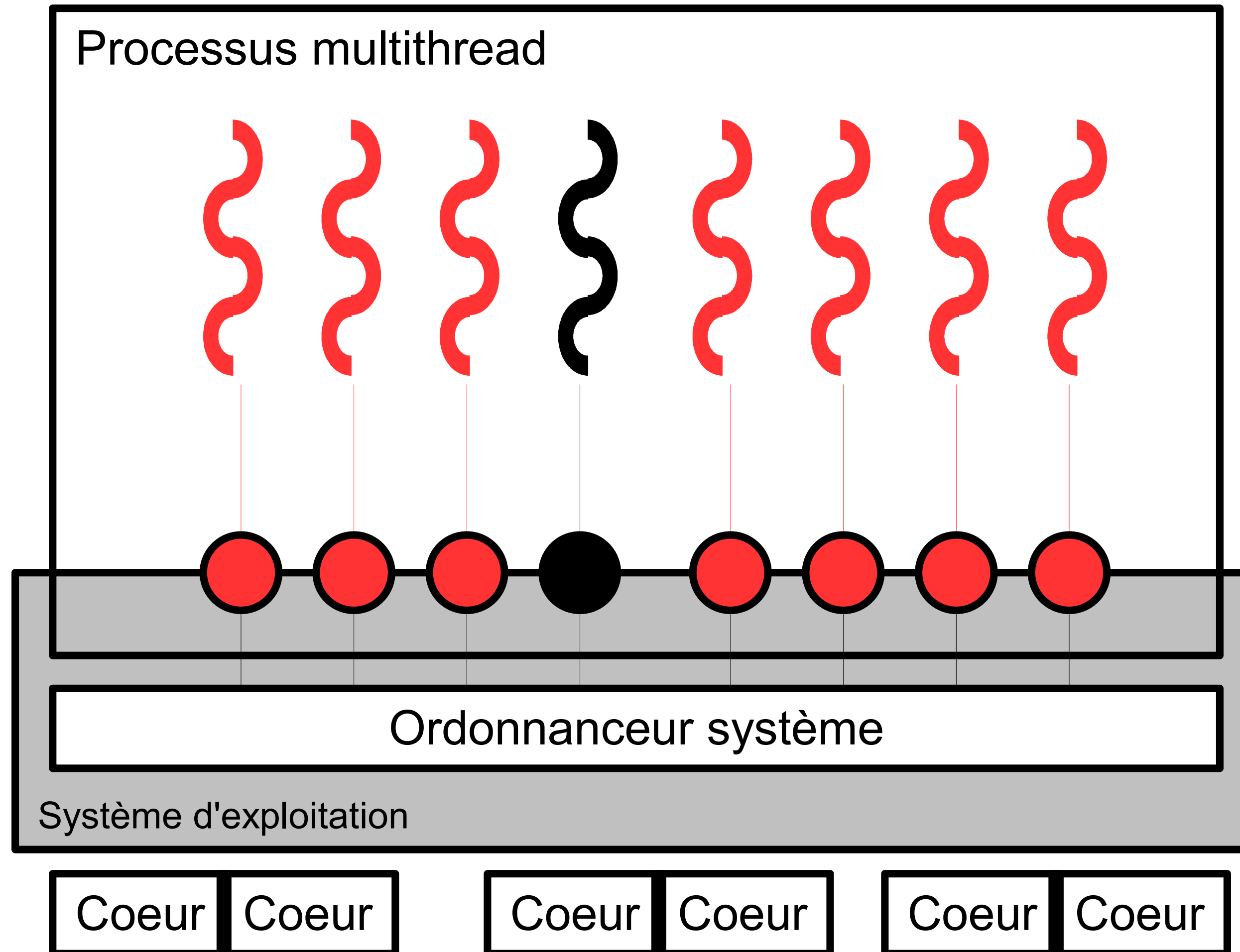
Les bibliothèques de threads

- Bibliothèque système:



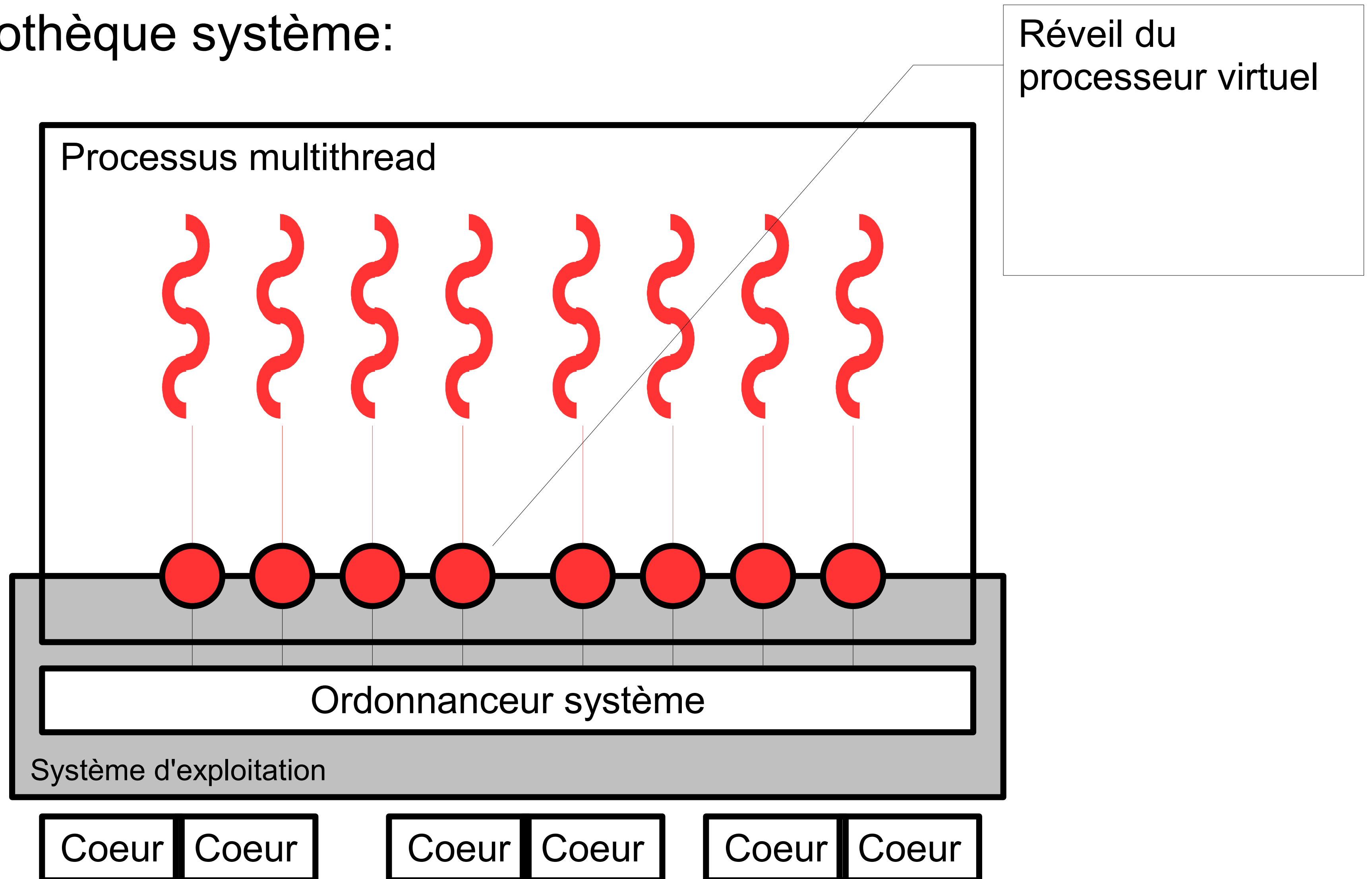
Les bibliothèques de threads

- Bibliothèque système:



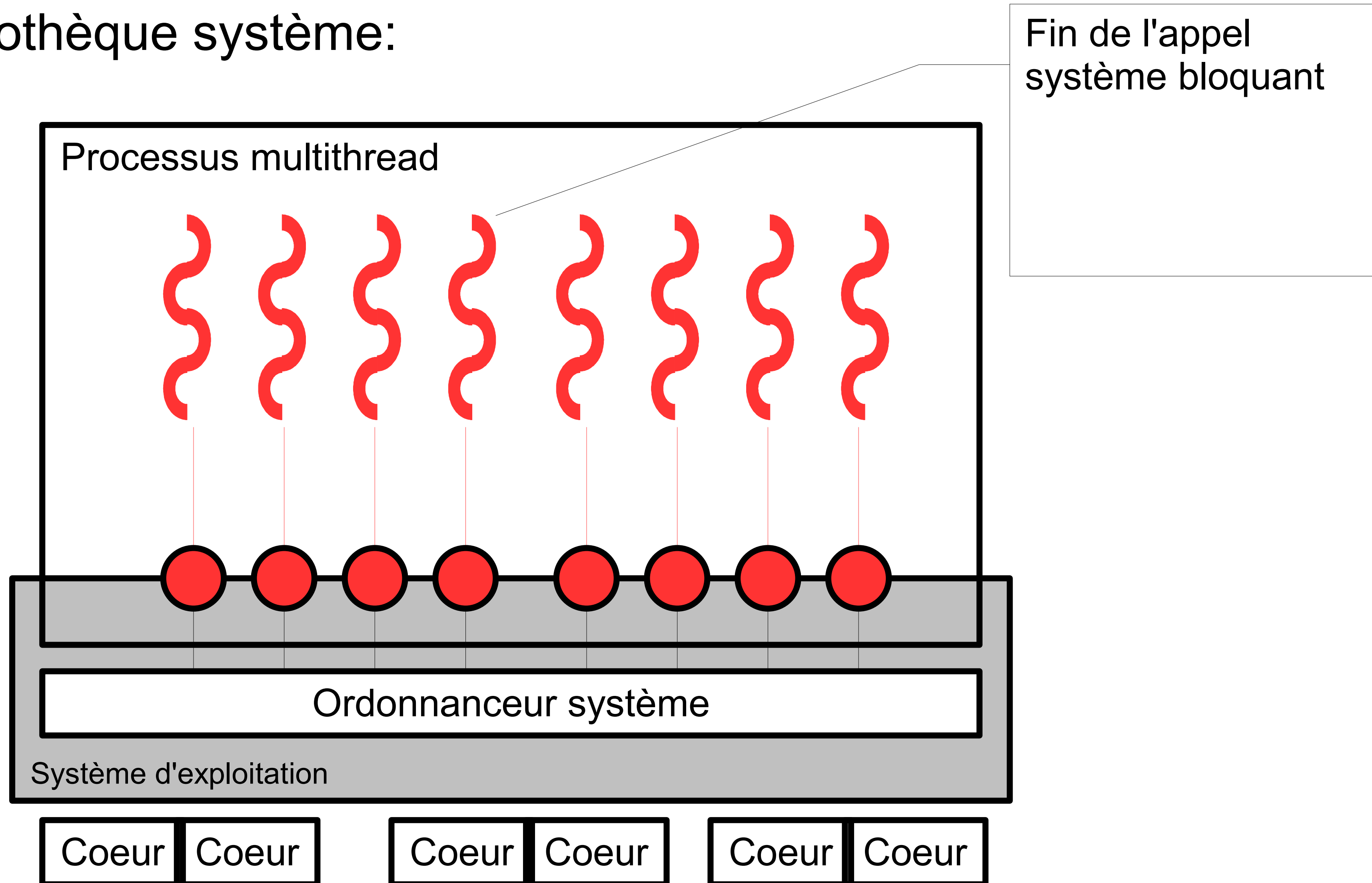
Les bibliothèques de threads

- Bibliothèque système:



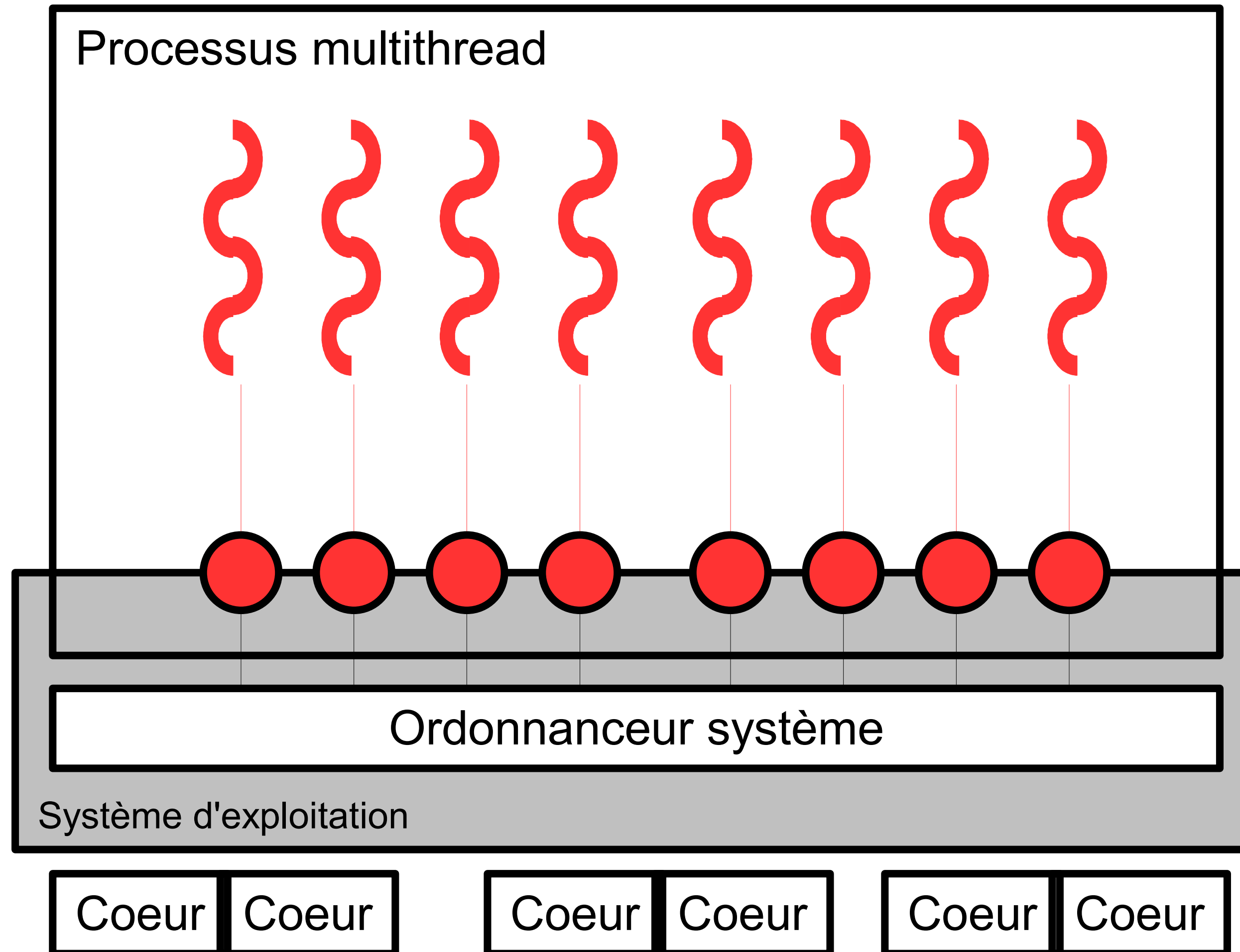
Les bibliothèques de threads

- Bibliothèque système:



Les bibliothèques de threads

- Bibliothèque système:

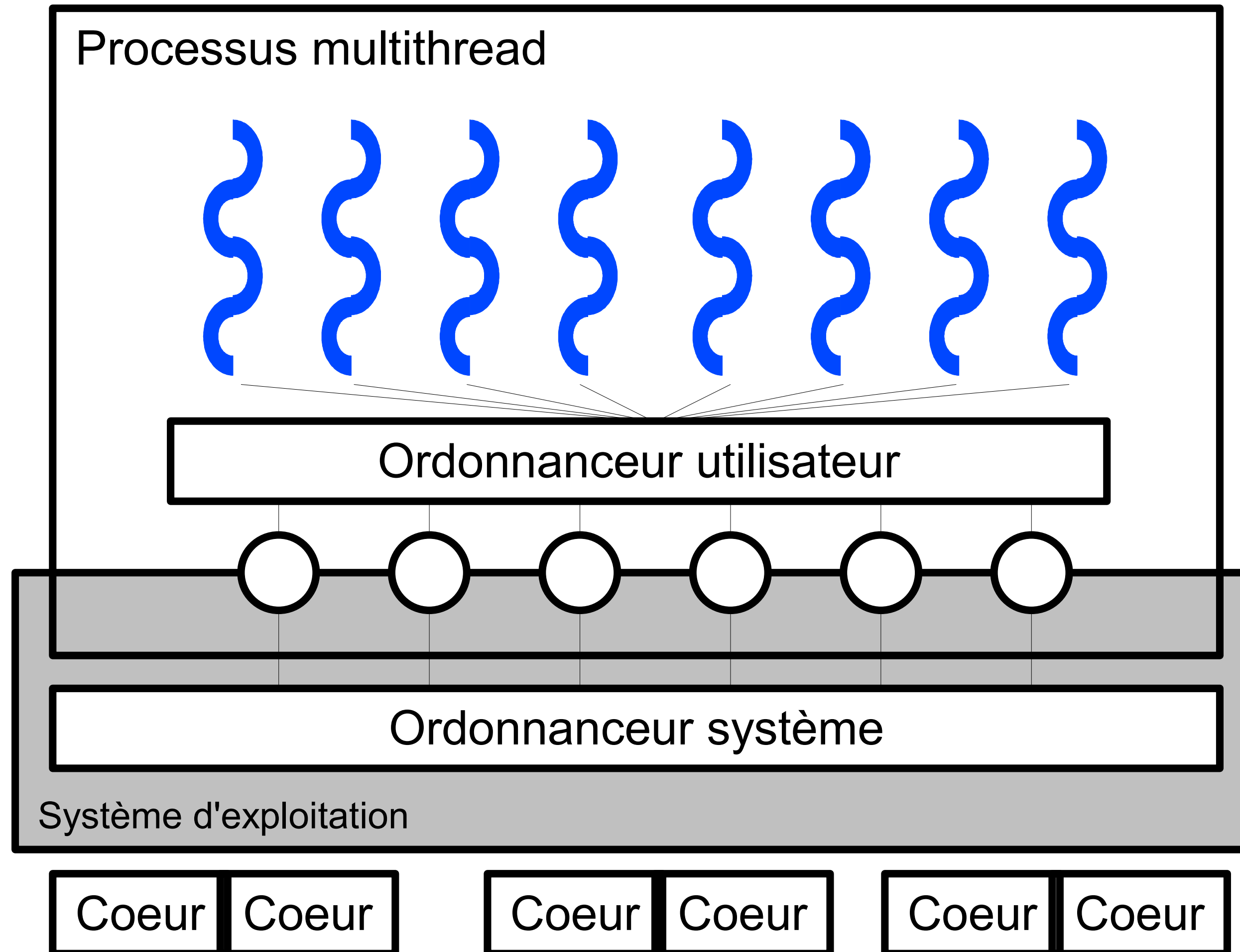


Les bibliothèques de threads

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

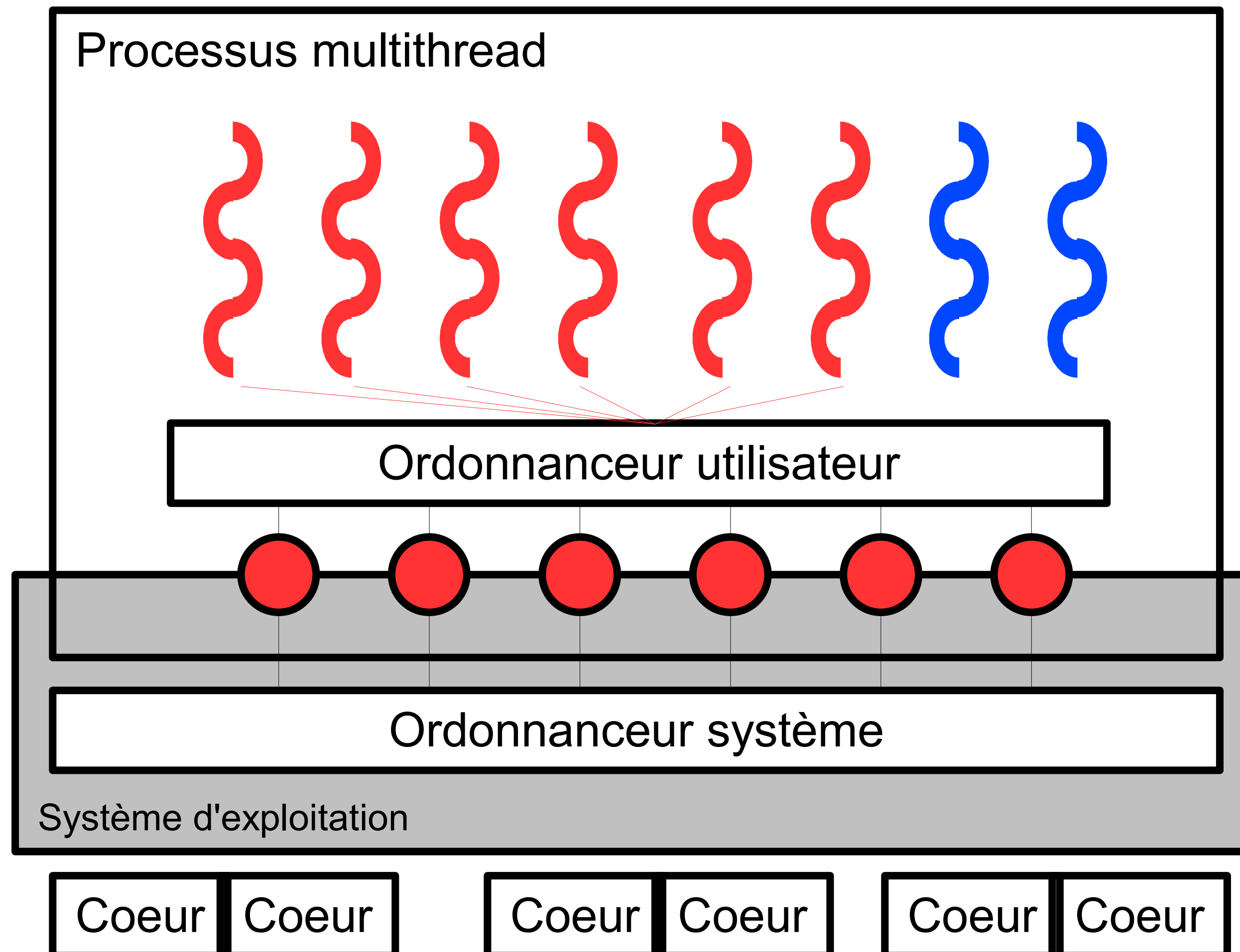
Les bibliothèques de threads

- Bibliothèque mixte (ou MxN):



