Programmation système

Ressource R3.05 - Processus

monnerat@u-pec.fr

IUT de Fontainebleau

Sommaire

Définition

Processus

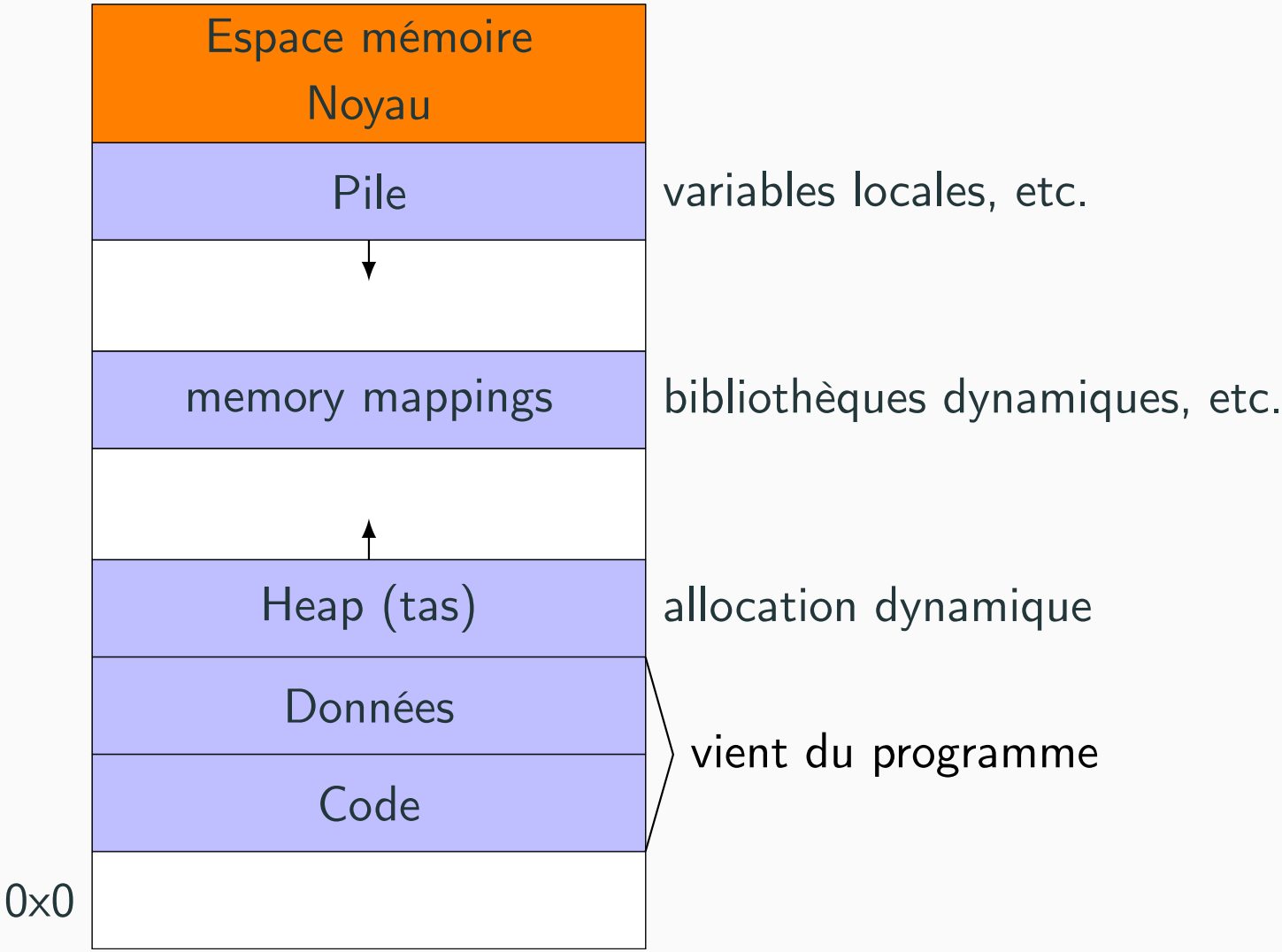
Image dynamique de l'exécution d'un programme.

- Un programme (fichier exécutable - format ELF) est statique. Il fournit :
  - Le code (instructions) - Code Segment
  - Les données - Data Segment
- Son exécution par le SE est dynamique ⇒ Processus

Durant toute sa vie, l'état d'un processus comprend :

- mémoire : code, data, tas, pile.
- contexte d'exécution : registres, compteur ordinal, sommet de pile, etc.
- Process control Bloc (PCB) : état vis à vis du SE.

