



# Programmation Parallèle et Distribuée

---

Cours 6 : Programmation *multithreads*

Patrick Carribault

David Dureau

Marc Pérache (marc.perache@cea.fr)



# Introduction

---

- Modèles de programmation parallèle
  - Mémoire distribuée
  - Mémoire partagée
- Mémoire distribuée
  - Vision séparée de la mémoire
  - Bibliothèque MPI
  - Cf. cours précédents
- Mémoire partagée
  - Vision unifiée de la mémoire (espace d'adressage unique)
  - Utilisation de threads



# Plan du cours 6

---

- Introduction aux threads
  - Notions & définitions
  - Présentation de l'API POSIX
  - Premiers exemples
- Modèle d'exécution
  - Types de bibliothèques de threads
- Fonctionnalités
  - Atomicité, volatilité, synchronisation



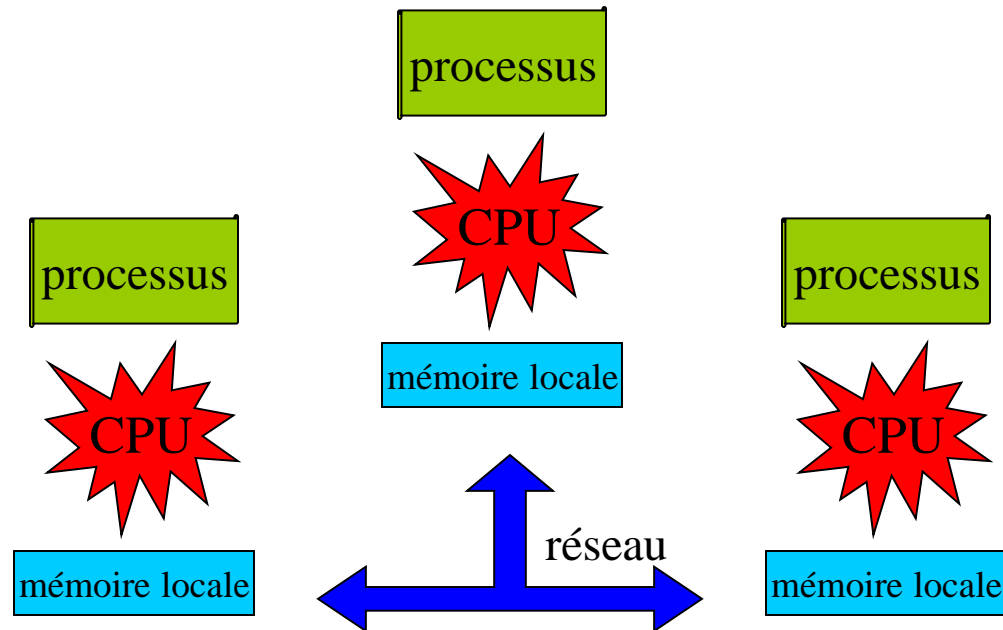
# Introduction aux threads

---



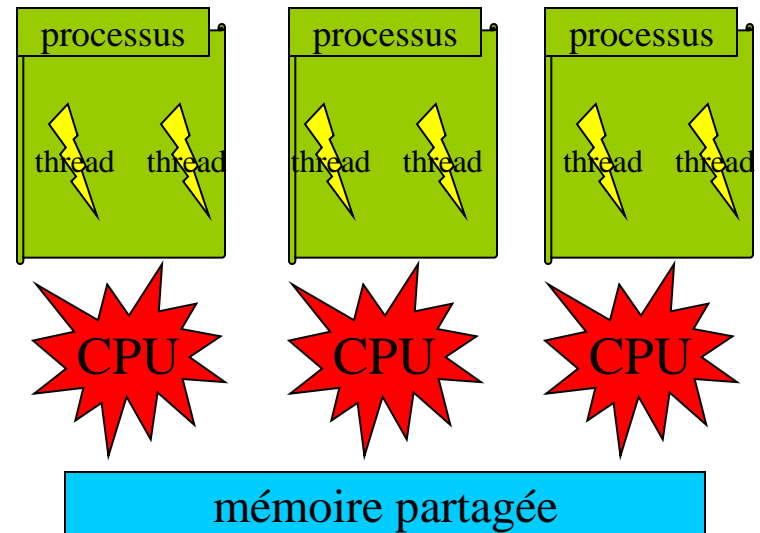
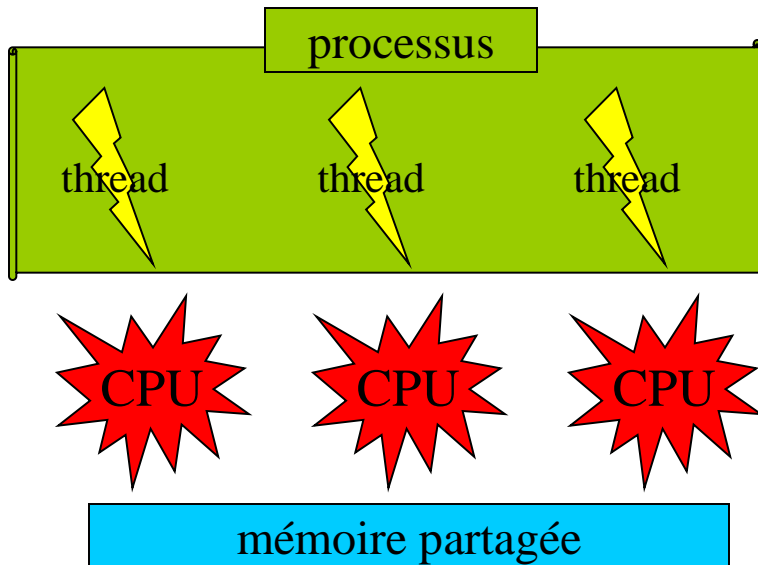
# Rappel : Système à mémoire distribuée

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



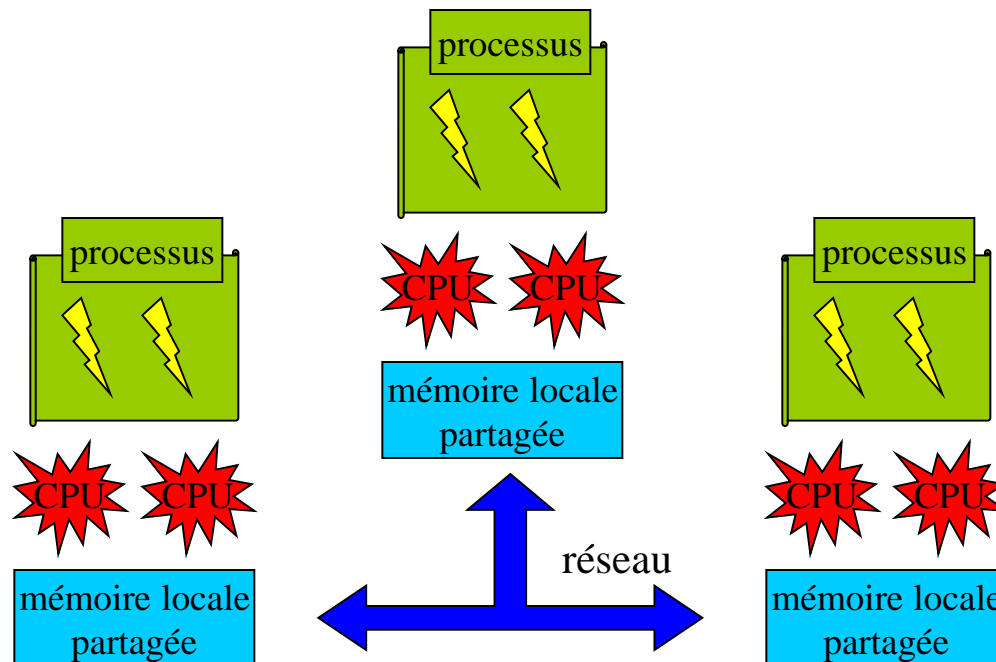
# Rappel : Système à mémoire partagée

- 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.
  - Noeuds de calcul SMP et NUMA.



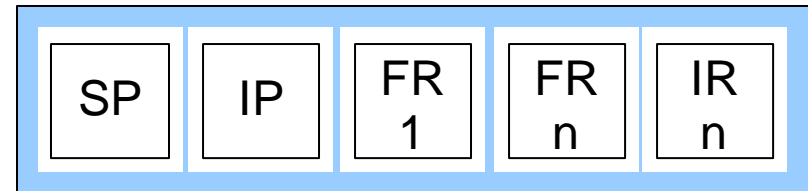
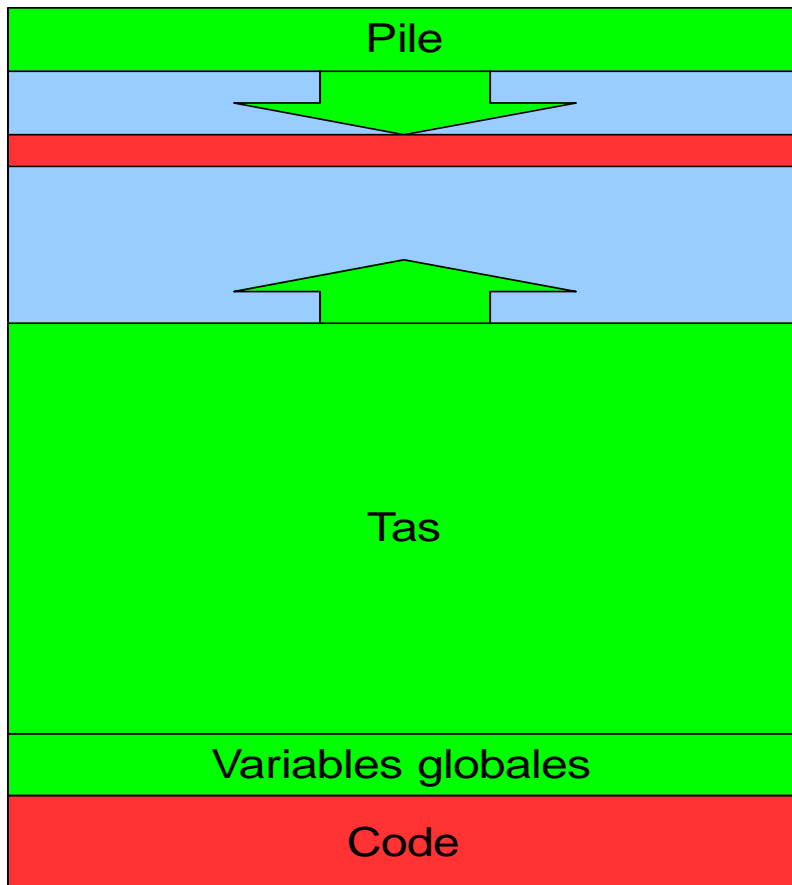


# Définition d'un thread

---

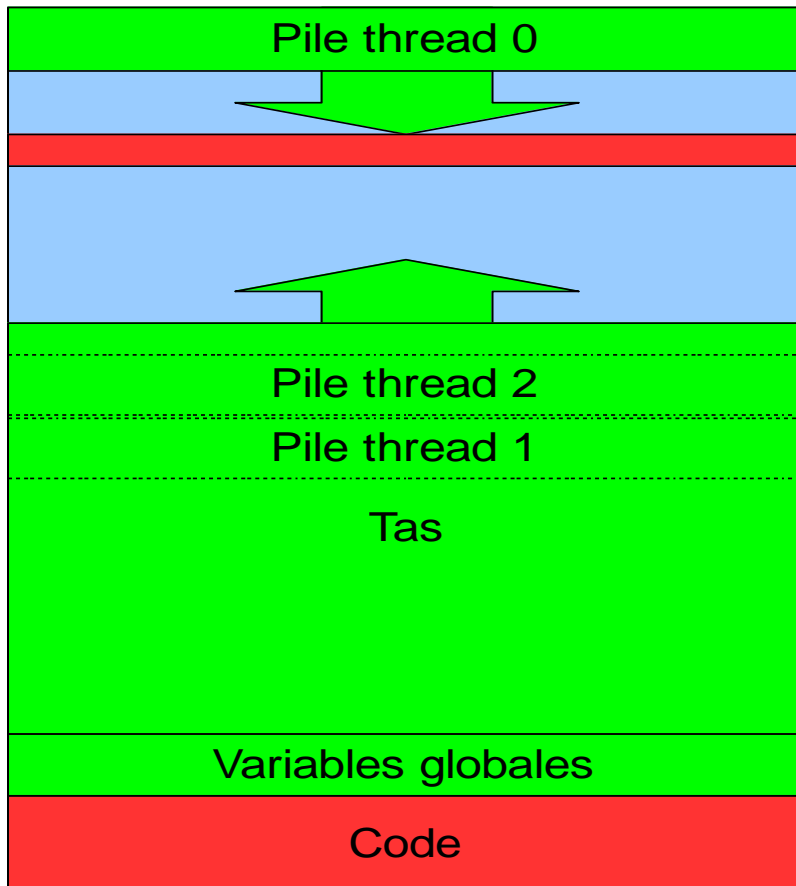
- Thread = processus léger
- Éléments d'un thread
  - Une pile
  - Un contexte : ensembles de registres
- Éléments d'un processus multithread
  - Une table de pages
  - Un ensemble de threads