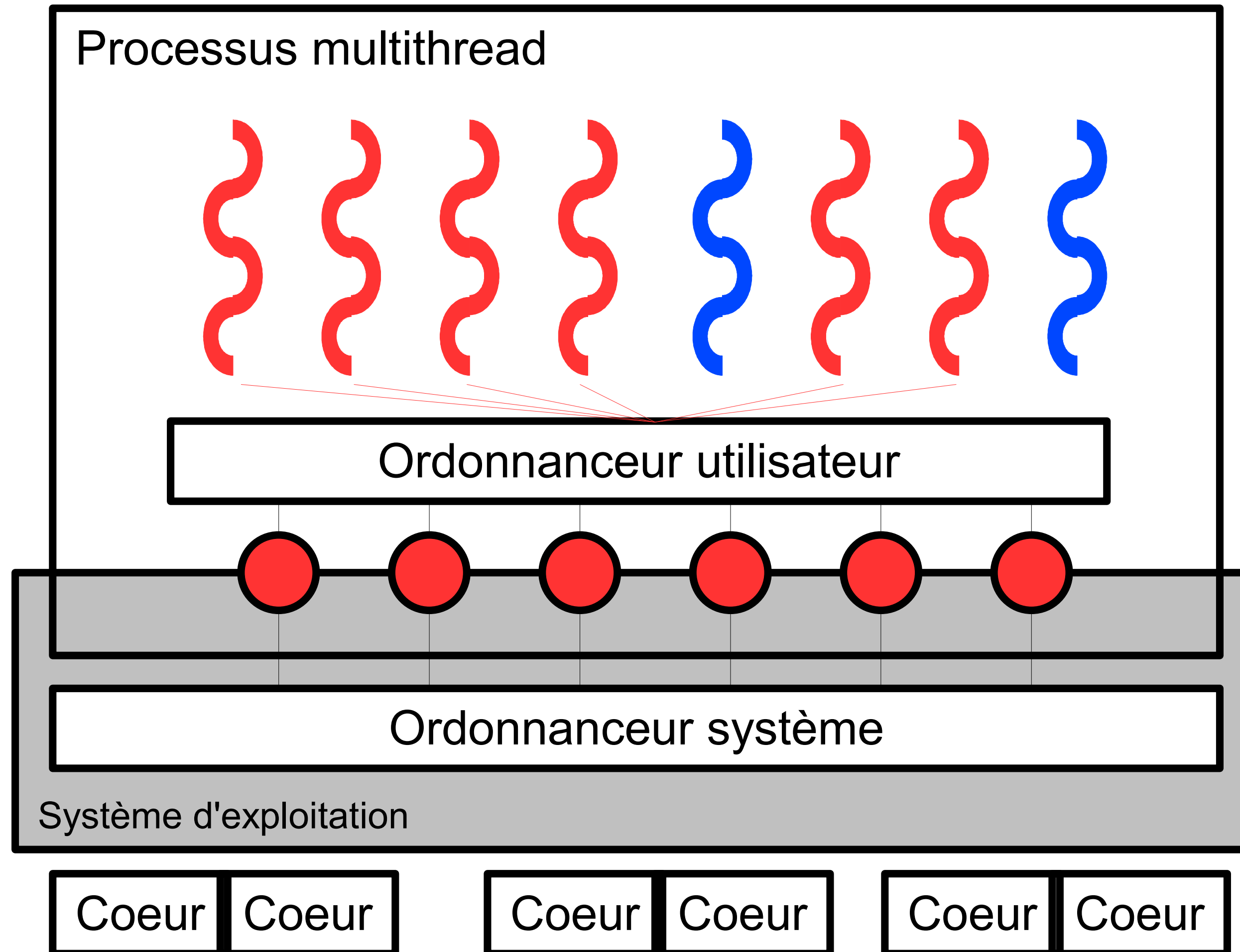
Les bibliothèques de threads

- Bibliothèque mixte (ou MxN):



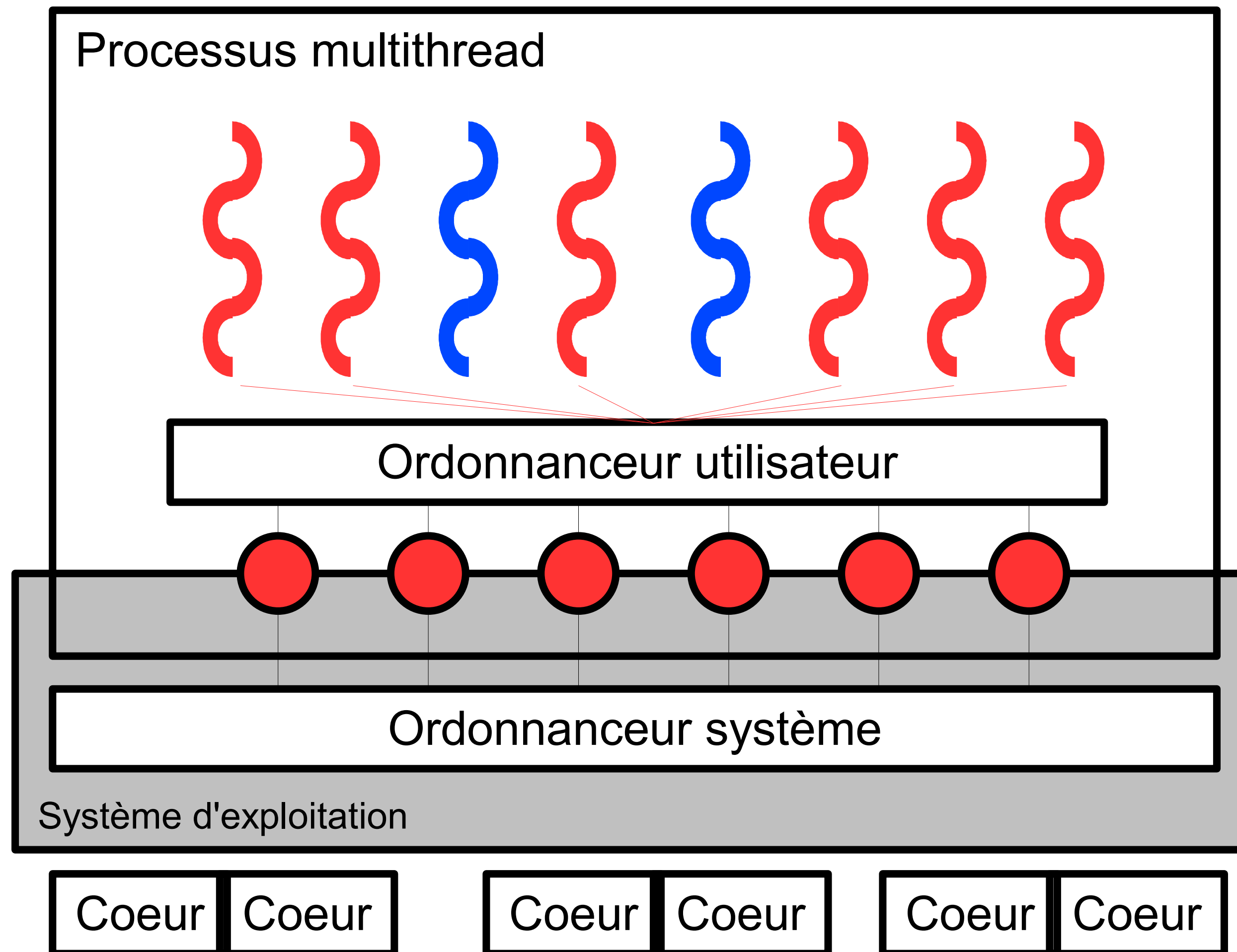
Les bibliothèques de threads

- Bibliothèque mixte (ou MxN):