Mémoire du processus

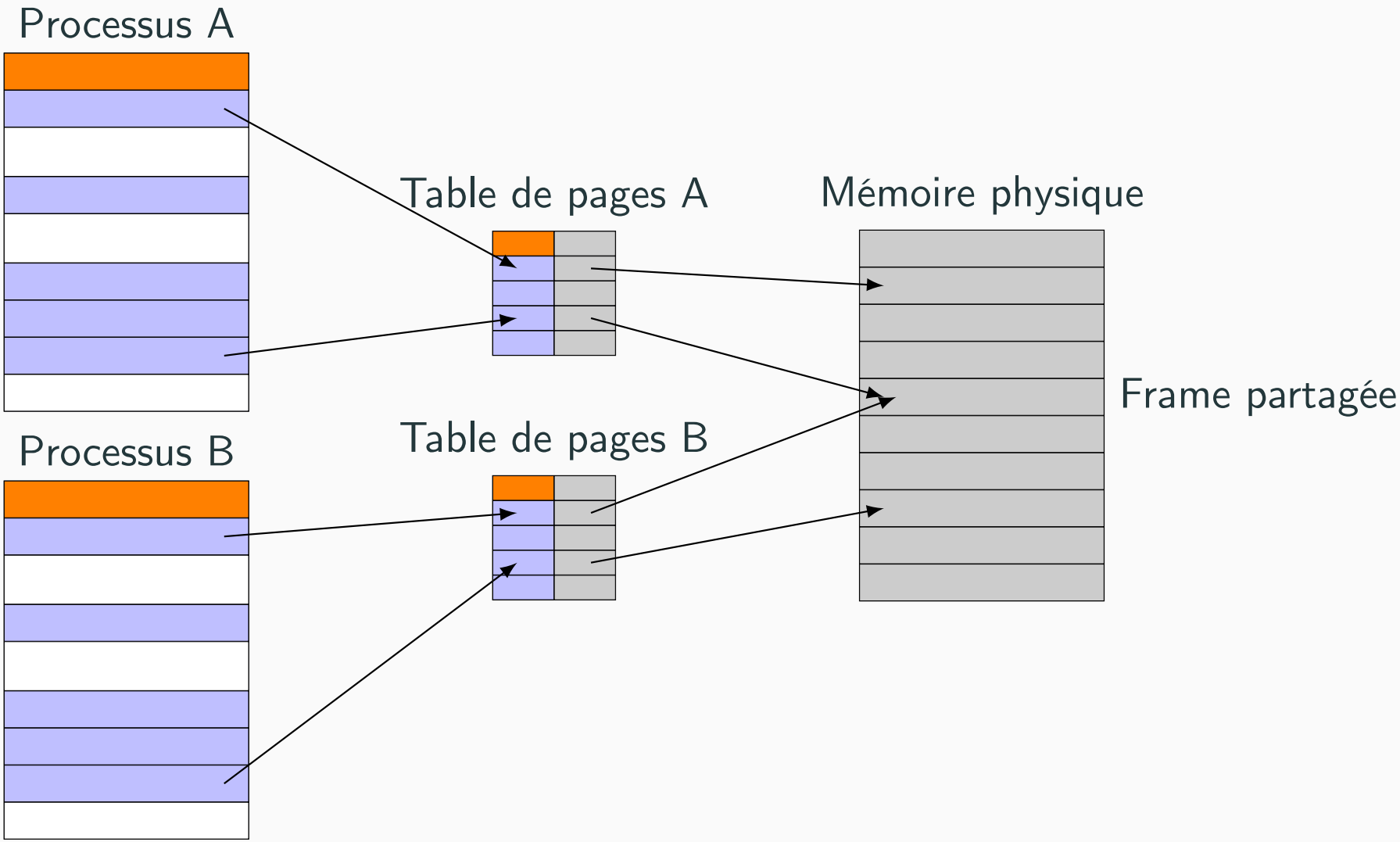


cat /proc/pid/maps

```
558d6b6fa000-558d6b6fb000 r--p 00000000 08:04 541510                     a.out
558d6b6fb000-558d6b6fc000 r-xp 00001000 08:04 541510                     a.out
558d6b6fc000-558d6b6fd000 r--p 00002000 08:04 541510                     a.out
558d6b6fd000-558d6b6fe000 r--p 00002000 08:04 541510                     a.out
558d6b6fe000-558d6b6ff000 rw-p 00003000 08:04 541510                     a.out
558d6ce5f000-558d6ce80000 rw-p 00000000 00:00 0                          [heap]
7f0fc83ed000-7f0fc83ef000 rw-p 00000000 00:00 0
7f0fc83ef000-7f0fc8415000 r--p 00000000 08:03 1969089                    /usr/lib/libc-2.32.so
7f0fc8415000-7f0fc8562000 r-xp 00026000 08:03 1969089                    /usr/lib/libc-2.32.so
7f0fc8562000-7f0fc85ae000 r--p 00173000 08:03 1969089                    /usr/lib/libc-2.32.so
7f0fc85ae000-7f0fc85b1000 r--p 001be000 08:03 1969089                    /usr/lib/libc-2.32.so
7f0fc85b1000-7f0fc85b4000 rw-p 001c1000 08:03 1969089                    /usr/lib/libc-2.32.so
7f0fc85b4000-7f0fc85ba000 rw-p 00000000 00:00 0
7f0fc8600000-7f0fc8602000 r--p 00000000 08:03 1969032                    /usr/lib/ld-2.32.so
7f0fc8602000-7f0fc8623000 r-xp 00002000 08:03 1969032                    /usr/lib/ld-2.32.so
7f0fc8623000-7f0fc862c000 r--p 00023000 08:03 1969032                    /usr/lib/ld-2.32.so
7f0fc862c000-7f0fc862d000 r--p 0002b000 08:03 1969032                    /usr/lib/ld-2.32.so
7f0fc862d000-7f0fc862f000 rw-p 0002c000 08:03 1969032                    /usr/lib/ld-2.32.so
7fffbe4a3000-7fffbe4c4000 rw-p 00000000 00:00 0                          [stack]
7fffbe59b000-7fffbe59f000 r--p 00000000 00:00 0                          [vvar]
7fffbe59f000-7fffbe5a1000 r-xp 00000000 00:00 0                          [vdso]
ffffffffff600000-ffffffffff601000 --xp 00000000 00:00 0                  [vsyscall]
```

Il s'agit d'adresses virtuelles !

Mémoire virtuelle vs mémoire physique



Caractéristiques - PCB

Le SE maintient (table) pour chaque processus des informations nécessaires (bloc de contrôle) :

- Identification : process ID (PID), parent PID (PPID), user ID (UID), effective UID (EUID), etc.
- État : actif, prêt, en attente, arrêté, terminé
- Contexte d'exécution (quand inactif)
- Mémoire : les adresses des segments, la table de pages
- Ressources : les fichiers ouverts, le terminal de rattachement, outils de synchronisation, etc.
- Ordonnancement : les priorités, les horloges et les alarmes
- Relations : les processus fils, le groupe, la session
- Signaux : pendants, bloqués, les gestionnaires, etc.
- Comptabilité pour des statistiques : temps, mémoire, entrées-sorties, etc.

Sources du noyau Linux : `struct task_struct` dans `linux/sched.h`

Arbre des processus

`ps -u denis -o stat,tty,pid,tpgid,sess,pgrp,ppid,comm`

```
STAT TT         PID TPGID  SESS  PGRP  PPID COMMAND
Ss   ?          965    -1   965   965     1 systemd
S    ?          966    -1   965   965   965 (sd-pam)
S    ?          976    -1   719   719   719 i3
Sl   ?          982    -1   981   981     1 dunst
S    ?          988    -1   987   987     1 i3bar
Ss   ?          990    -1   990   990     1 udisks-glue
Ss   ?          993    -1   993   993   965 dbus-daemon
Sl   ?         1008    -1   987  1008   988 python /home/de
Ssl  ?         1033    -1  1033  1033   965 pulseaudio
Sl   ?         1048    -1  1047  1047     1 thunderbird
Ssl  ?         1068    -1  1068  1068   965 gvfsd
Sl   ?         1073    -1  1068  1068   965 gvfsd-fuse
Ssl  ?         1113    -1  1113  1113   965 at-spi-bus-laun
Sl   ?         1158    -1  1135  1135     1 chromium
S    ?         1185    -1  1135  1135  1158 chromium
S+   pts/0     2907  2907  2243  2907  2243 vim
Sl   ?         2953    -1  2231  2231     1 termite
Ss   pts/2     2959 18942  2959  2959  2953 bash
Sl   ?         3687    -1  1135  1135  1188 chromium
Sl   ?         8666    -1  2313  2313     1 termite
Ss   pts/3     8671  8767  8671  8671  8666 bash
S+   pts/3     8767  8767  8671  8767  8671 vim
R+   pts/2    18942 18942  2959 18942  2959 ps
```

Tout processus a un père !

- PID : numéro unique attribué par le SE à la création.

```
pid_t getpid(void);
```

- PPID : PID du pére du processus.

```
pid_t getppid(void);
```

- UID : utilisateur qui a créé le processus.

```
uid_t getuid(void);
```

- EUID : utilisateur effectif.

```
uid_t geteuid(void);
```

- GID : groupe du propriétaire

```
gid_t getgid(void);
```

```c
#include <unistd.h>
#include <stdio.h>
int main(int argc, char *argv[])
{
  printf("Mon PID est %d\n", getpid());
  printf("Le PID de mon père est %d\n", getppid());
  sleep(15); // pour garder le processus vivant
  return 0;
}
```

```
[denis@portabledenis]$ ./a.out
Mon PID est 4020
Le PID de mon père est 2959
```

```
[denis@portabledenis]$ ps --forest
PID TTY          TIME CMD
2959 pts/2    00:00:00 bash
4020 pts/2    00:00:00  \_ a.out
```

Cycle de vie

Cycle de vie

Création

Appel système `fork()`

```c
#include <sys/types.h>
#include <unistd.h>
pid_t fork(void);
```

Crée un nouveau processus qui hérite presque tout (de qui ?) :

- le contenu mémoire avec copy-on-write
  même code et mêmes données
- le propriétaire (UID, GID, EUID, etc.)
- les fichiers ouverts
- gestion des signaux
- variables d'environnement
- etc.

Après sa création, le processus est placé dans l'état `TASK_RUNNING`

Le code du pére et du fils est commun. Comment les différencier ?