# Processus

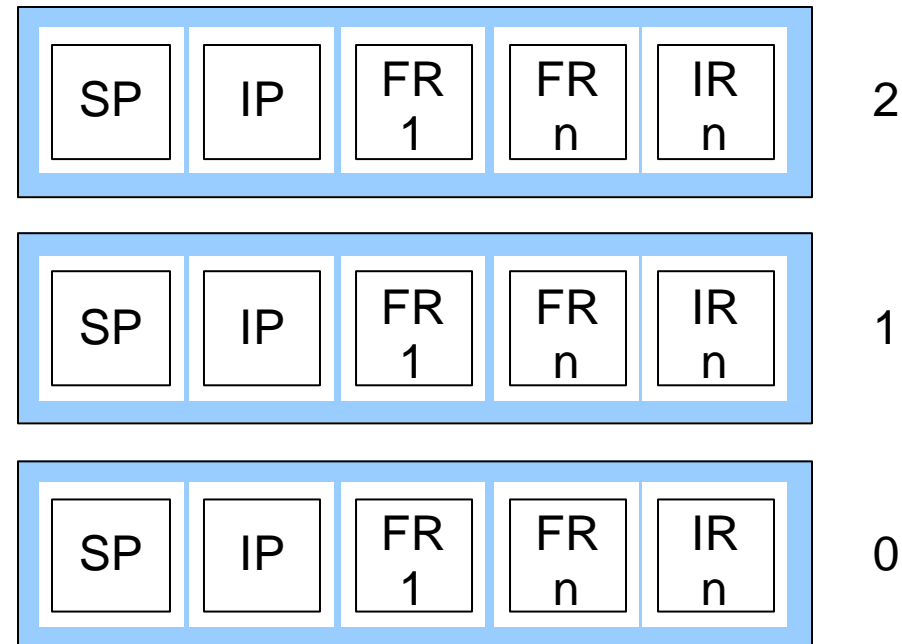


Structures

# 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 : Prémption – La prémption 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

```
#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

---

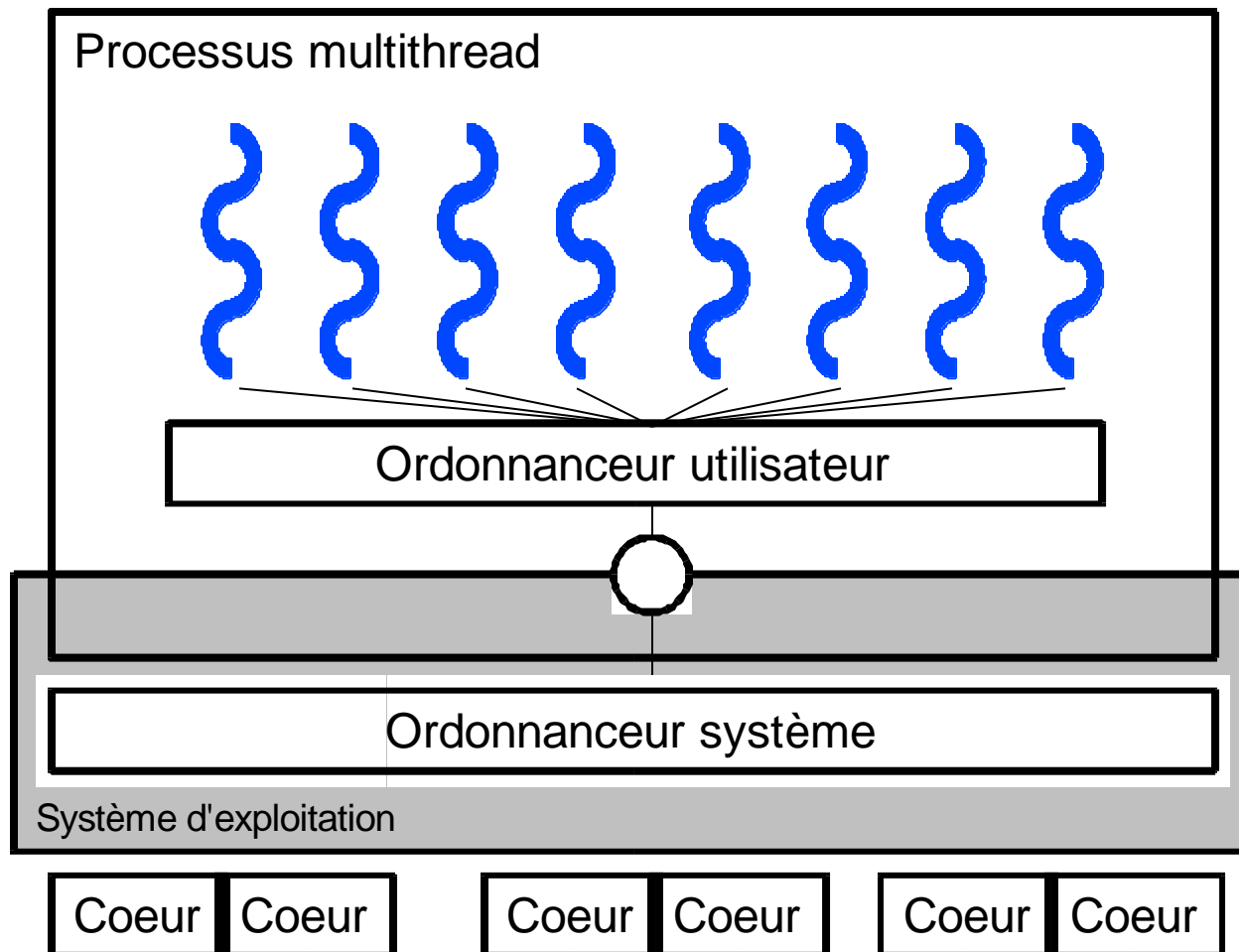


# Bibliothèques de threads

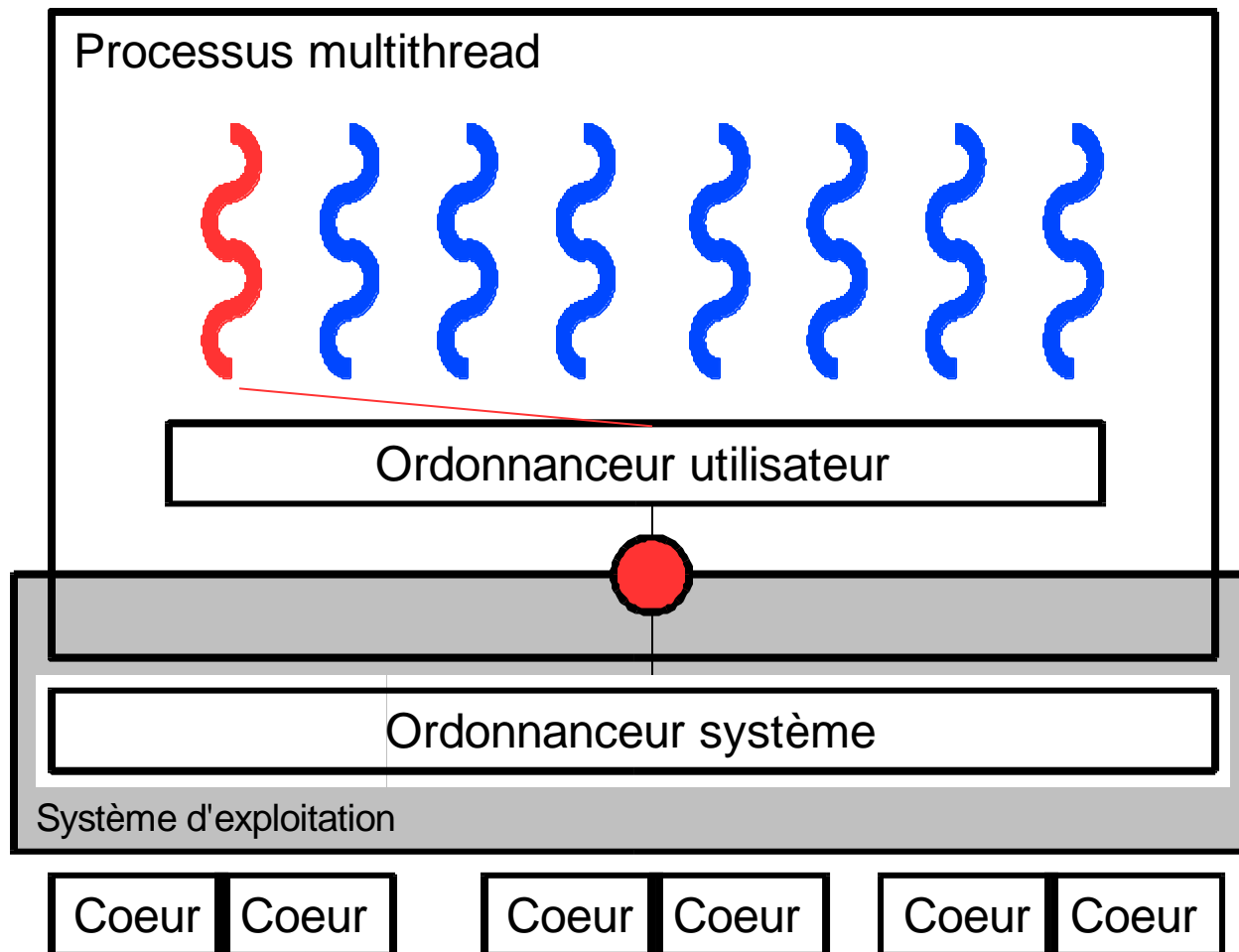
---