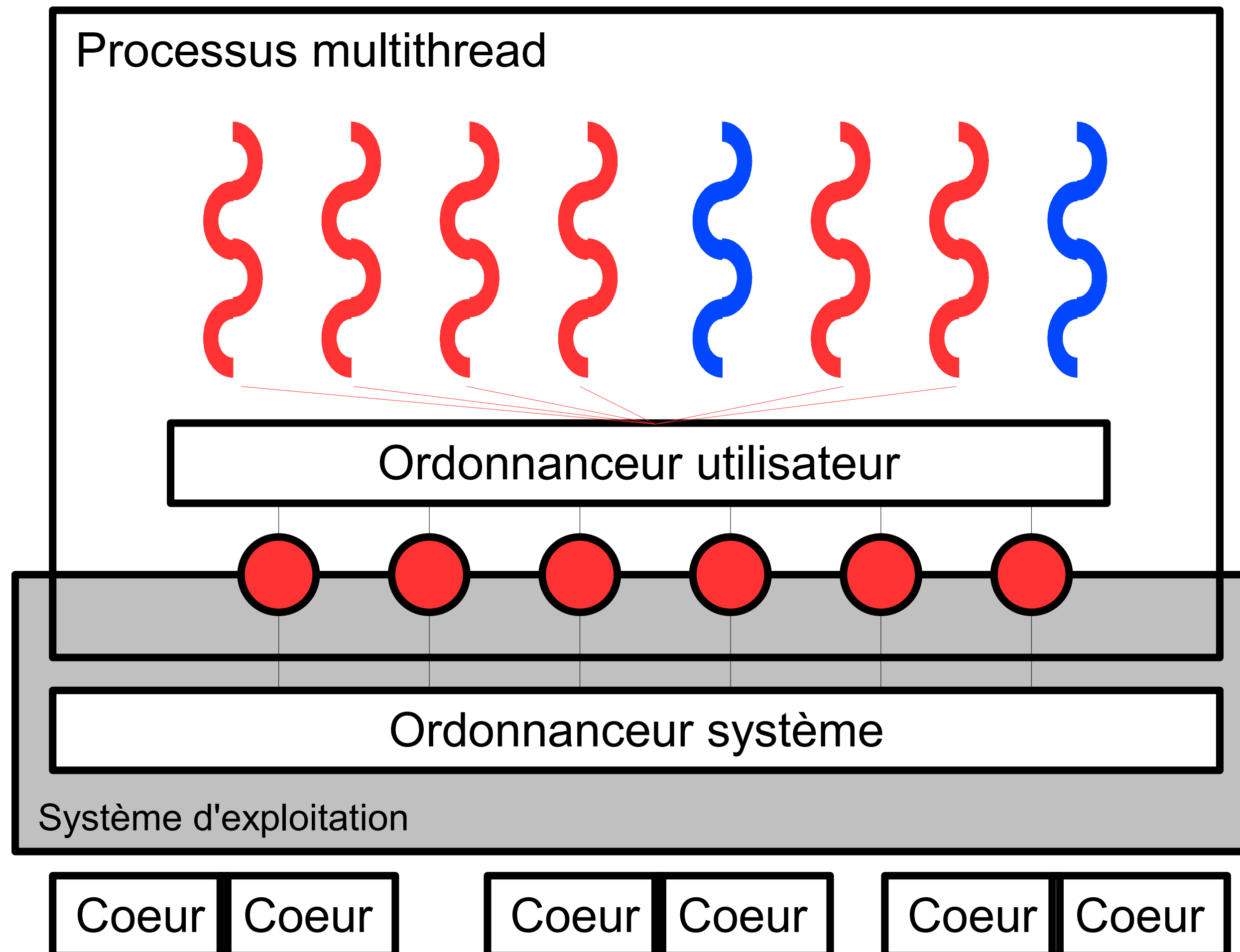
Les bibliothèques de threads

- Bibliothèque mixte (ou MxN):



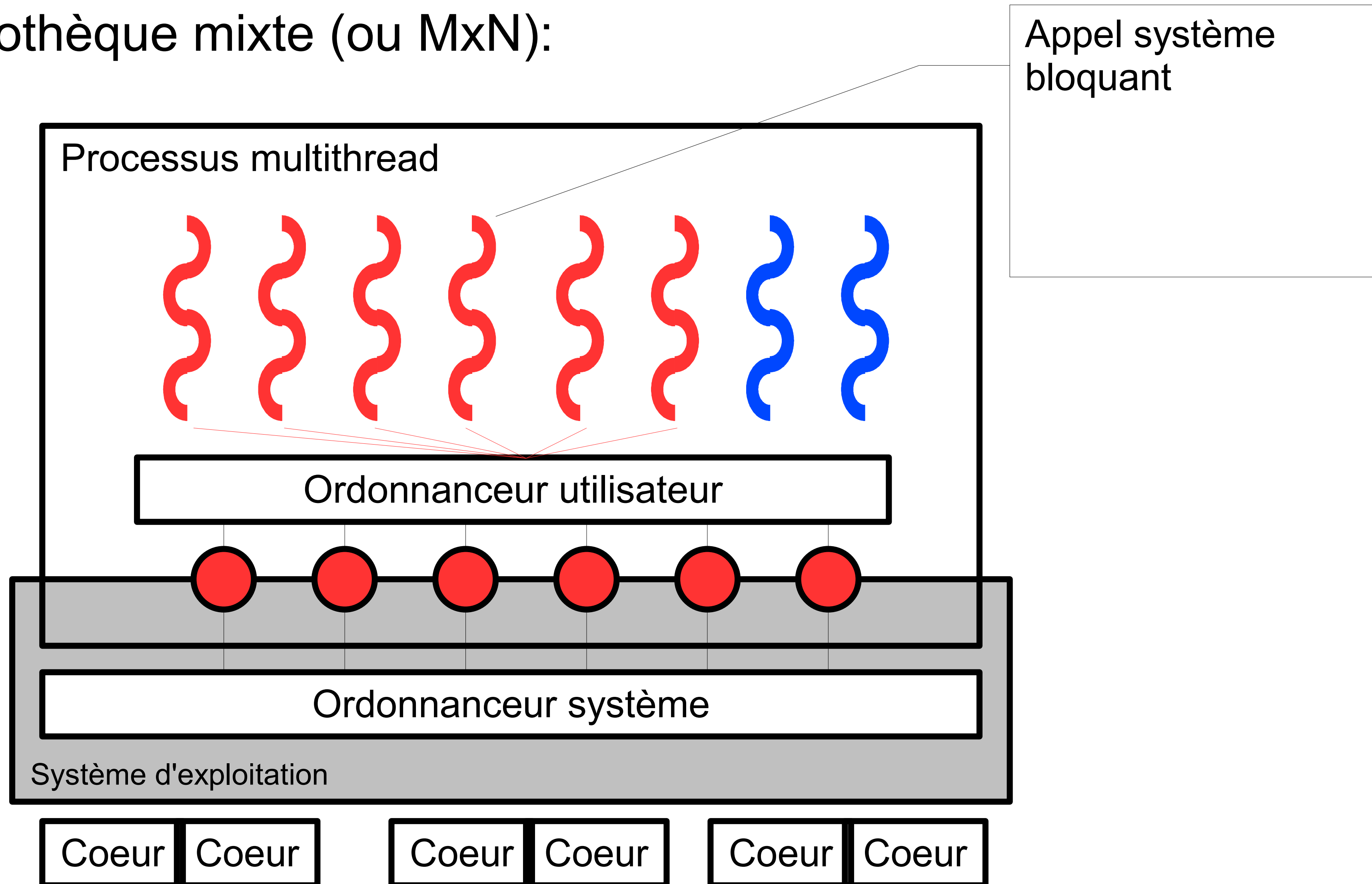
Les bibliothèques de threads

- Bibliothèque mixte (ou MxN):