Le retour de l'appel `fork()` est différent suivant que l'on est dans le processus appelant (le père) ou le processus nouvellement créé (le fils)

Le retour de la primitive `fork()` permet de savoir "où" l'on est.

```
pid_t p=fork();
```

- `p < 0` : erreur, pas de création de processus
- `p > 0` : on est dans le père. p vaut le pid du fils nouvellement créé.
- `p == 0` : on est dans le fils nouvellement créé.

```c
#include <headers_qui_vont_bien.h>
int main()
{
  pid_t p;
  p=fork();
  switch(p){
    case -1: perror("erreur fork");
         exit(1);

    case 0: printf("\t FILS \n");
        break;
    default: printf("PERE \n");
  }
  printf("Aurevoir\n");
}
```

Combien de fois "Aurevoir" est-il affiché ?

La mémoire est-elle partagée ? Expliquez ?

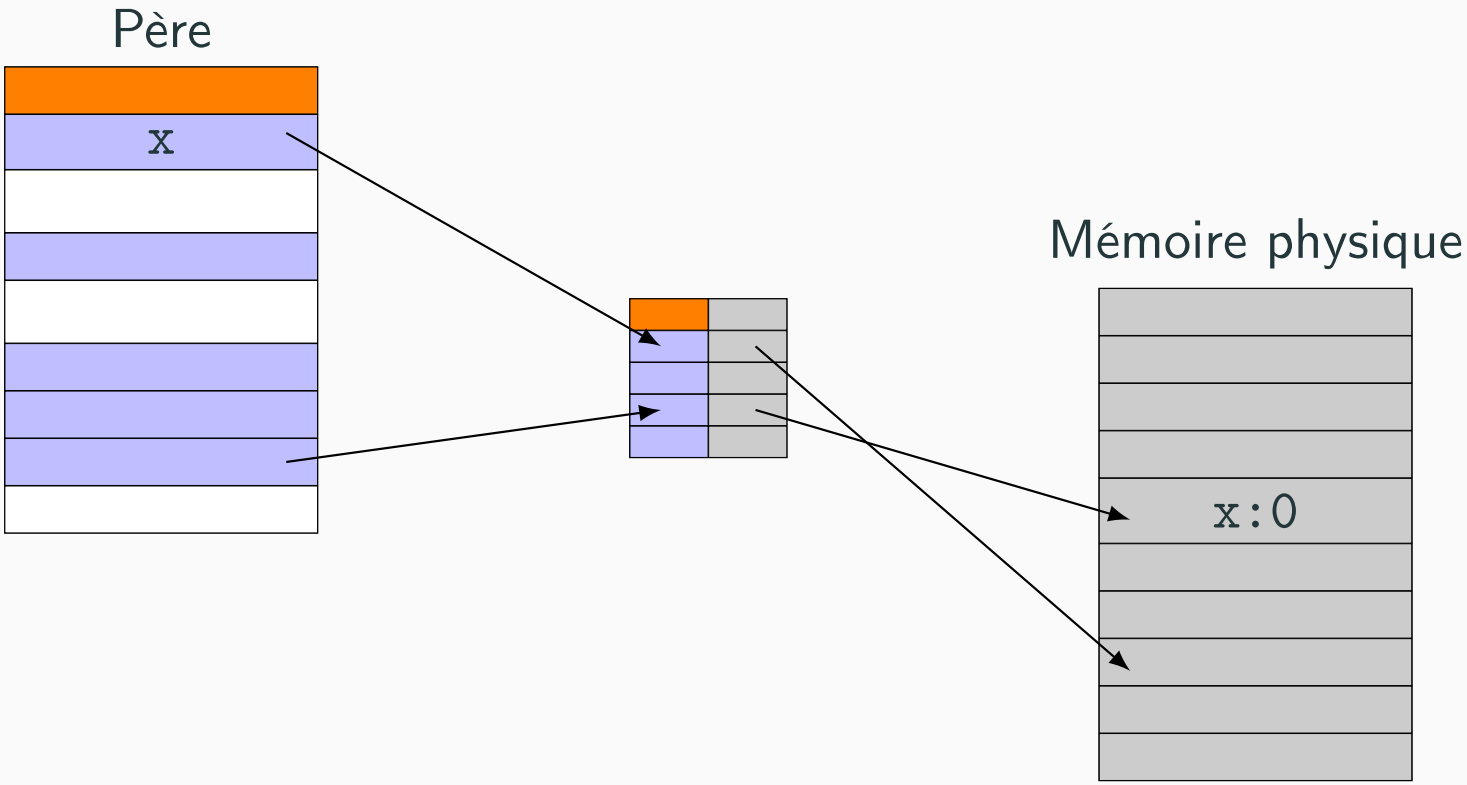
```c
int main(int argc, char *argv[])
{
  pid_t p;
  int x=0;

  p=fork();
  assert(p != -1);

  if (p>0) {
    printf("Je suis le pere %d\n",getpid());
    x=1;
  }else{
    printf("Je suis le fils %d\n",getpid());
    x=2;
  }
  printf("Dans le processus %d, x vaut %d\n",getpid(),x);
}
```