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

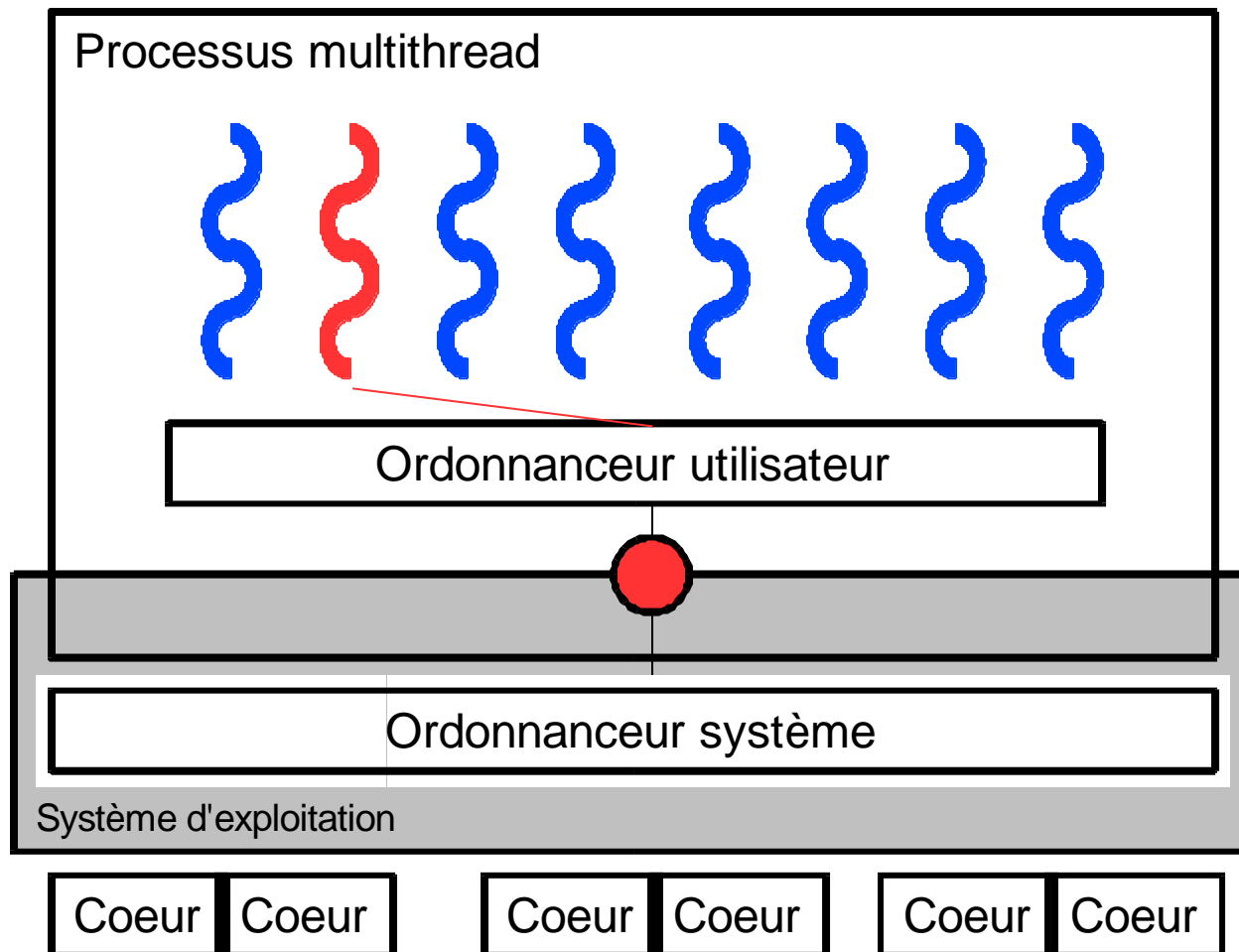
# Bibliothèque utilisateur



# Bibliothèque utilisateur

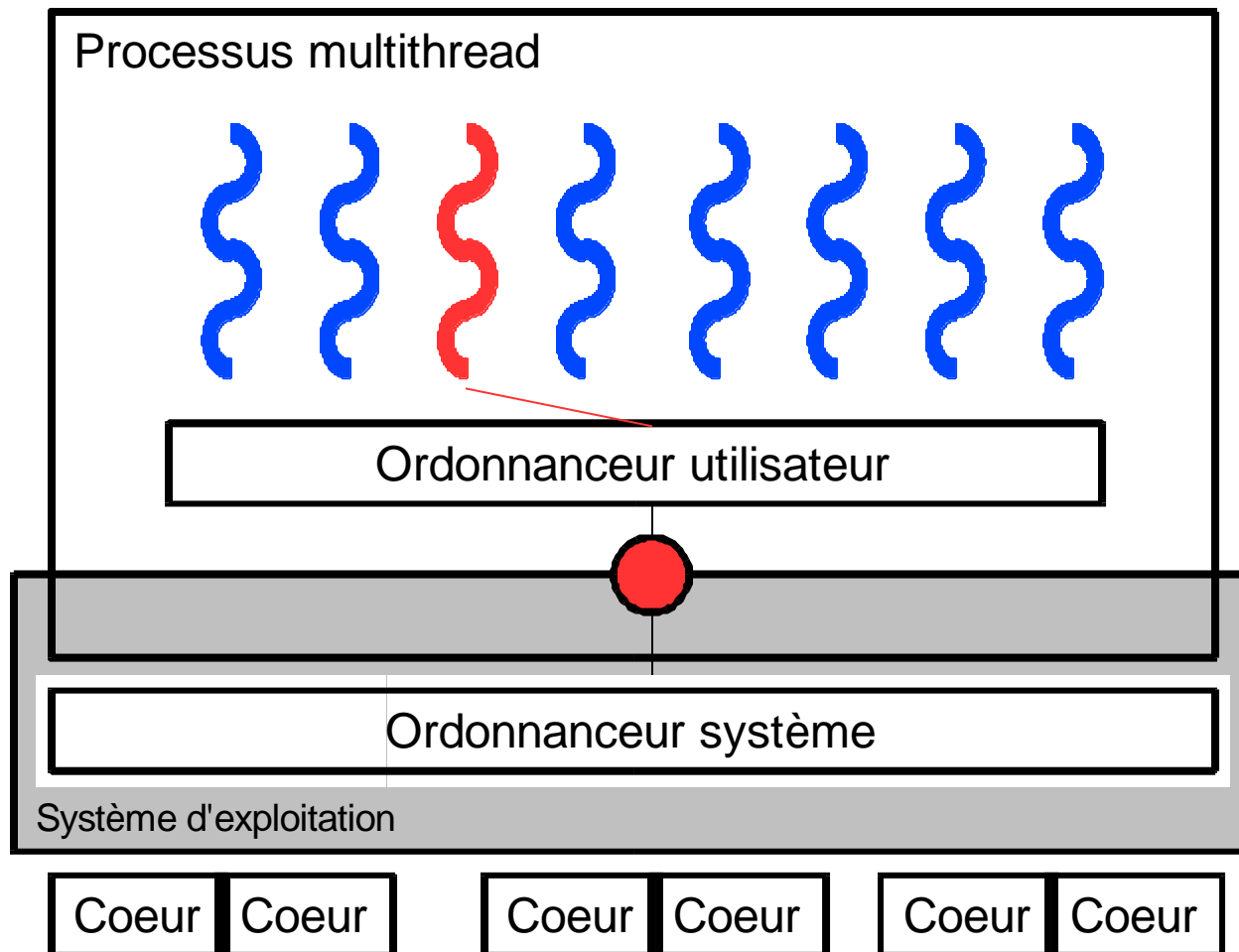


# Bibliothèque utilisateur

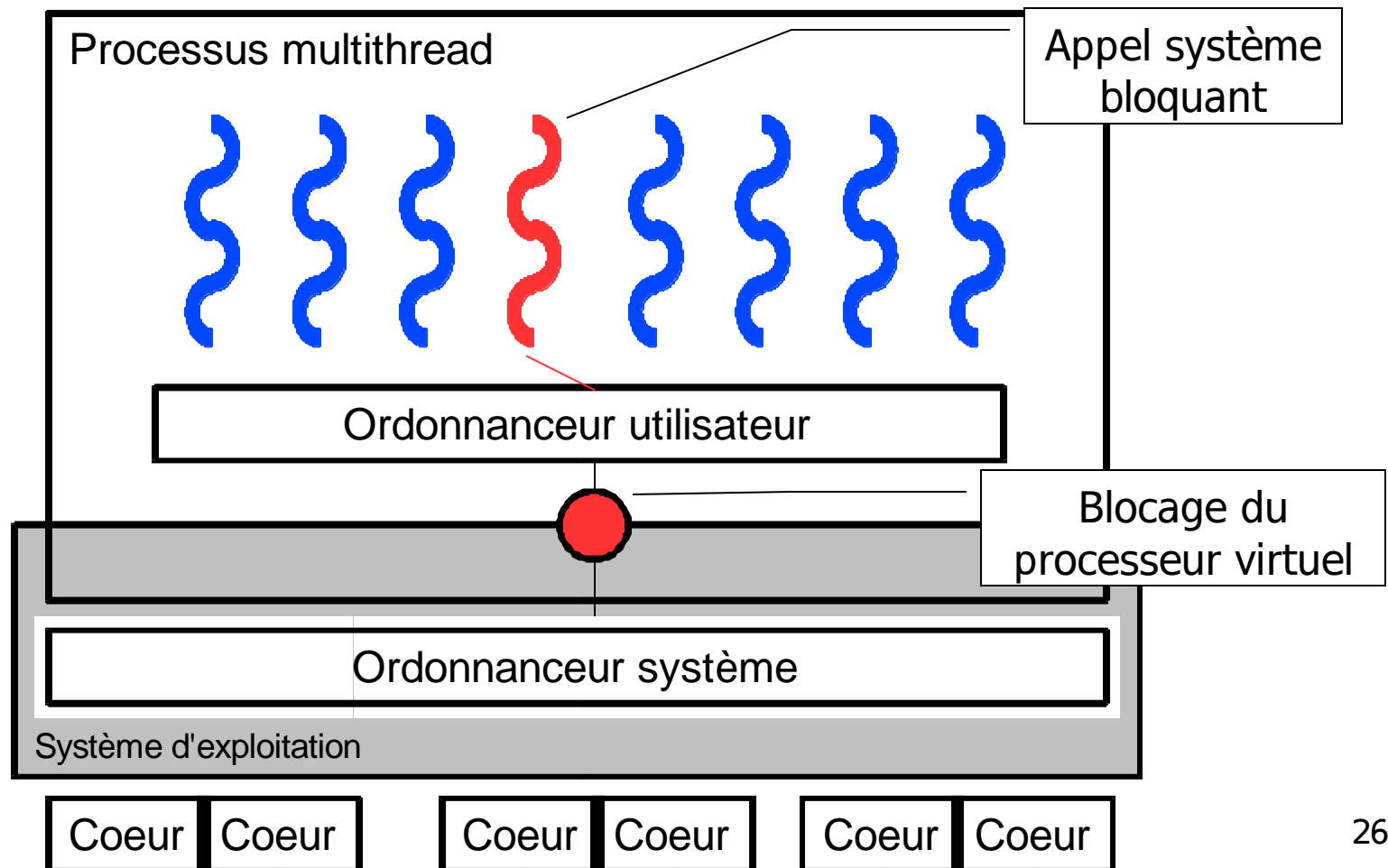




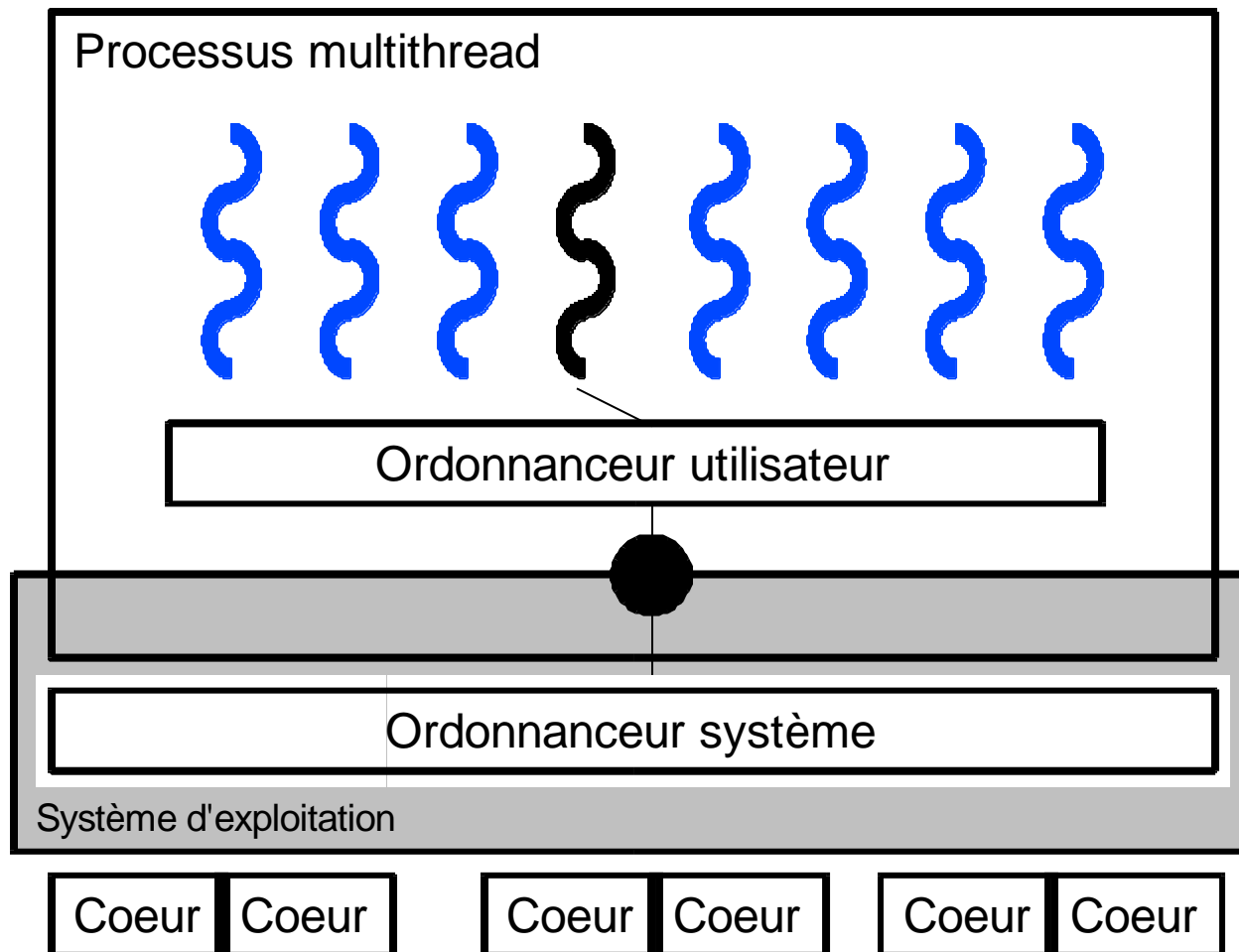
# Bibliothèque utilisateur



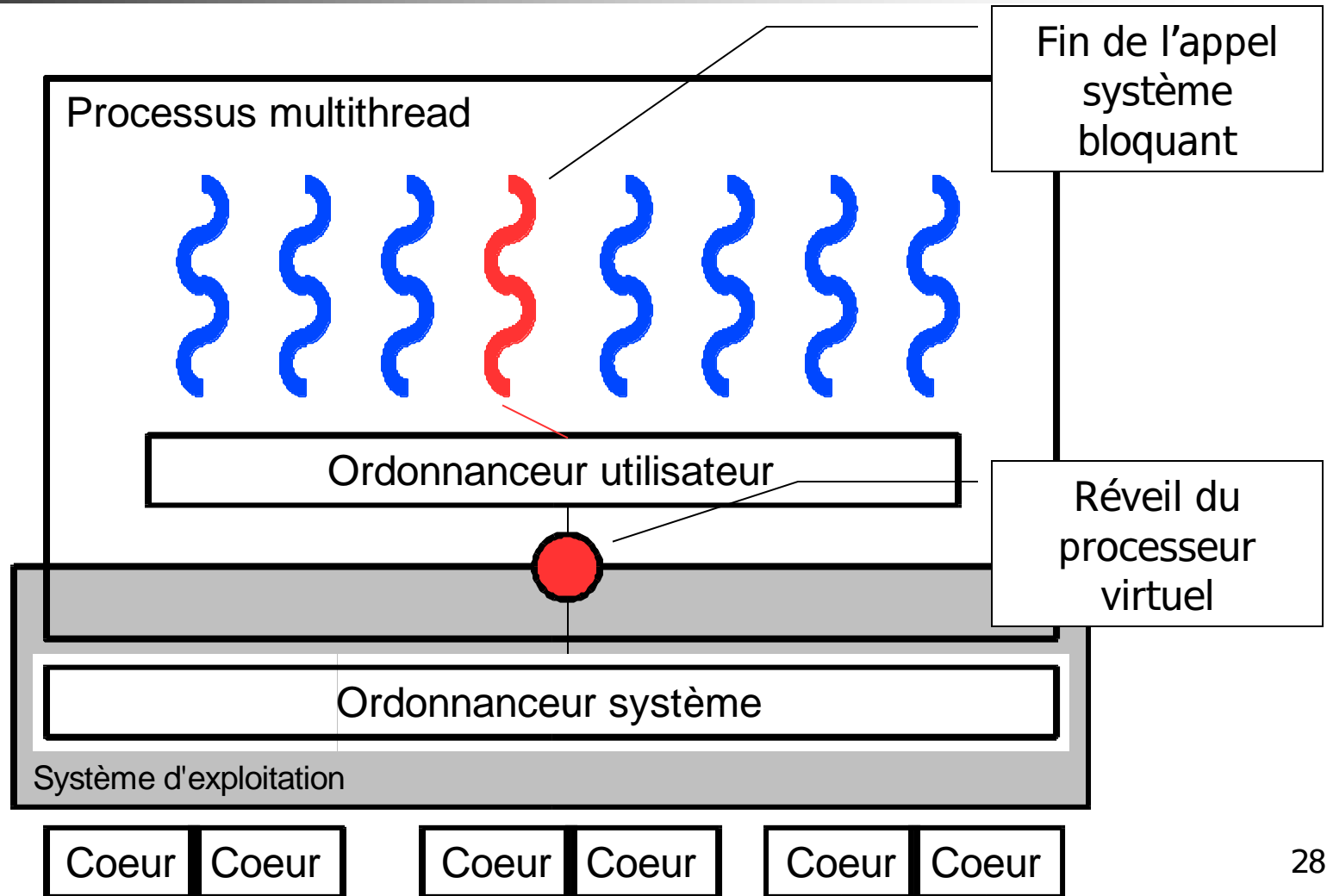