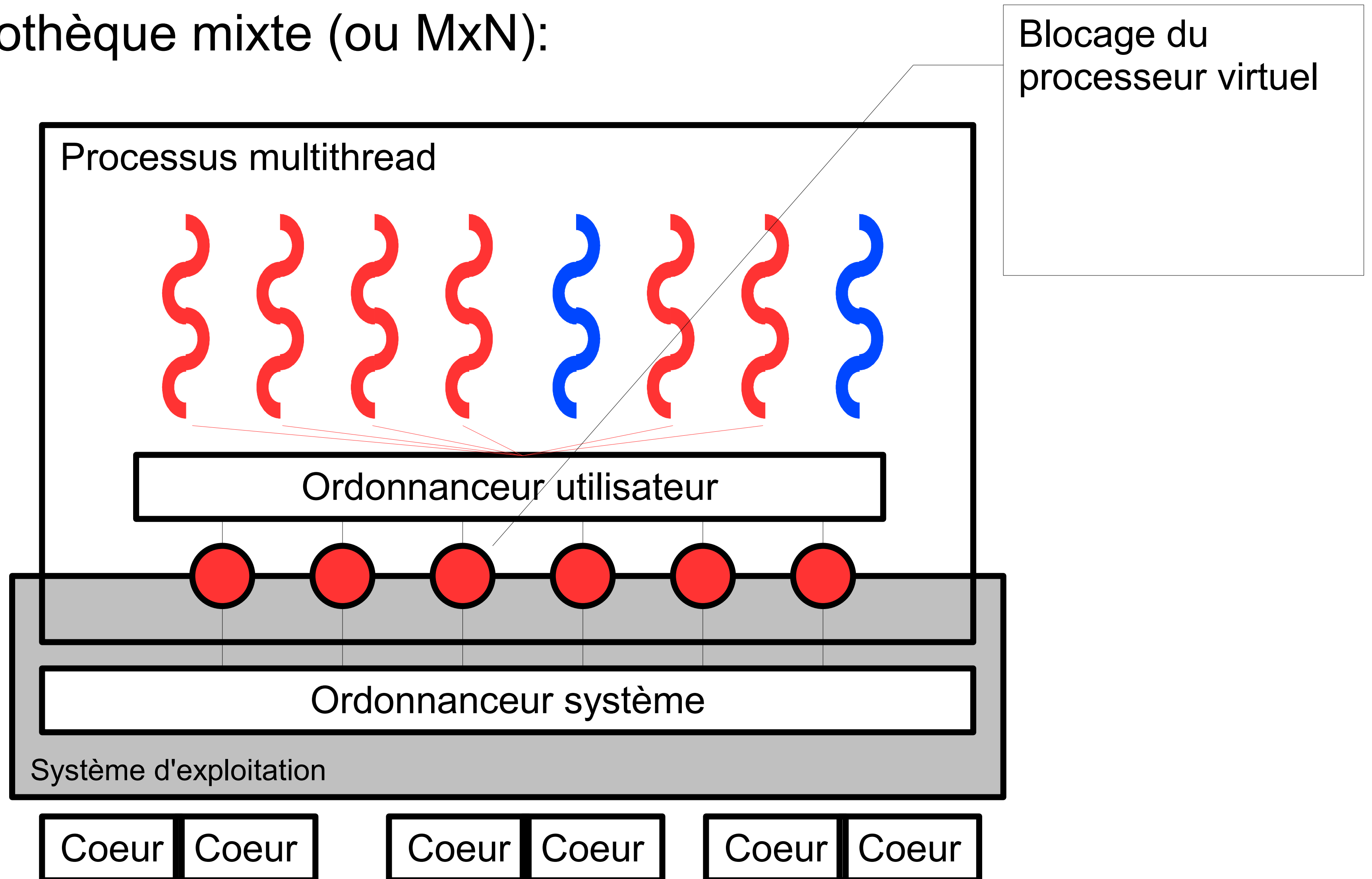
Les bibliothèques de threads

- Bibliothèque mixte (ou MxN):



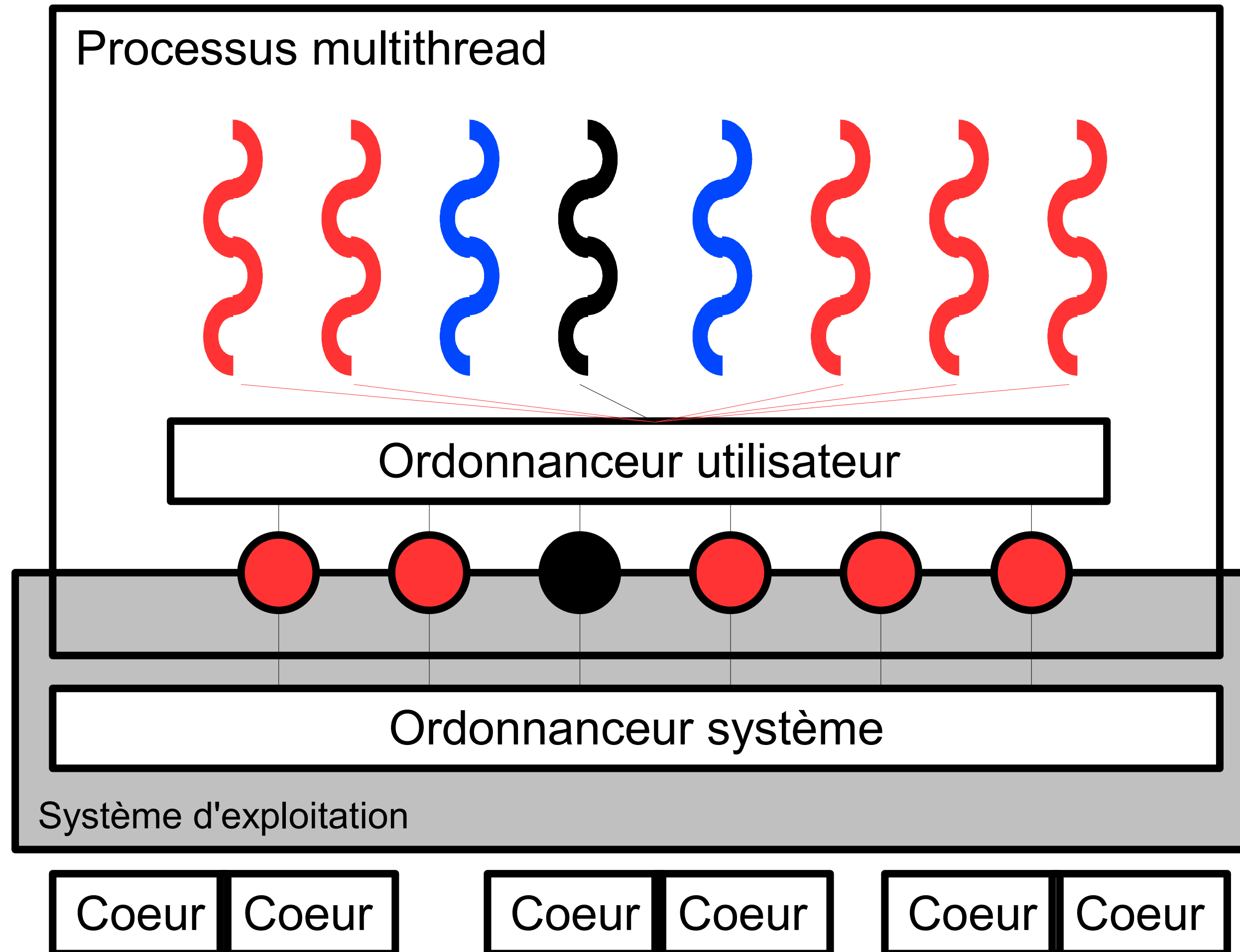
Les bibliothèques de threads

- Bibliothèque mixte (ou MxN):