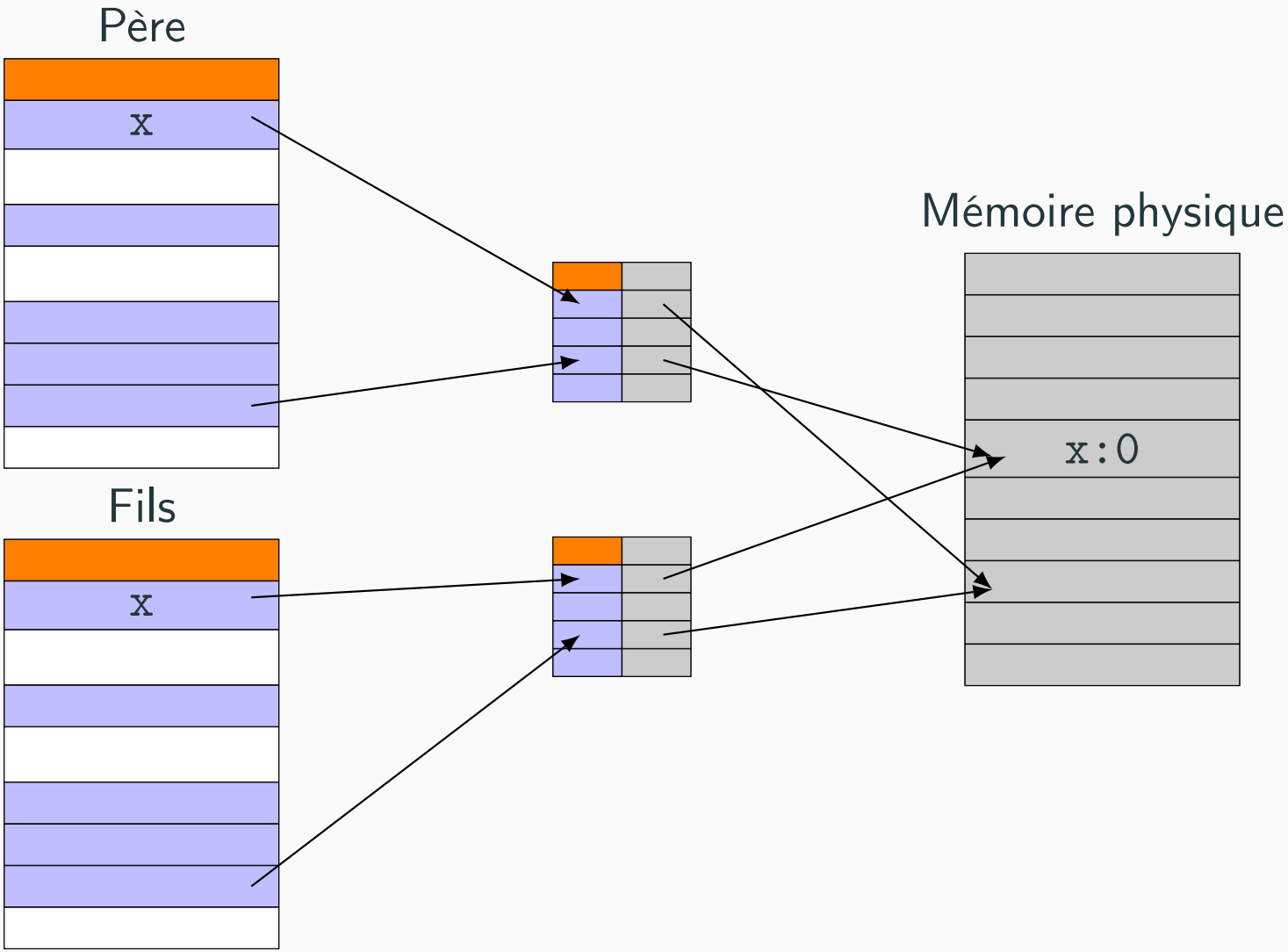
Copy on write

Une variable locale x sur la pile du père initialisé à 0.



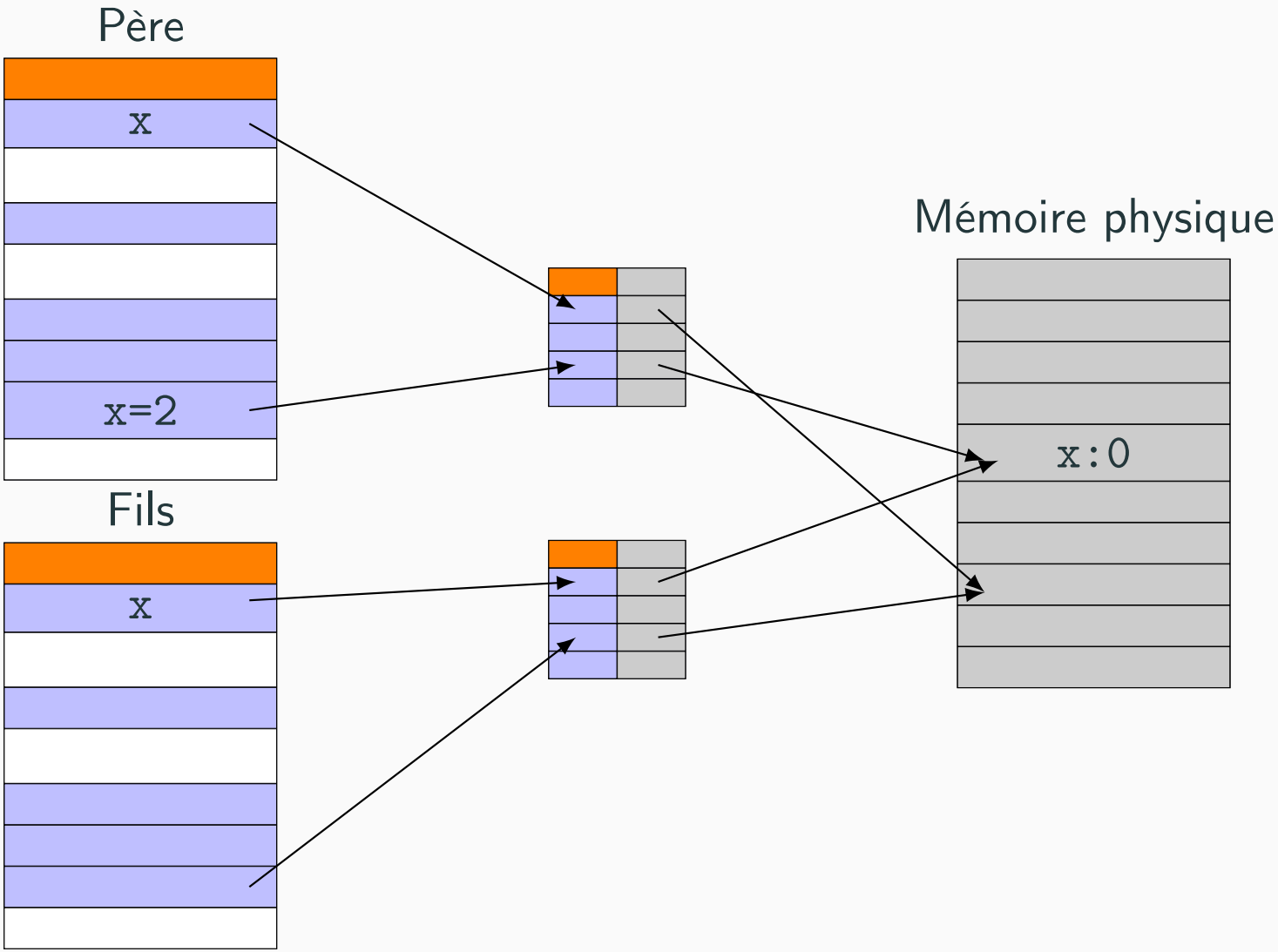
Copy on write

Une variable locale x sur la pile du père initialisé à 0.