# Bibliothèque utilisateur



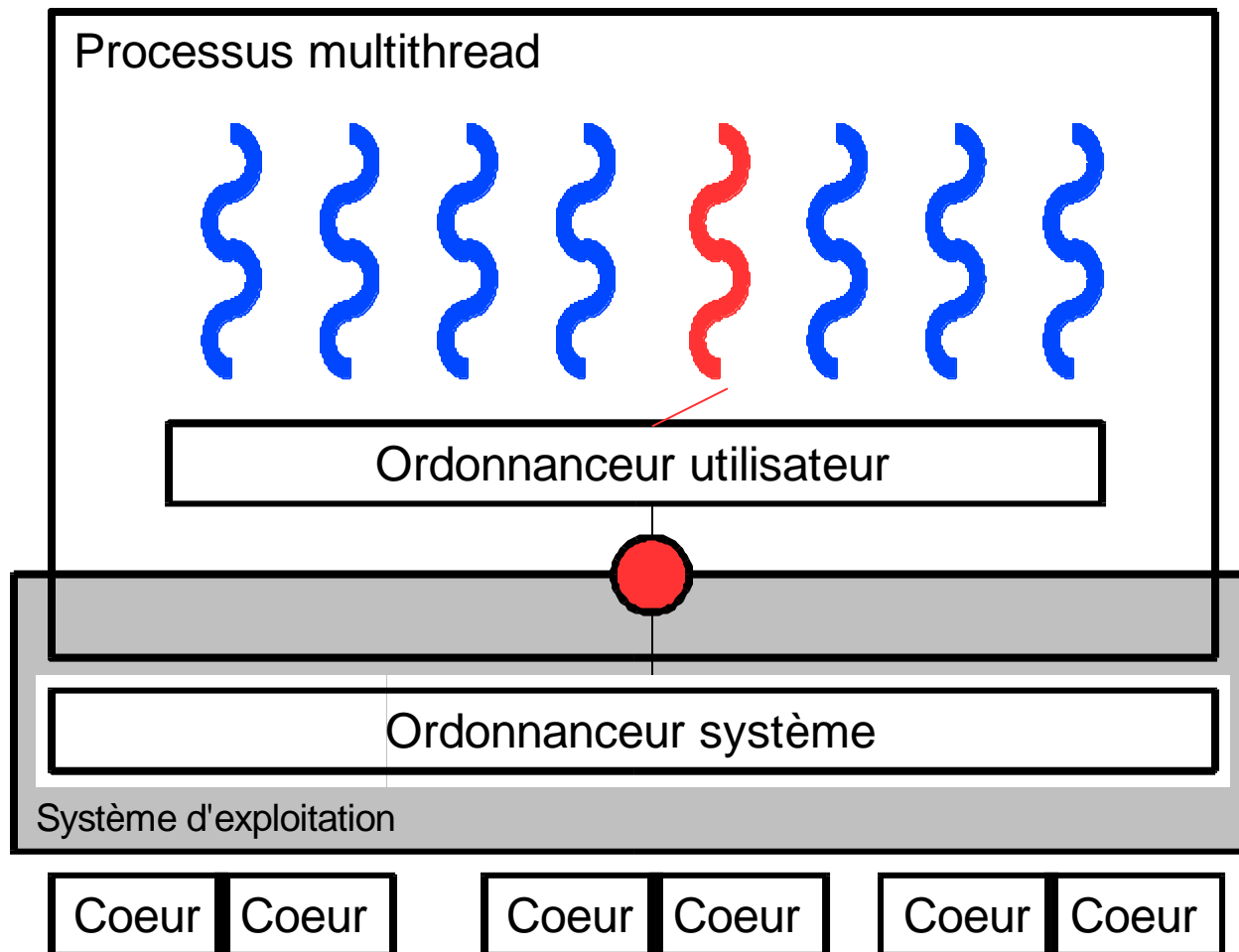
# Bibliothèque utilisateur



# Bibliothèque utilisateur



# Bibliothèque utilisateur



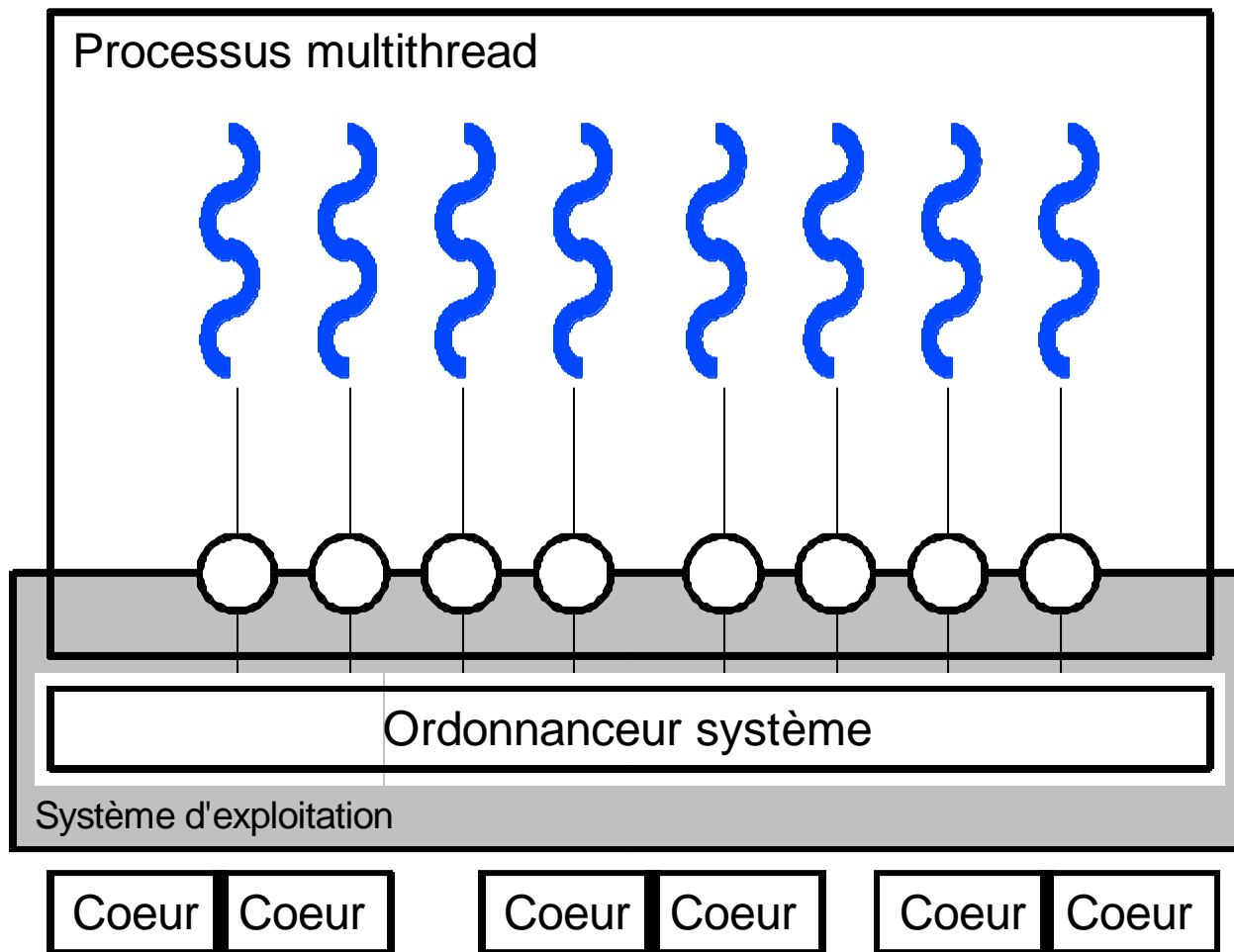


# Bibliothèque utilisateur

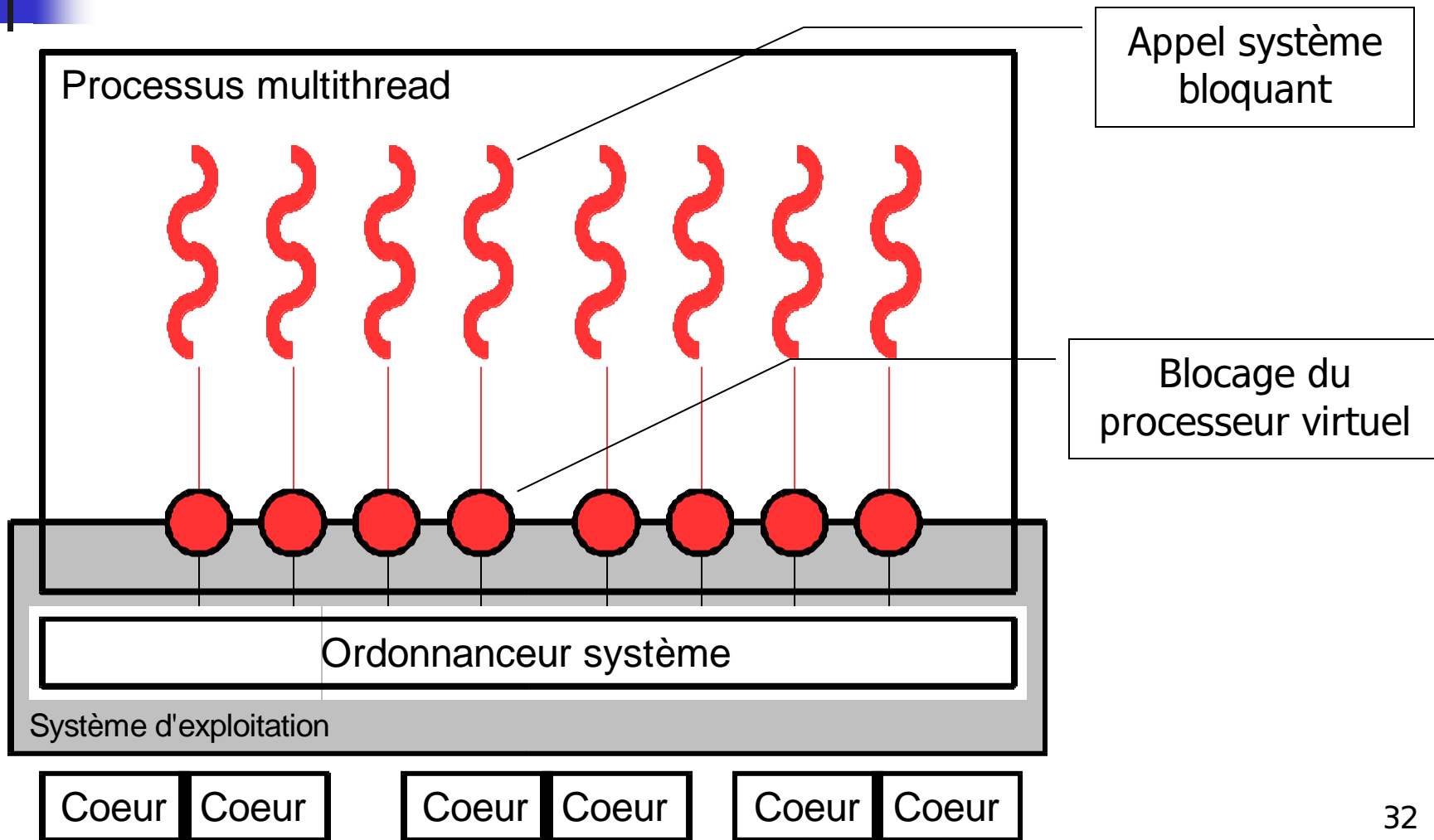
---

- 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

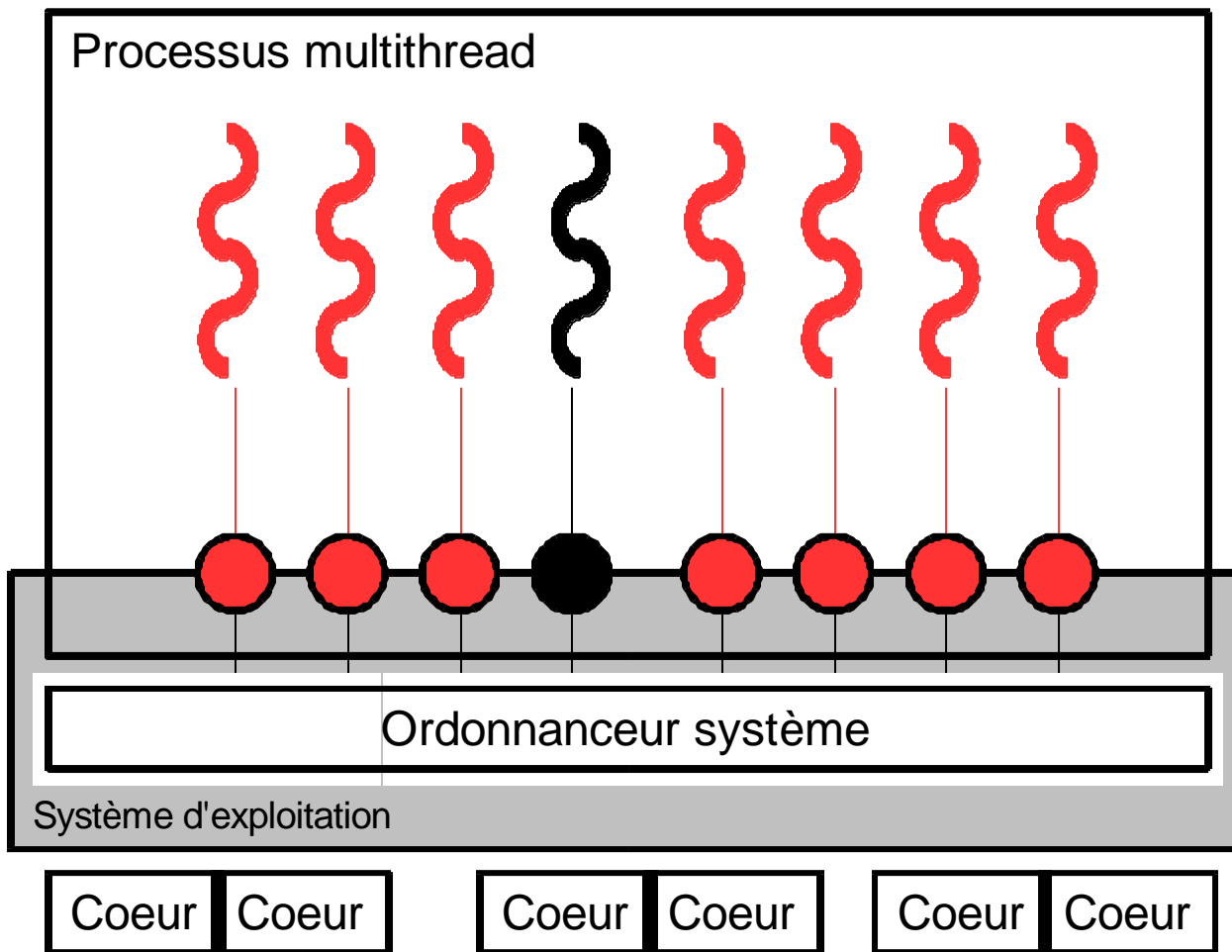