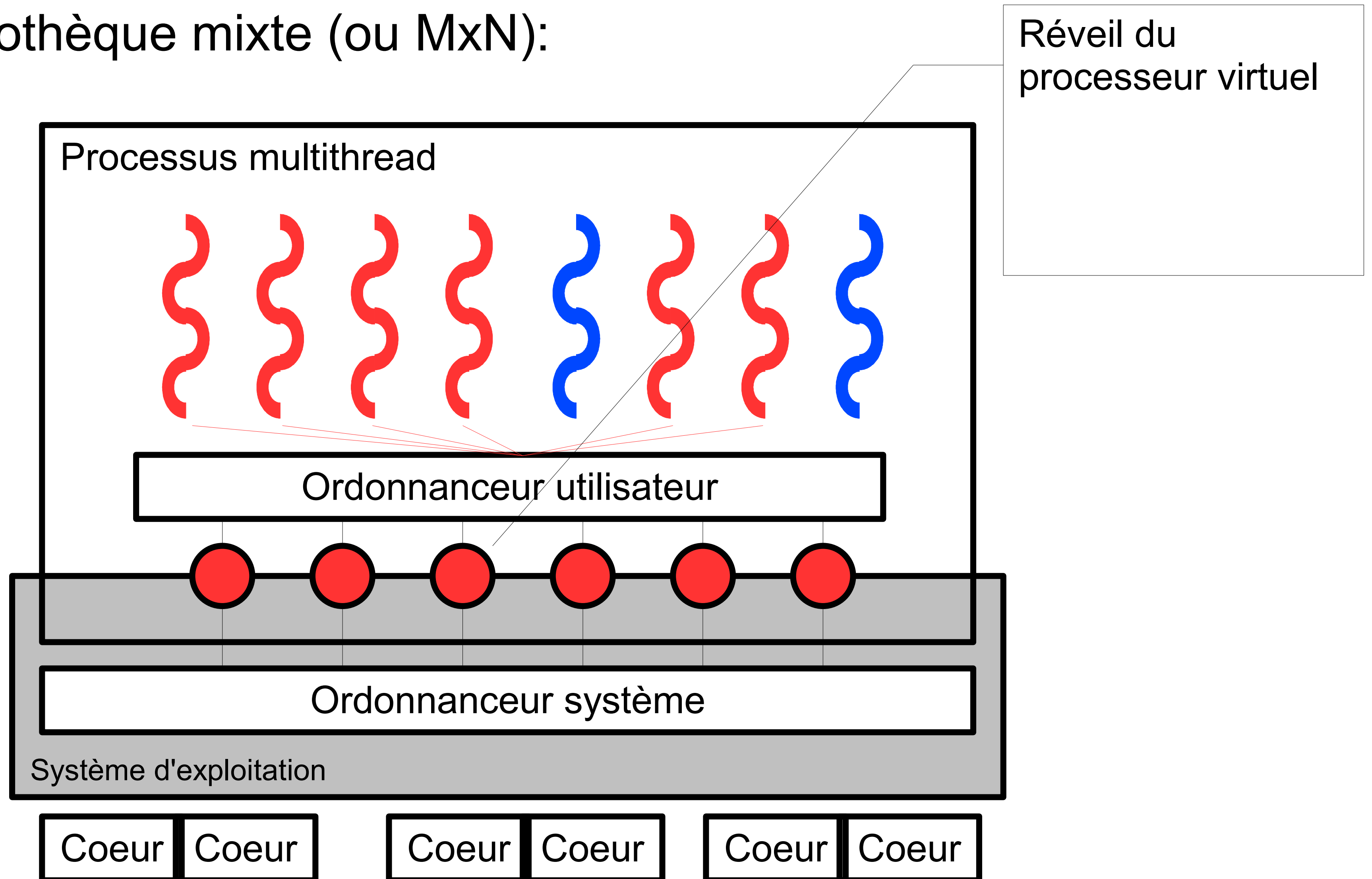
Les bibliothèques de threads

- Bibliothèque mixte (ou MxN):



Les bibliothèques de threads

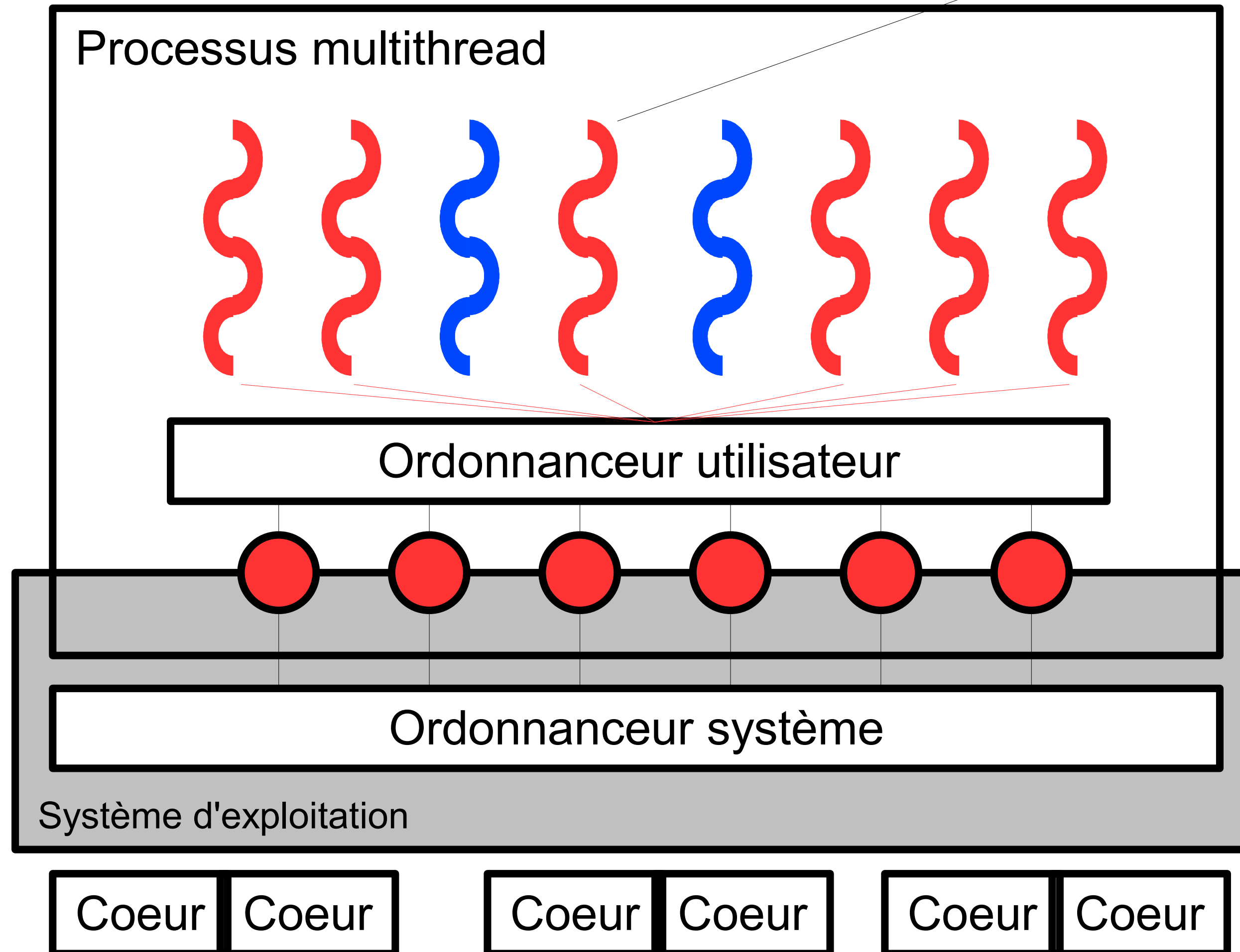
- Bibliothèque mixte (ou MxN):