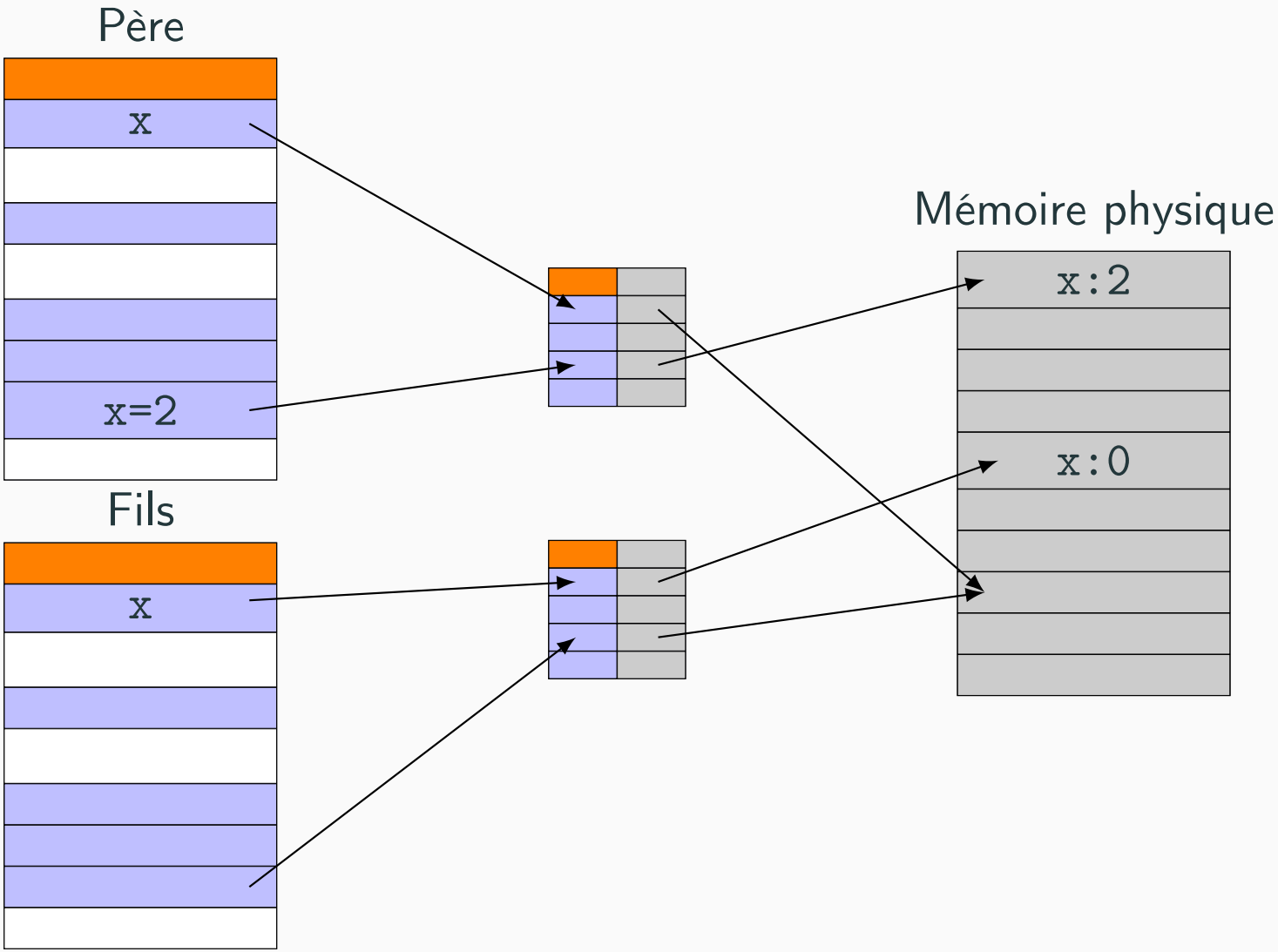
Copy on write

Une variable locale x sur la pile du père initialisé à 0.



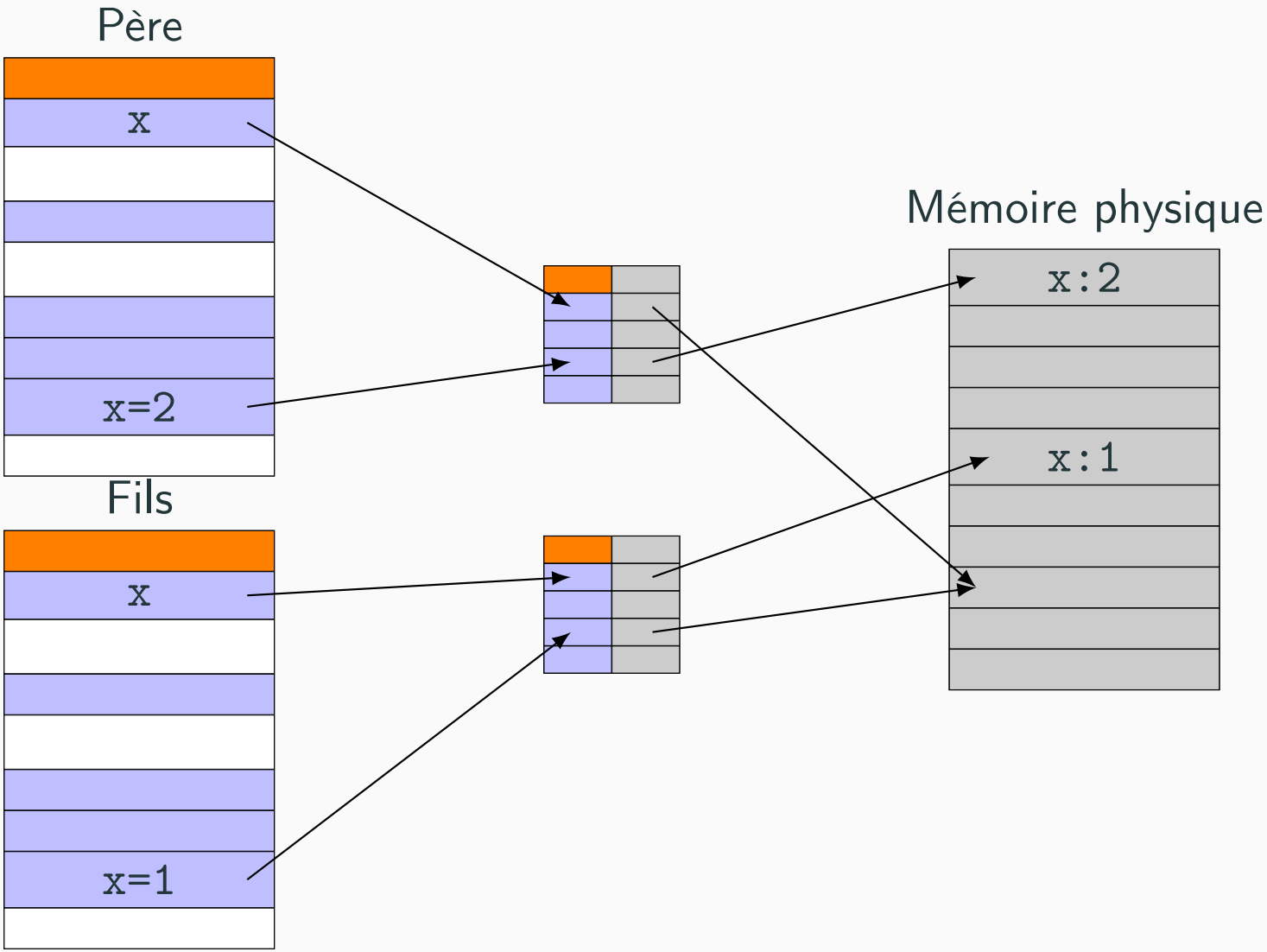
Copy on write

Une variable locale x sur la pile du père initialisé à 0.



Copy on write

Une variable locale x sur la pile du père initialisé à 0.



Cycle de vie

Vie du processus

États du processus

La documentation de la commande ps nous donne :

```
PROCESS STATE CODES
   R  running or runnable (on run queue)
   D  uninterruptible sleep (usually IO)
   S  interruptible sleep (waiting for an event to complete)
   Z  defunct/zombie, terminated but not reaped by its parent
   T  stopped, either by a job control signal or because
      it is being traced
   [...]
```