# Bibliothèque système

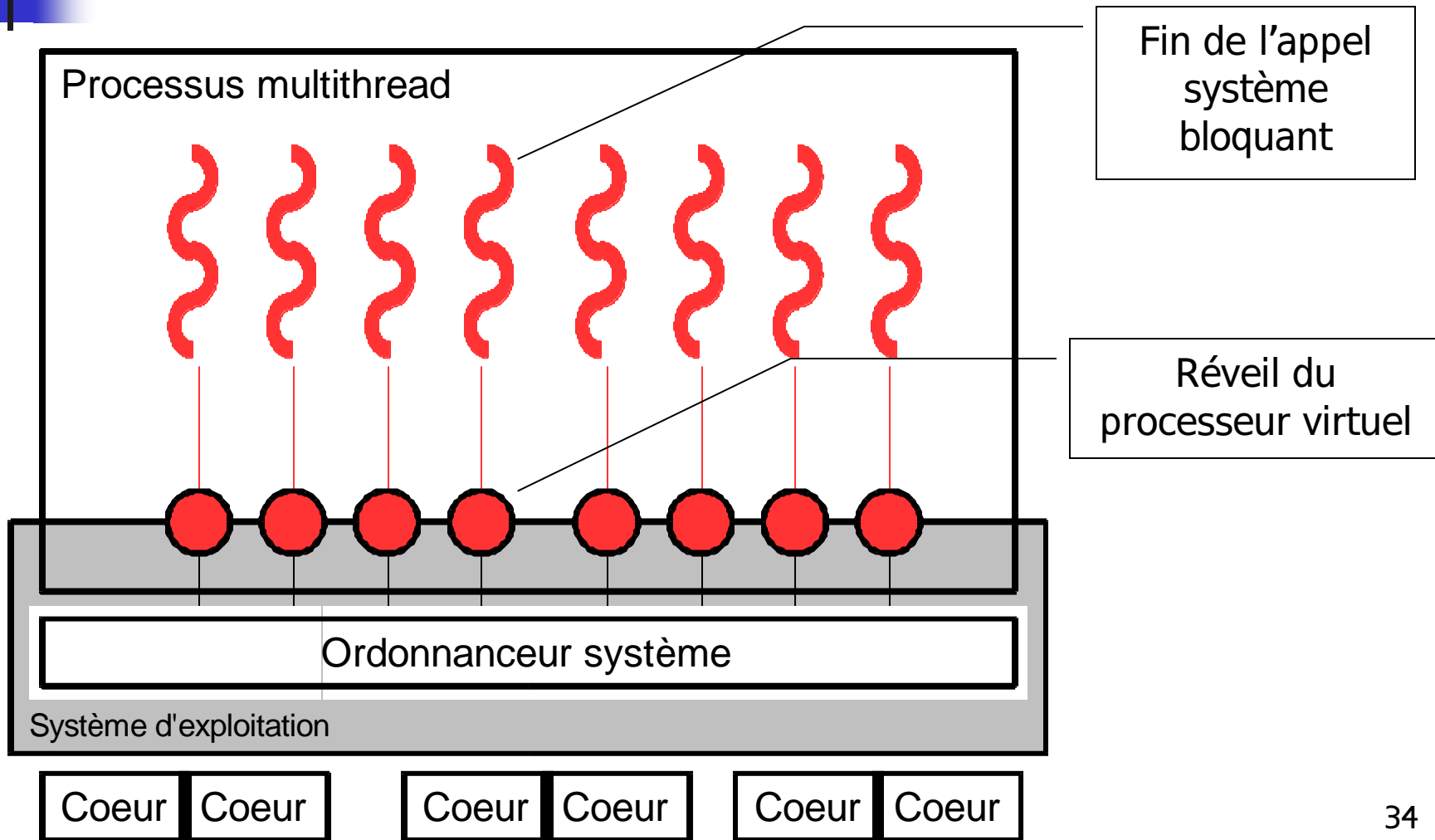




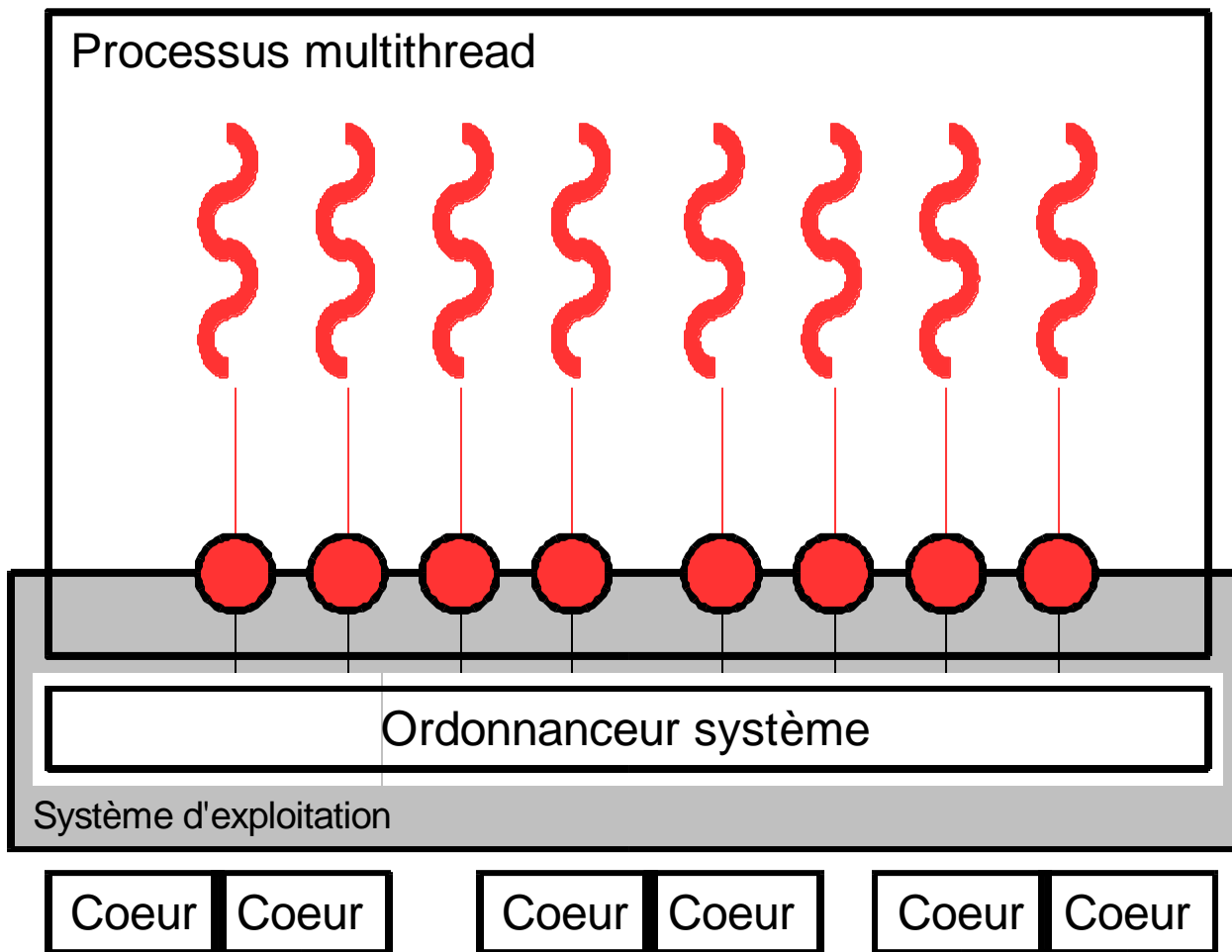
# Bibliothèque système



# Bibliothèque système



# Bibliothèque système



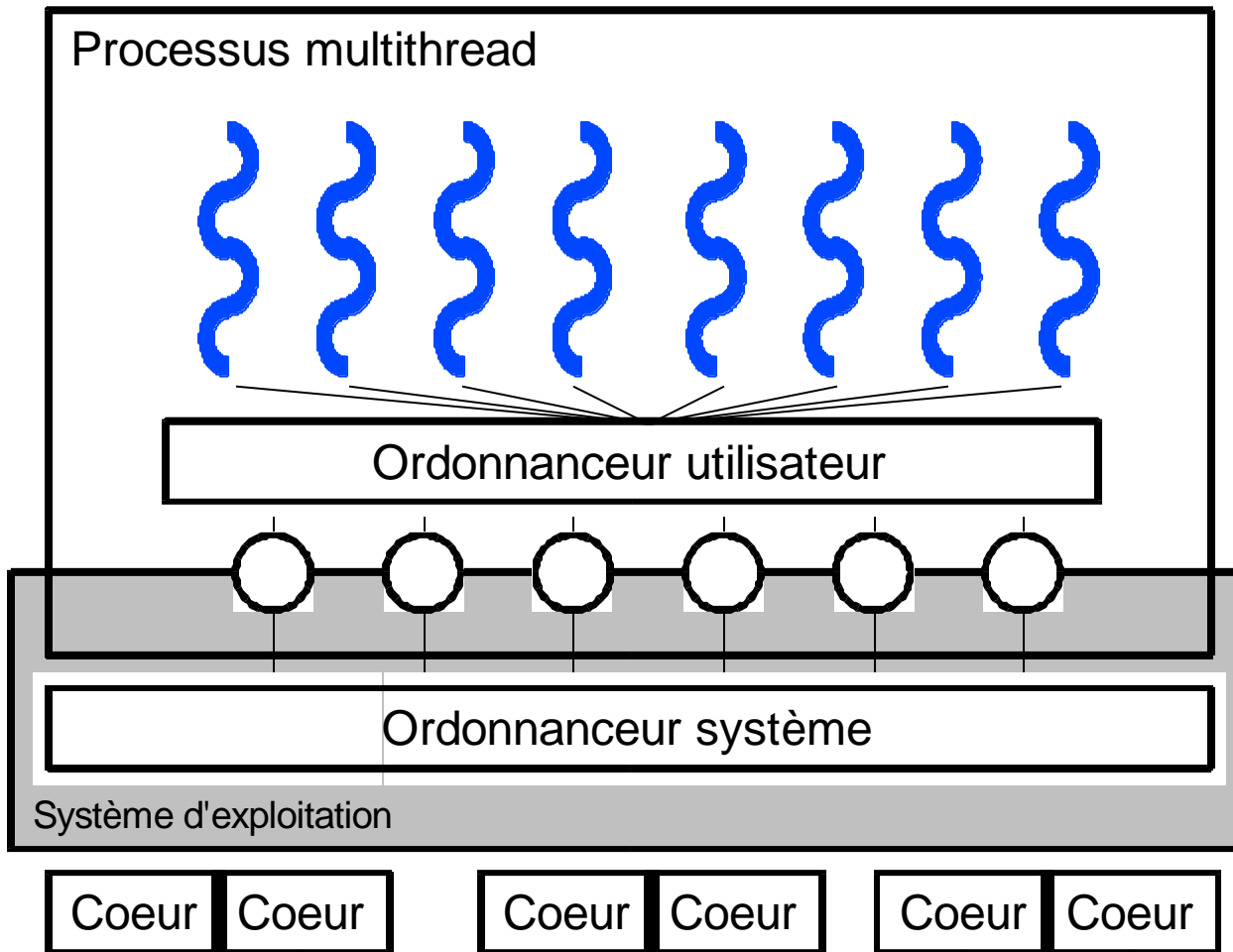


# Bibliothèque système

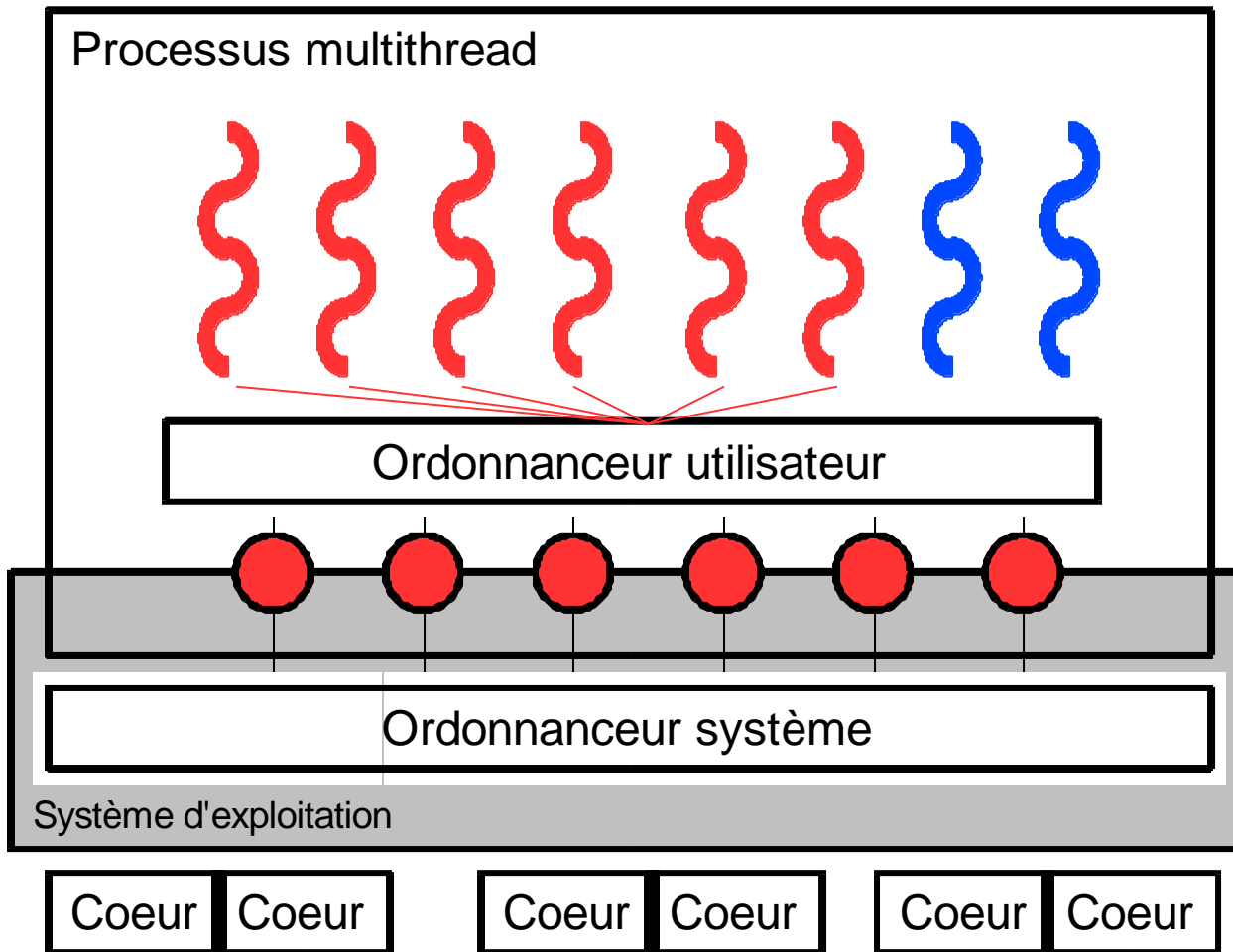
---

- 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

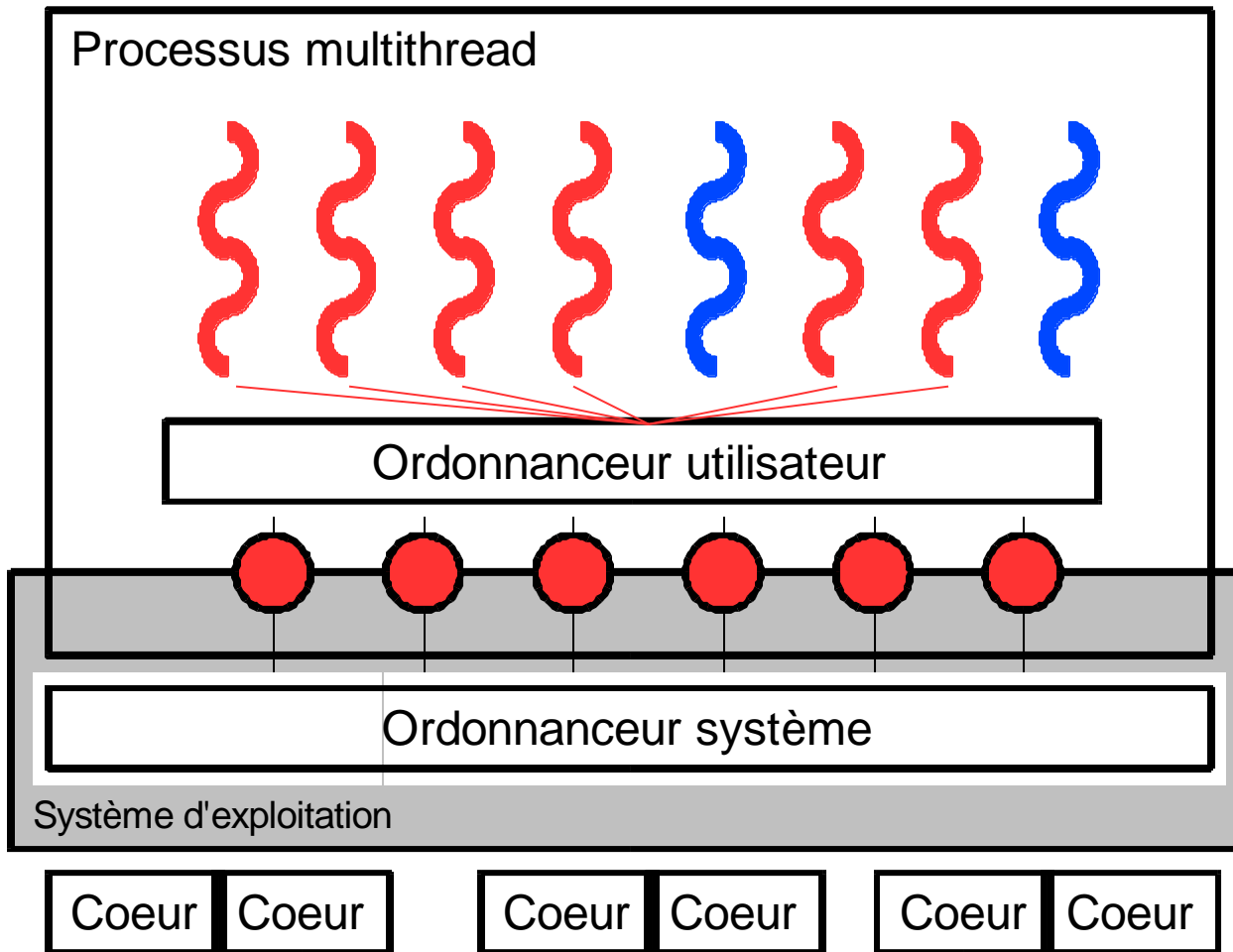