Les bibliothèques de threads

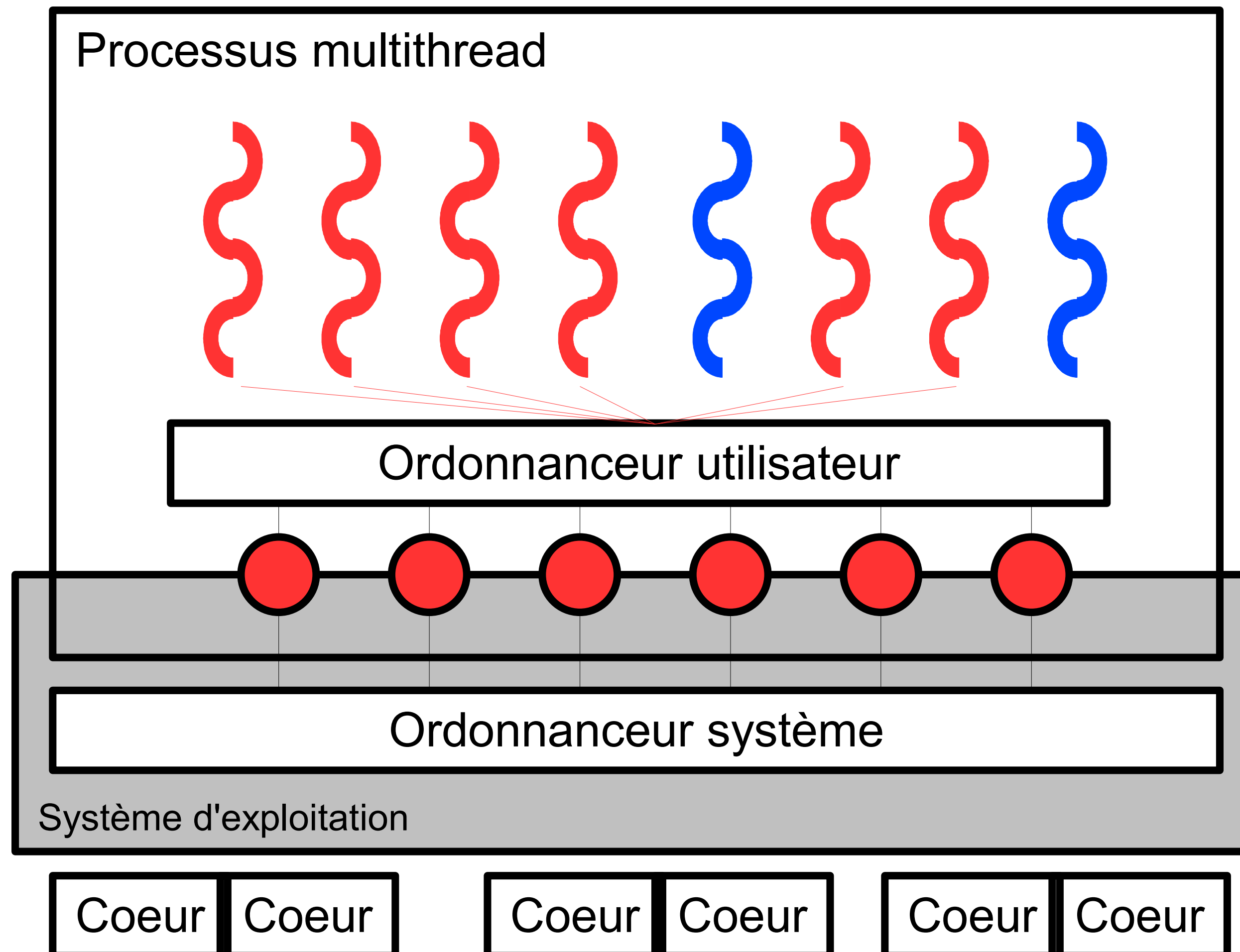
- Bibliothèque mixte (ou MxN):

Fin de l'appel système bloquant



Les bibliothèques de threads

- Bibliothèque mixte (ou MxN):



Les bibliothèques de threads

- Bibliothèque mixte:
  - M threads noyau pour N threads utilisateurs.
  - Les plus complexes à implémenter.
  - Adaptée aux SMP et multicoeur.
  - Deux odonnanceurs:
    - Un système.
    - Un utilisateur.
  - Performante.
  - Quelques problèmes avec les appels systèmes.
  - Ex: Solaris threads, mthread.

Tableau récapitulatif

	Caractéristiques			
Bibliothèque	Performances	Flexibilité	SMP	Appels systèmes bloquants
Utilisateur	+	+	-	-
Système	-	-	+	+
Mixte	+	+	+	Limités

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 distincte 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.

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.

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 condition variables (condvar) permettent de réveiller un thread endormis en fonction de la valeur d'une variable.

La bibliothèque mthread

- Bibliothèque de thread MxN.
- Déjà présent:
  - L'ordonnanceur.
  - Les spinlocks.
  - Les corps de fonctions.
- A compléter:
  - Tous les types de verrous.
  - Les Clés.

Notion clés

- **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.

Notion clés

int i = 0; int main(int argc, char** argv){	

i = i + argc;	movl %esp, %ebp
	movl i, %eax
	addl 8(%ebp), %eax

i++;	incl %eax
printf("%d",i); }	

Notion clés

- **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.

Notion clés

C	Assembleur
int i = 0; int main(int argc, char** argv){	

i = i + argc;	movl %esp, %ebp
	movl i, %eax
	addl 8(%ebp), %eax

i++;	incl %eax
printf("%d",i); }	

Notion clés

volatile int i = 0; int main(int argc, char** argv){	

i = i + argc;	movl %esp, %ebp
	movl i, %eax
	addl 8(%ebp), %eax
	movl %eax, i

i++;	movl i, %eax
	incl %eax
	movl %eax, i

printf("%d",i); }	

L'API POSIX

- `pthread_create( thread, attribut, routine, argument )` Création d'un thread. Le nouveau flot d'exécution démarre en se branchant à la routine spécifiée. Cette routine reçoit l'argument prévu.
- `pthread_exit( résultat )` Suicide d'un thread.
- `pthread_join( thread, résultat )` Attendre la terminaison d'un autre thread.

L'API POSIX

- `pthread_kill( thread, nu_du_signal )` Envoyer un signal (UNIX) à un thread. C’est un moyen dur pour tuer un thread. Il existe des méthodes plus conviviales.
- `sched_yield()`ou `pthread_yield()` Abandonner le CPU pour le donner à un autre thread (ou un autre processus).
- `pthread_self()` Renvoie l'identifiant d'un thread.

Le premier code multithread

```
#include <pthread.h>

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

Synchronisations

- Problème du producteur/consommateur:
  - Le problème est le suivant: le programme est constitué de deux threads; le premier lit les caractères au clavier et le second se charge de les afficher.
  - Il faut noter que 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".

Synchronisations

- Comment faire des synchronisations entre threads ?

Synchronisations

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

Synchronisations

- 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 ?

Synchronisations

- 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.

Synchronisations

```
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;
}
```

Synchronisations

```
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);
}
```

Synchronisations

- 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

```
bash-3.0: 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!

Synchronisations

- Pourquoi ?

Synchronisations

- Pourquoi ?
- La boucle *while* du thread qui affiche consomme tout le temps CPU car le thread ne passe jamais la main aux autres threads.

Synchronisations

- 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 ?

Synchronisations

- 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) ; /* attendre */
```

par

```
while (afficher == 0) sched_yield(); /* attendre */
```

Synchronisations

- Ainsi le système redevient réactif, mais les applications sont tout de même fortement perturbées par notre application car la priorité de la tâche est très haute.
- Pour solutionner totalement le problème, il faut plutôt faire la modification en utilisant usleep

```
while (afficher == 0) ; /* attendre */
```

par

```
while (afficher == 0) usleep(5); /* attendre */
```

Synchronisations

- La commande ps -AeLf donne

```
bash-3.0: 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.