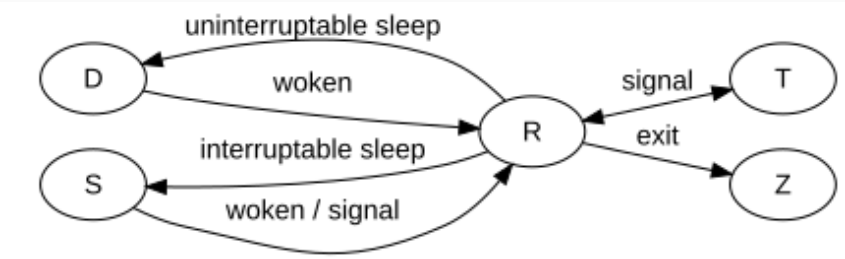
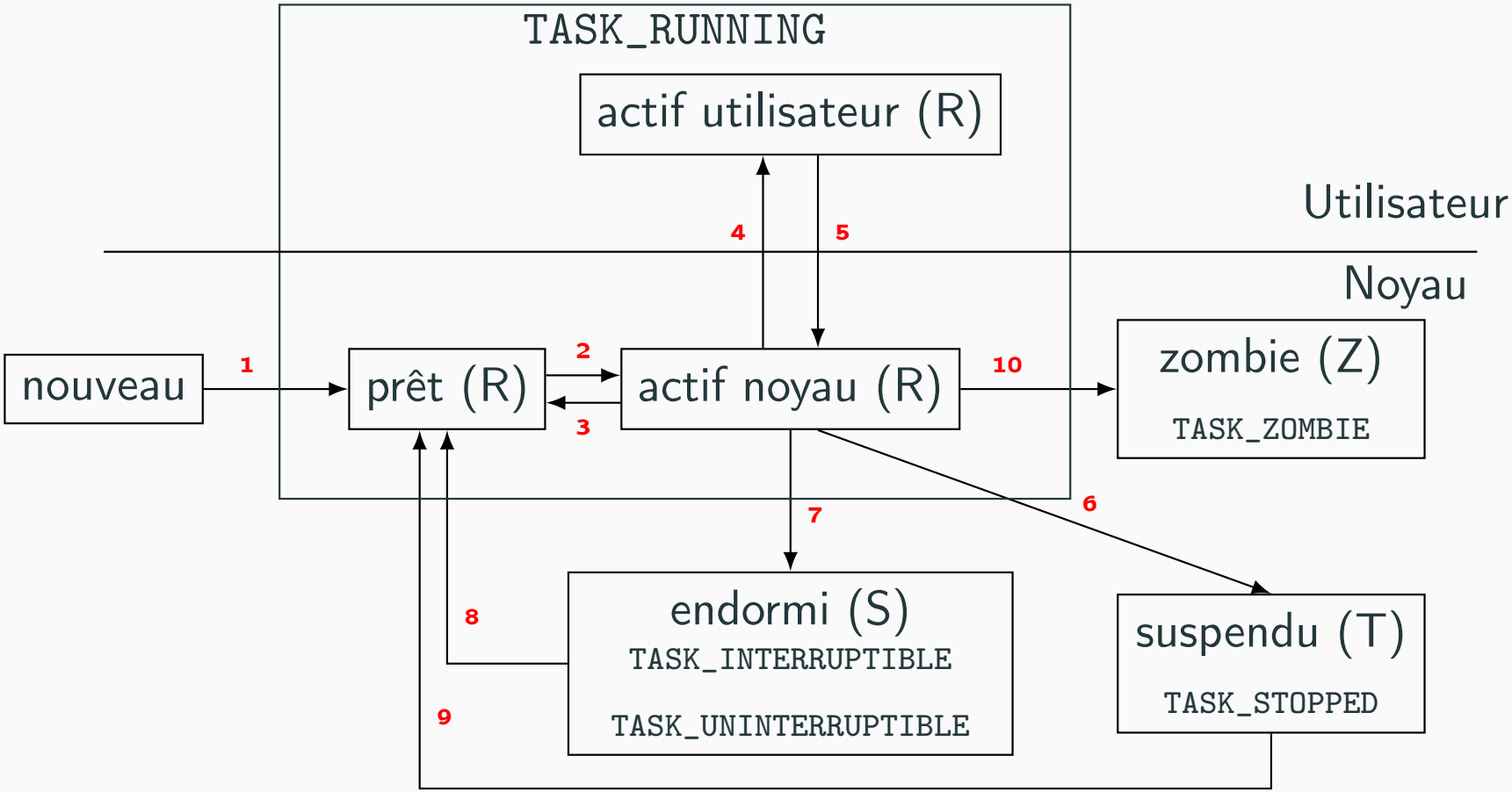


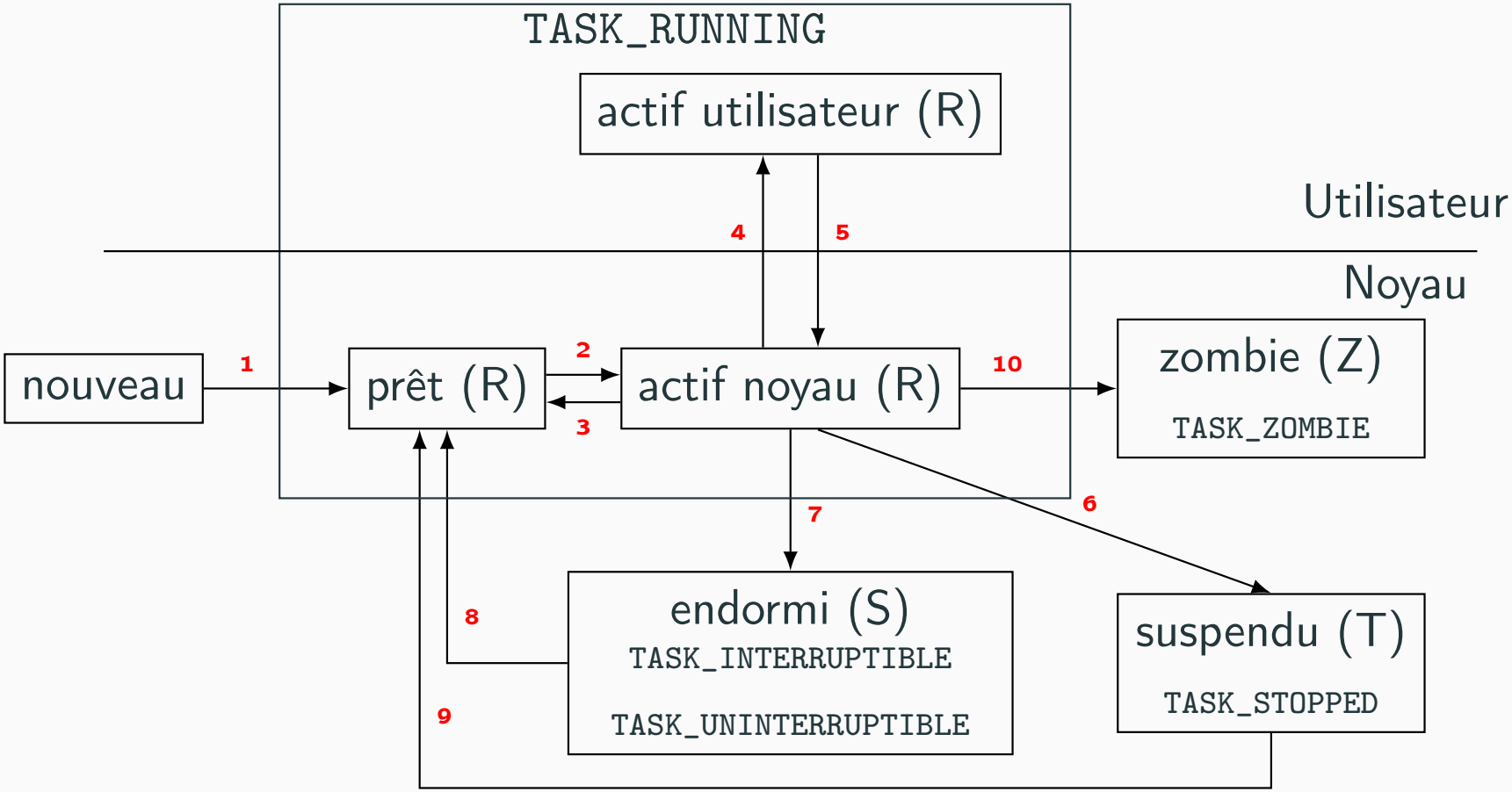
Figure 1: Etats d'un processus

Plus précisémment :