# Bibliothèque mixte (ou MxN)



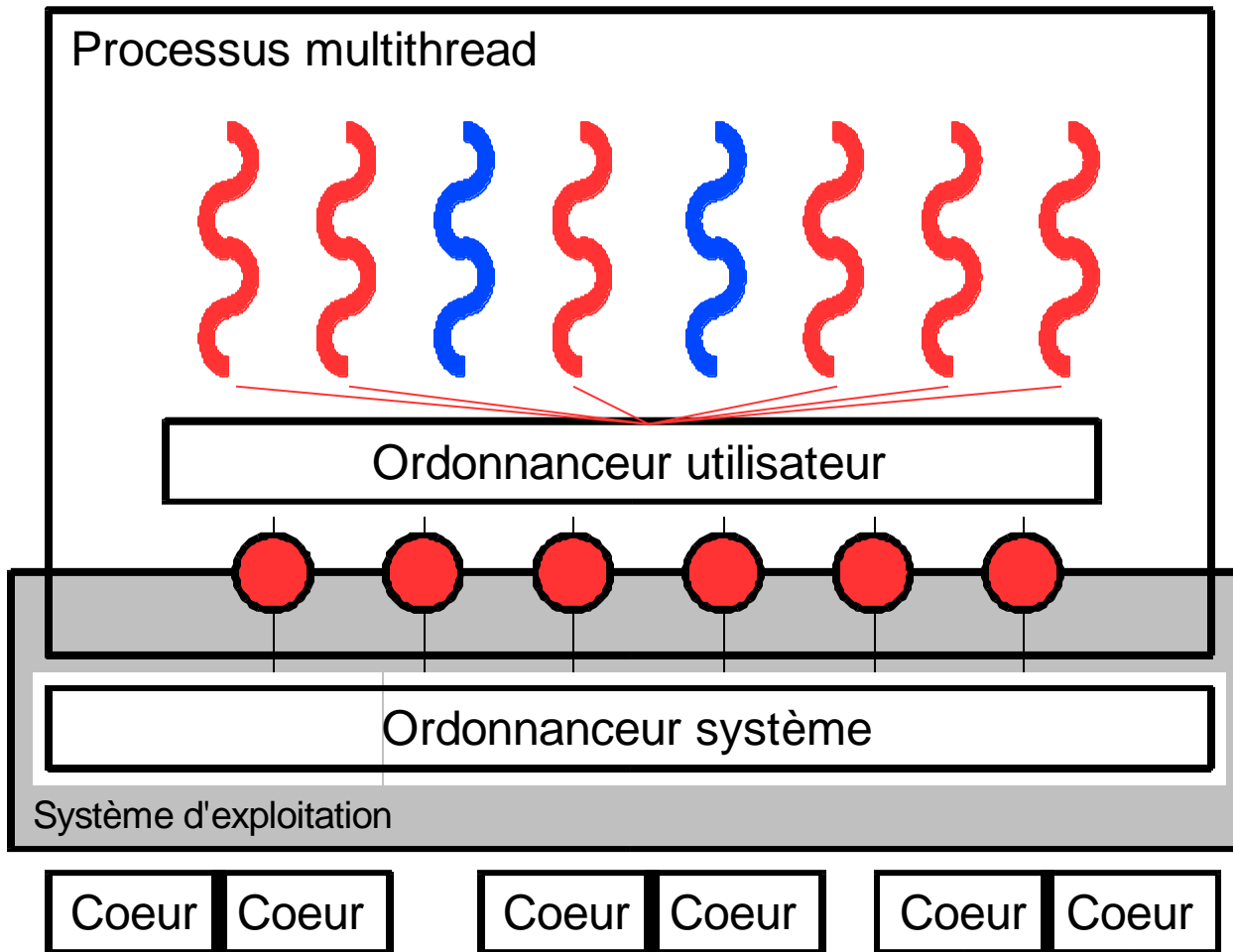
# Bibliothèque mixte (ou MxN)



# Bibliothèque mixte (ou MxN)

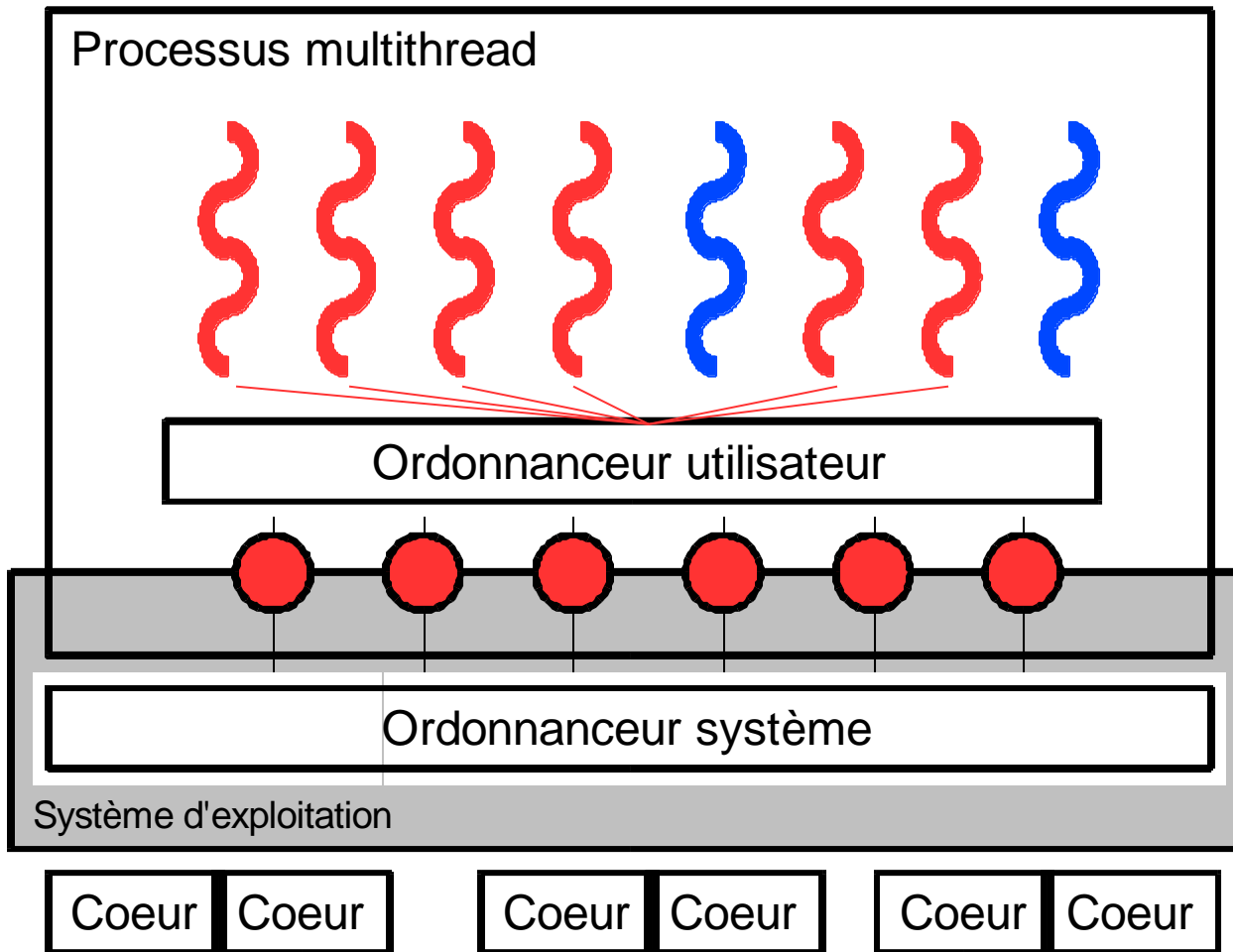


# Bibliothèque mixte (ou MxN)

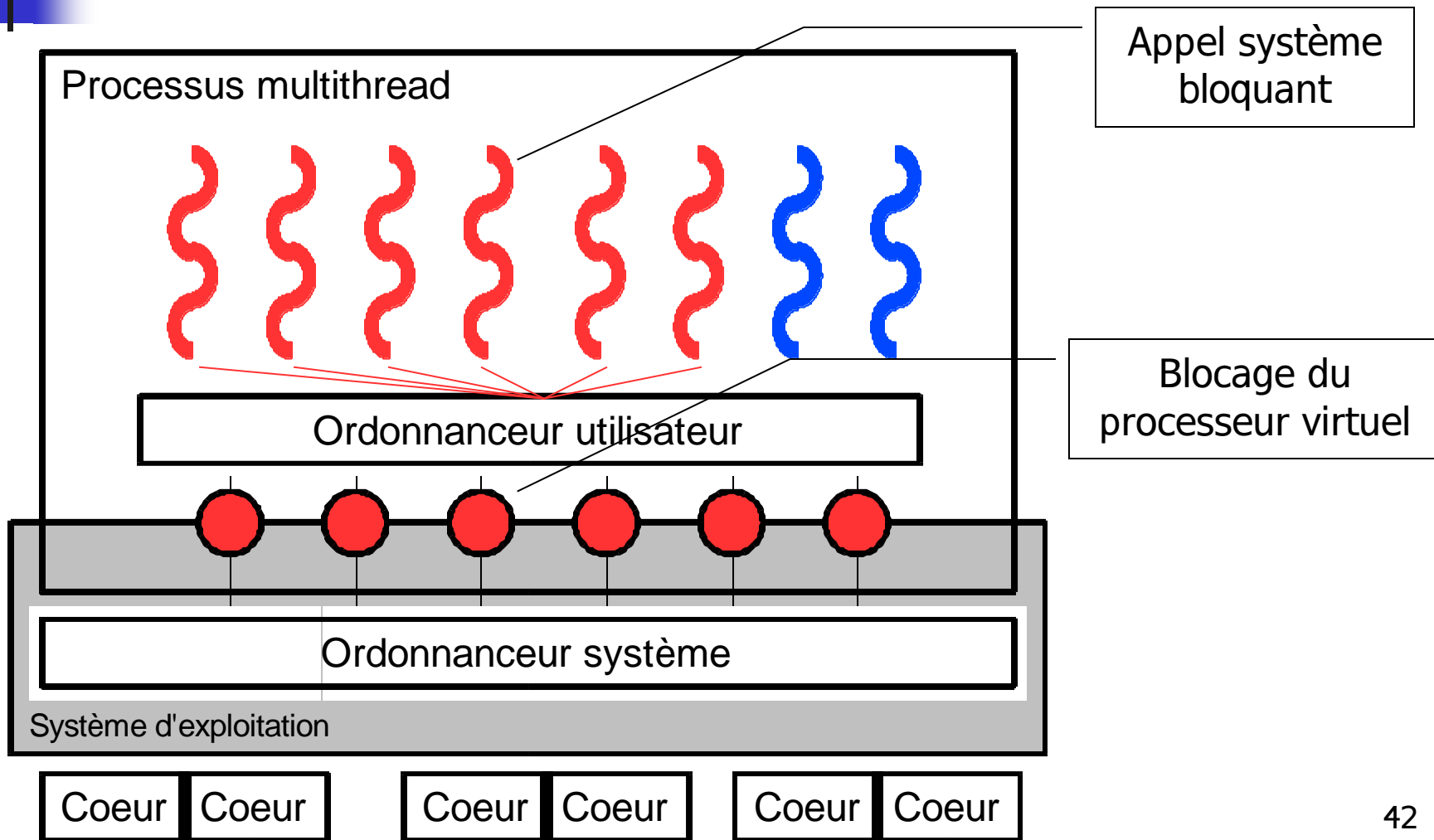




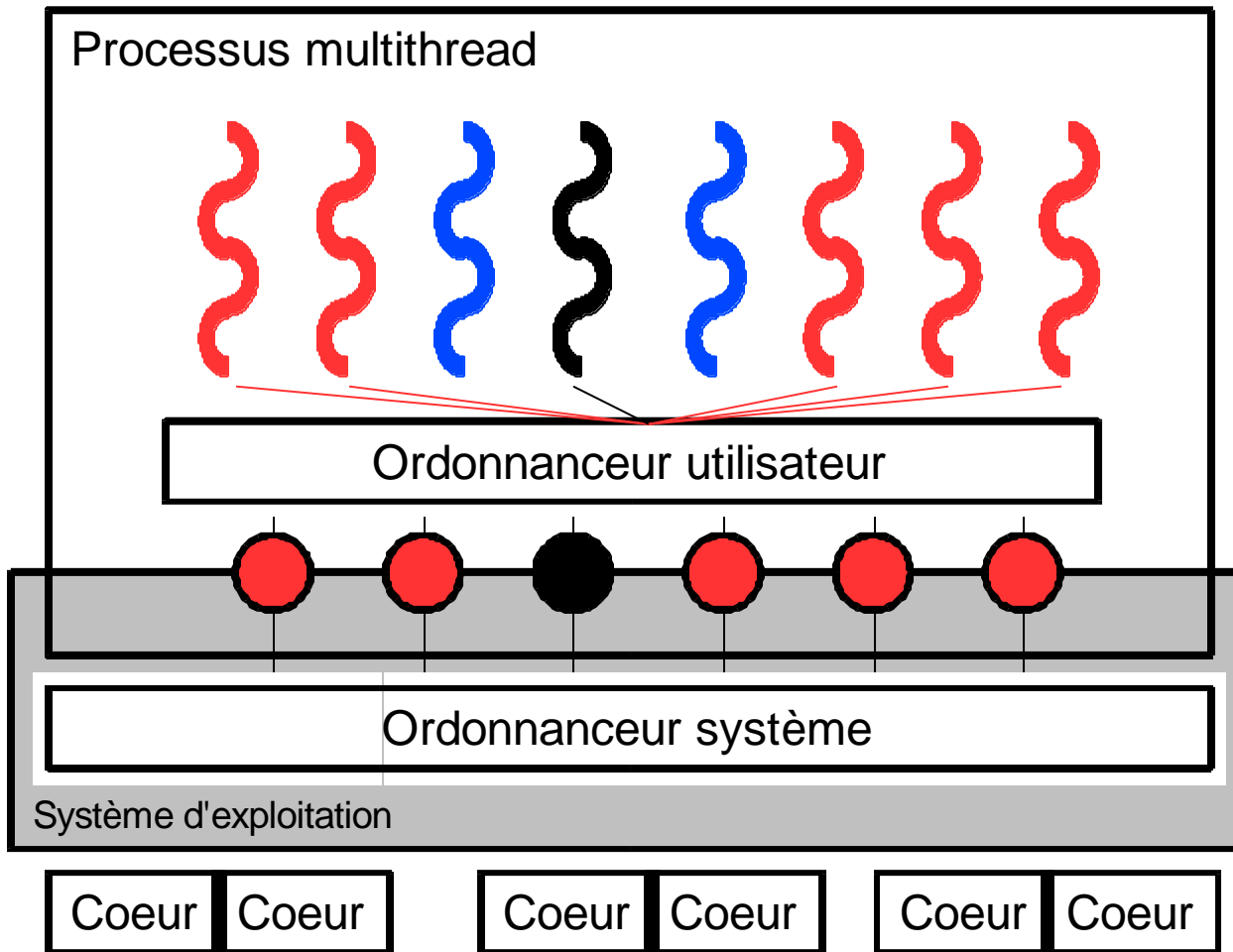
# Bibliothèque mixte (ou MxN)



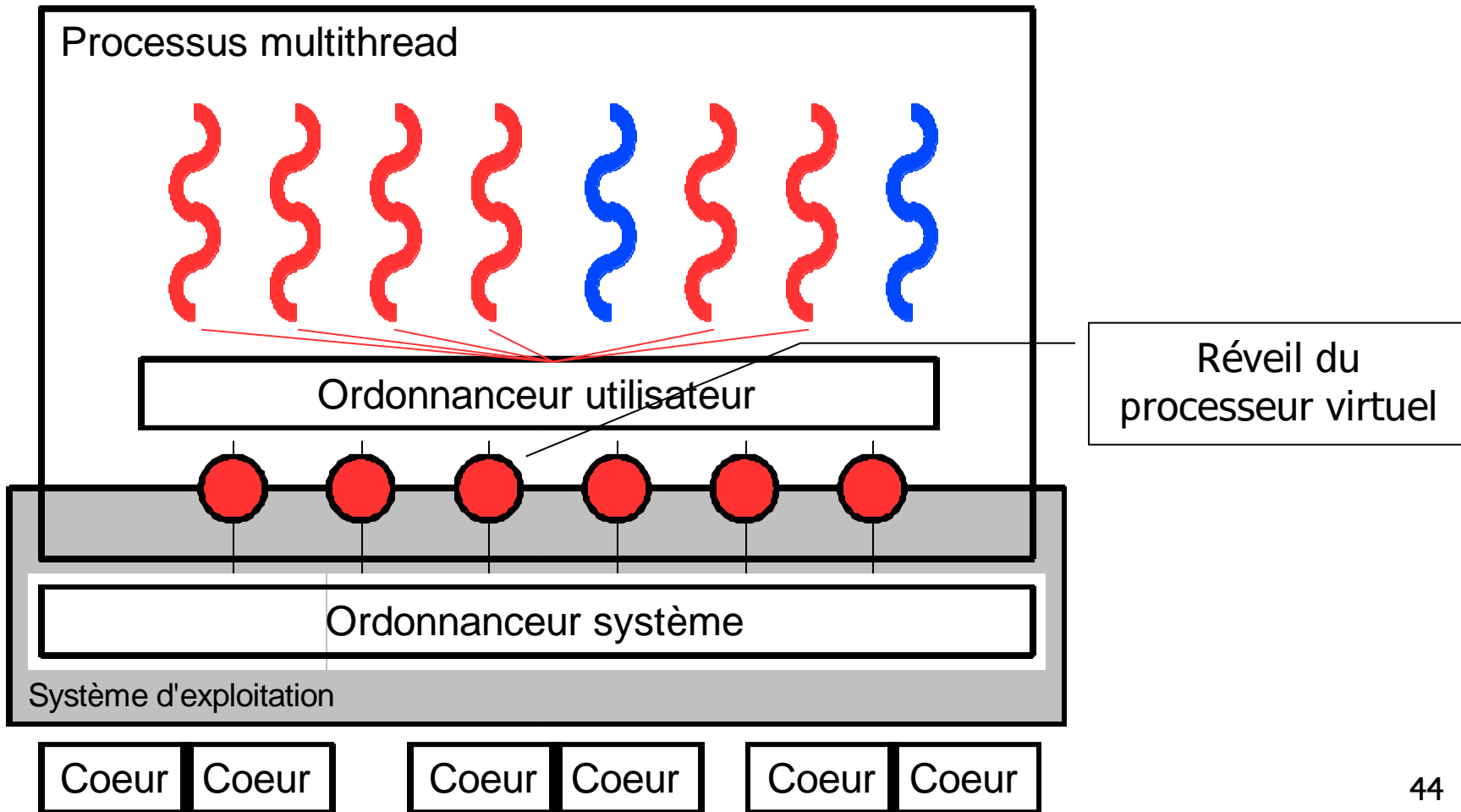
# Bibliothèque mixte (ou MxN)



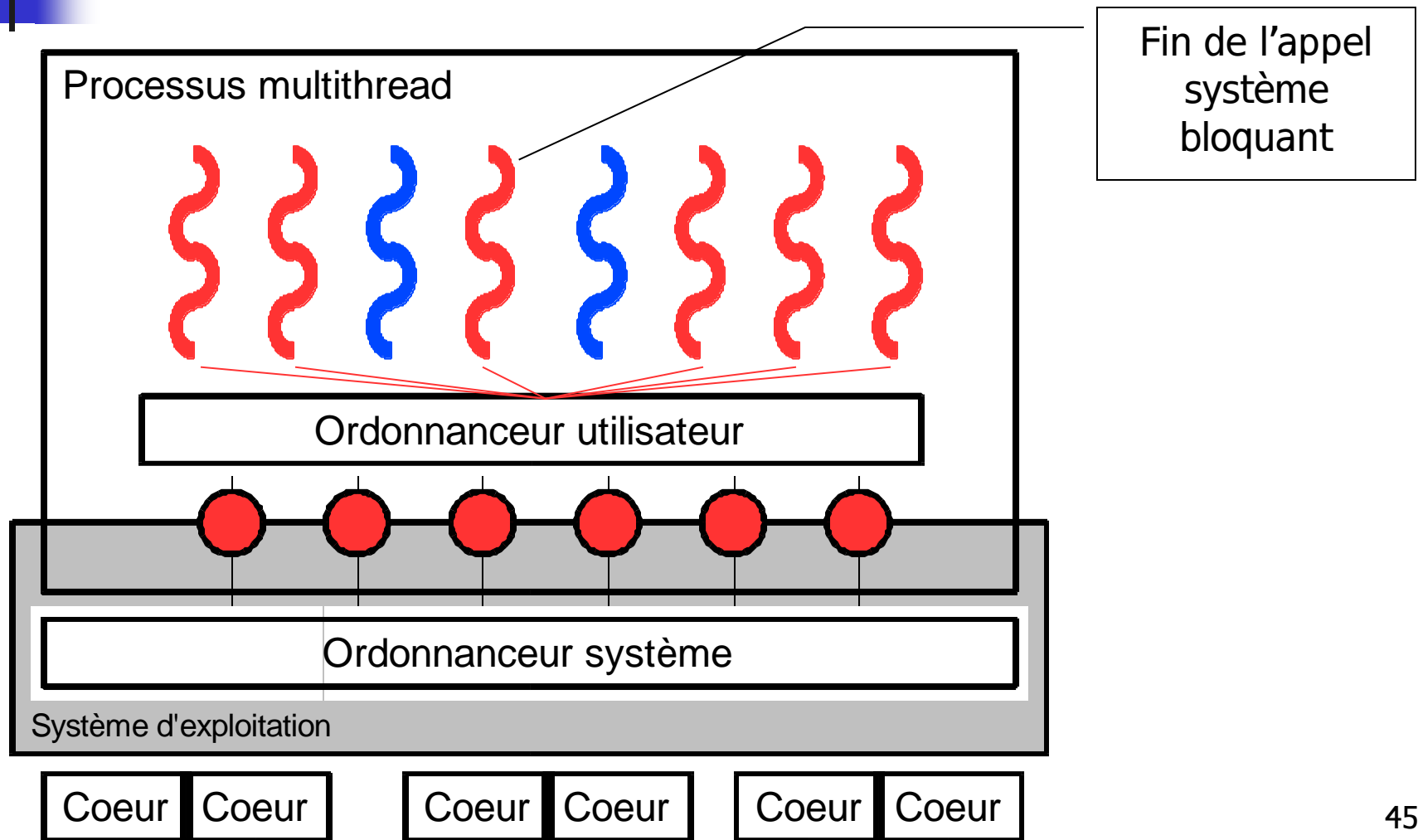
# Bibliothèque mixte (ou MxN)



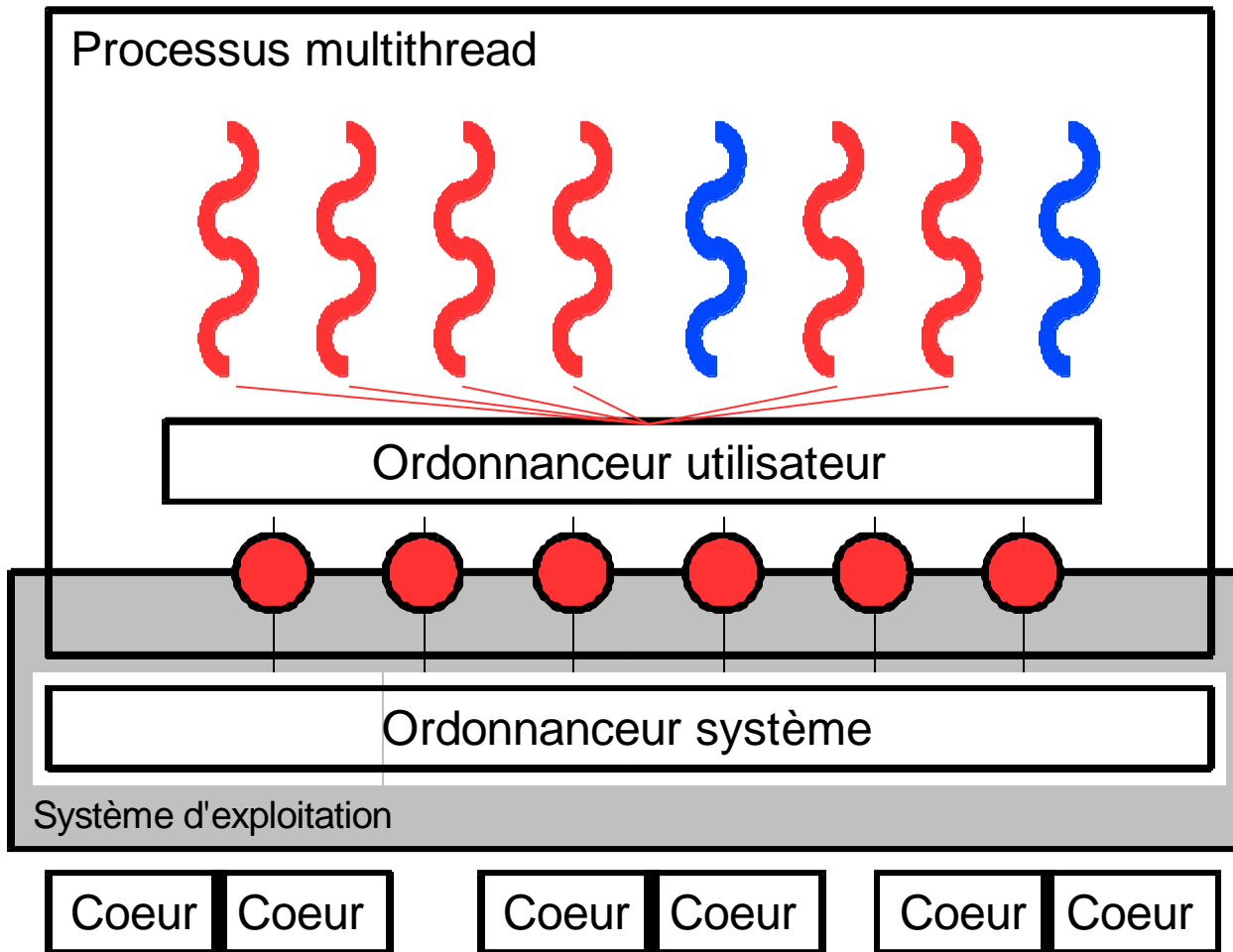
# Bibliothèque mixte (ou MxN)



# Bibliothèque mixte (ou MxN)



# Bibliothèque mixte (ou MxN)





# Bibliothèque mixte (ou MxN)

---

- M threads noyau pour N threads utilisateurs.
  - Deux ordonnanceurs : 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

---



# 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'incrémentement 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.

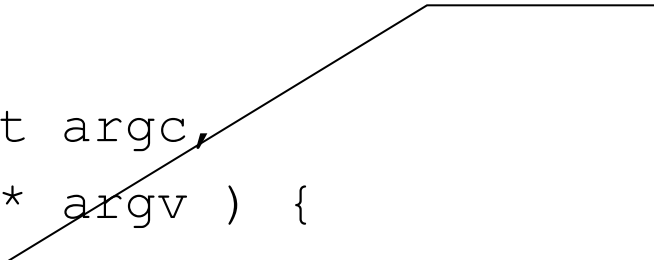


# Atomicité

---

```
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.
- Voir <https://gitlab.com/perache.marc/app>





# 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 partagent 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 !