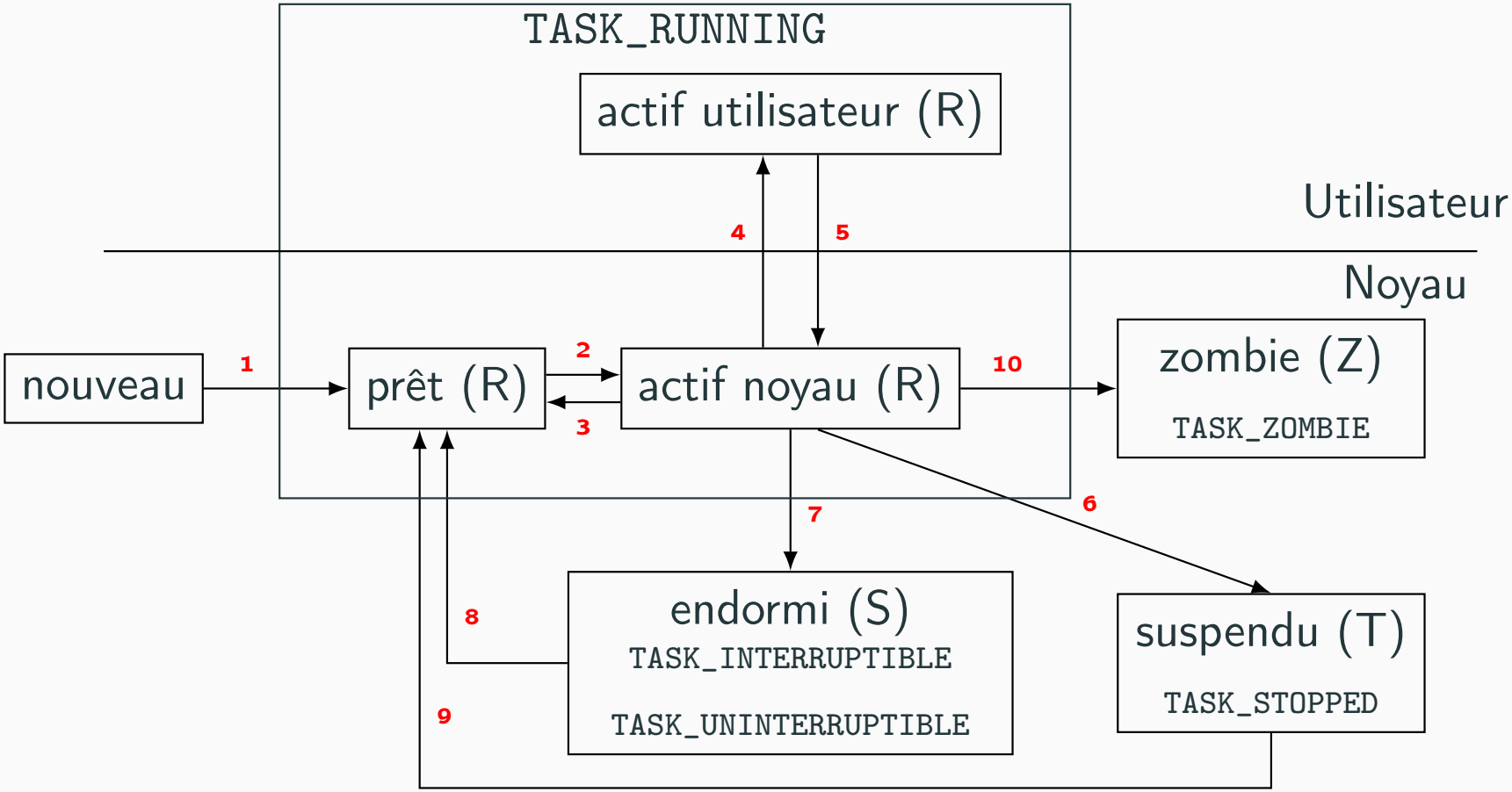
1. Création

Plus précisémment :



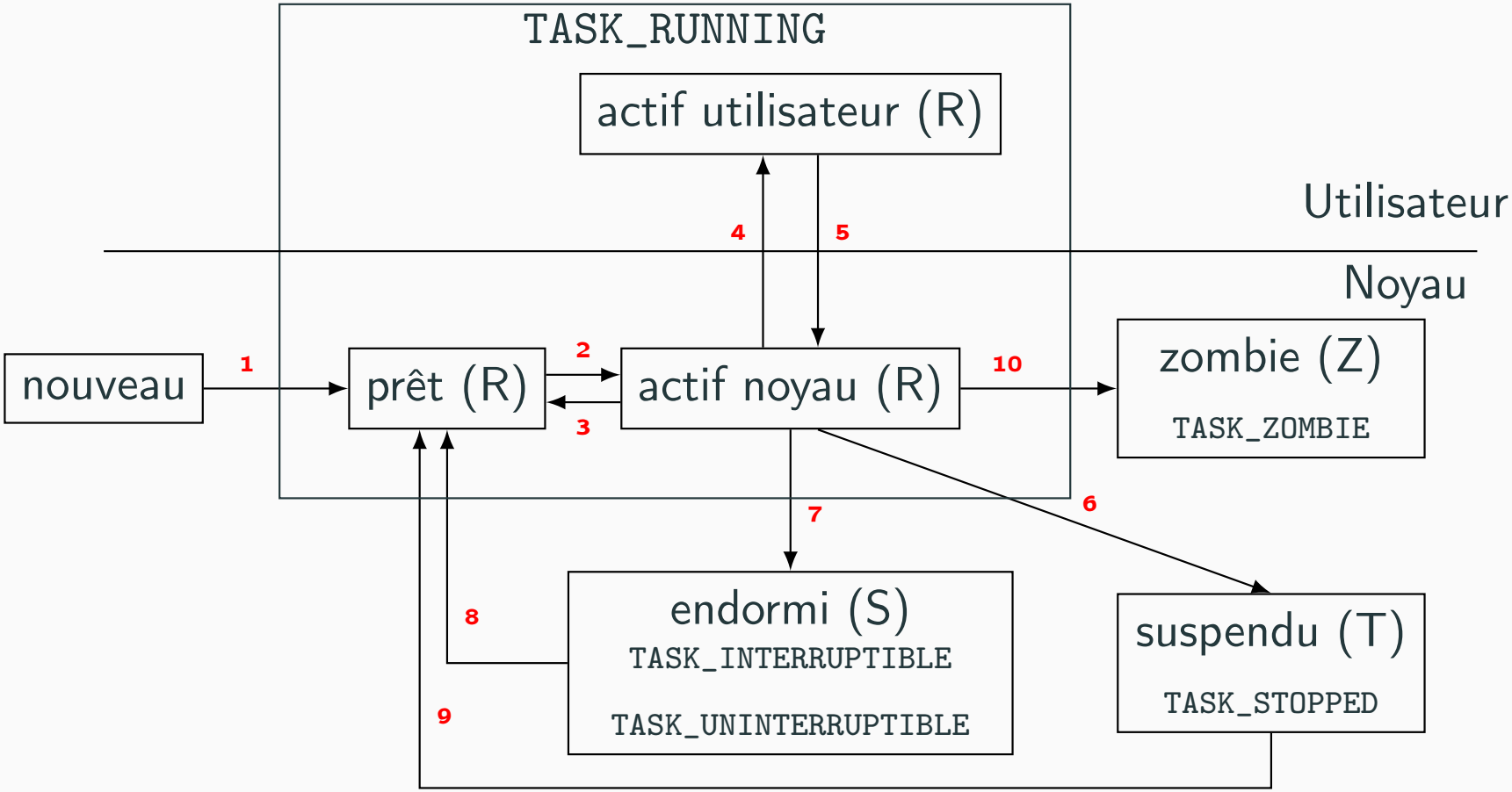
2. Election : le processus est élu par l'ordonnanceur.

Plus précisémment :



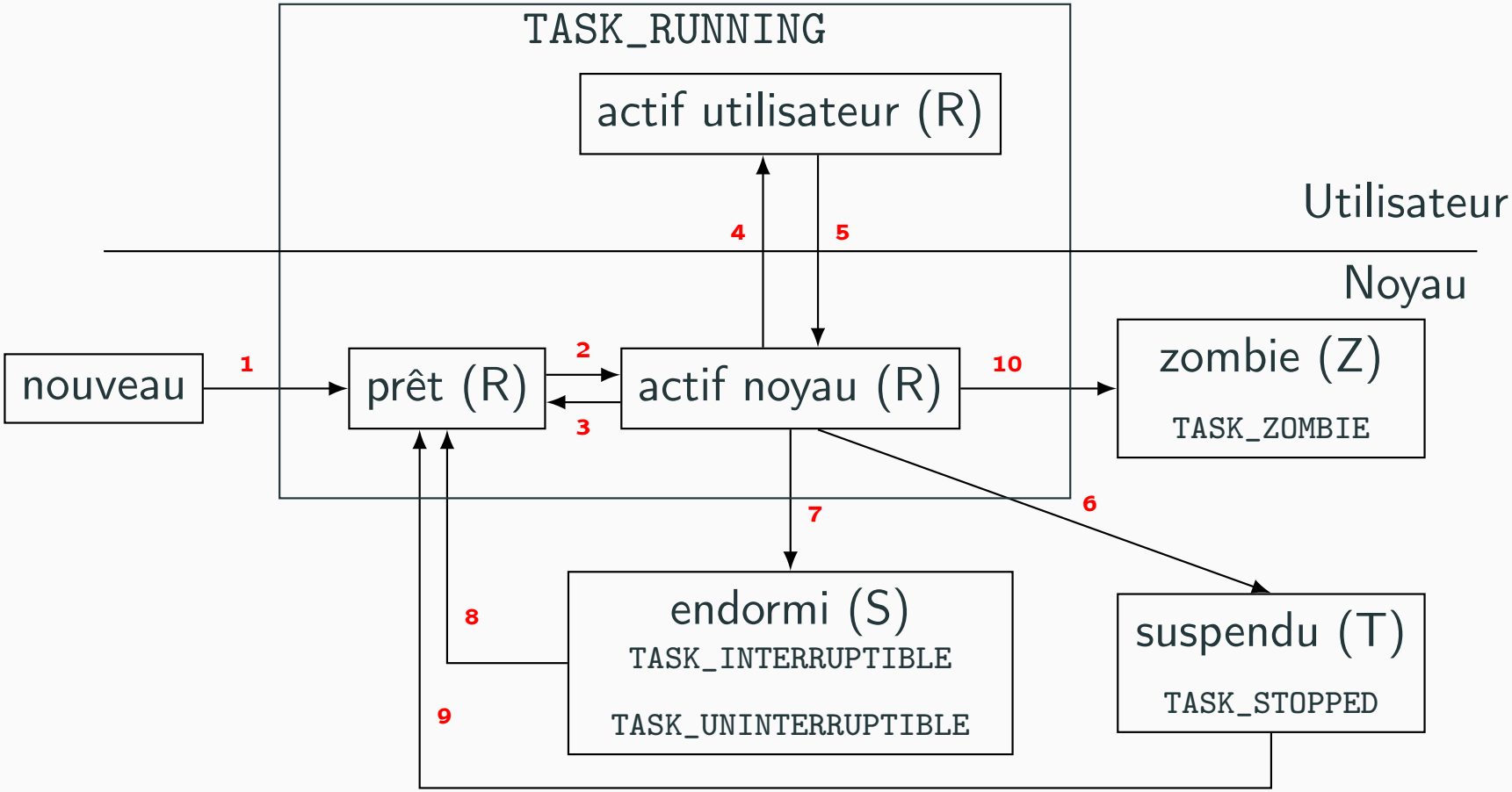
3. Préemption : le processus élu est replacé dans le file des prêts.

Plus précisémment :



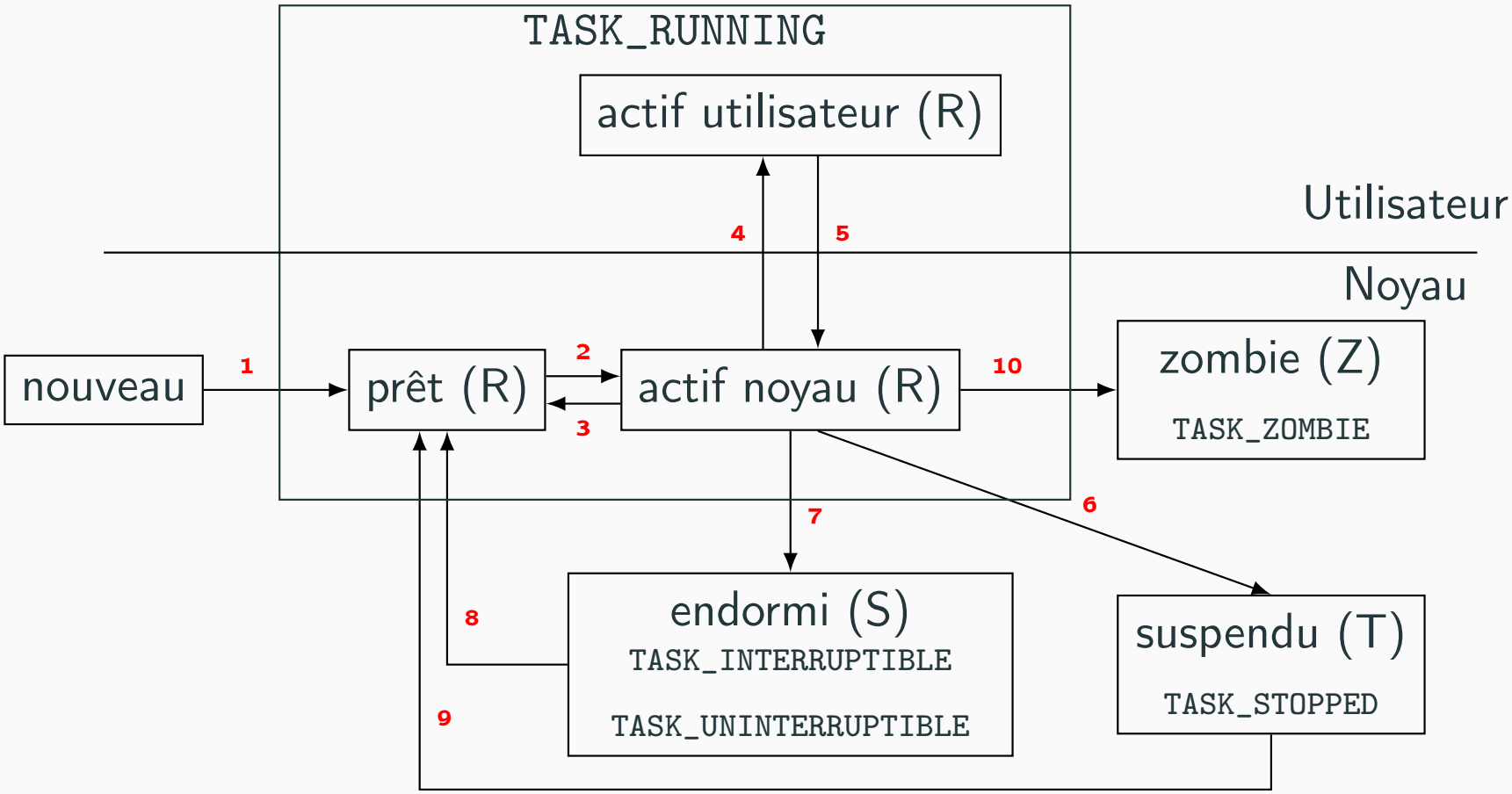
4. Le processus revient d'un appel système, ou d'une interruption.

Plus précisémment :



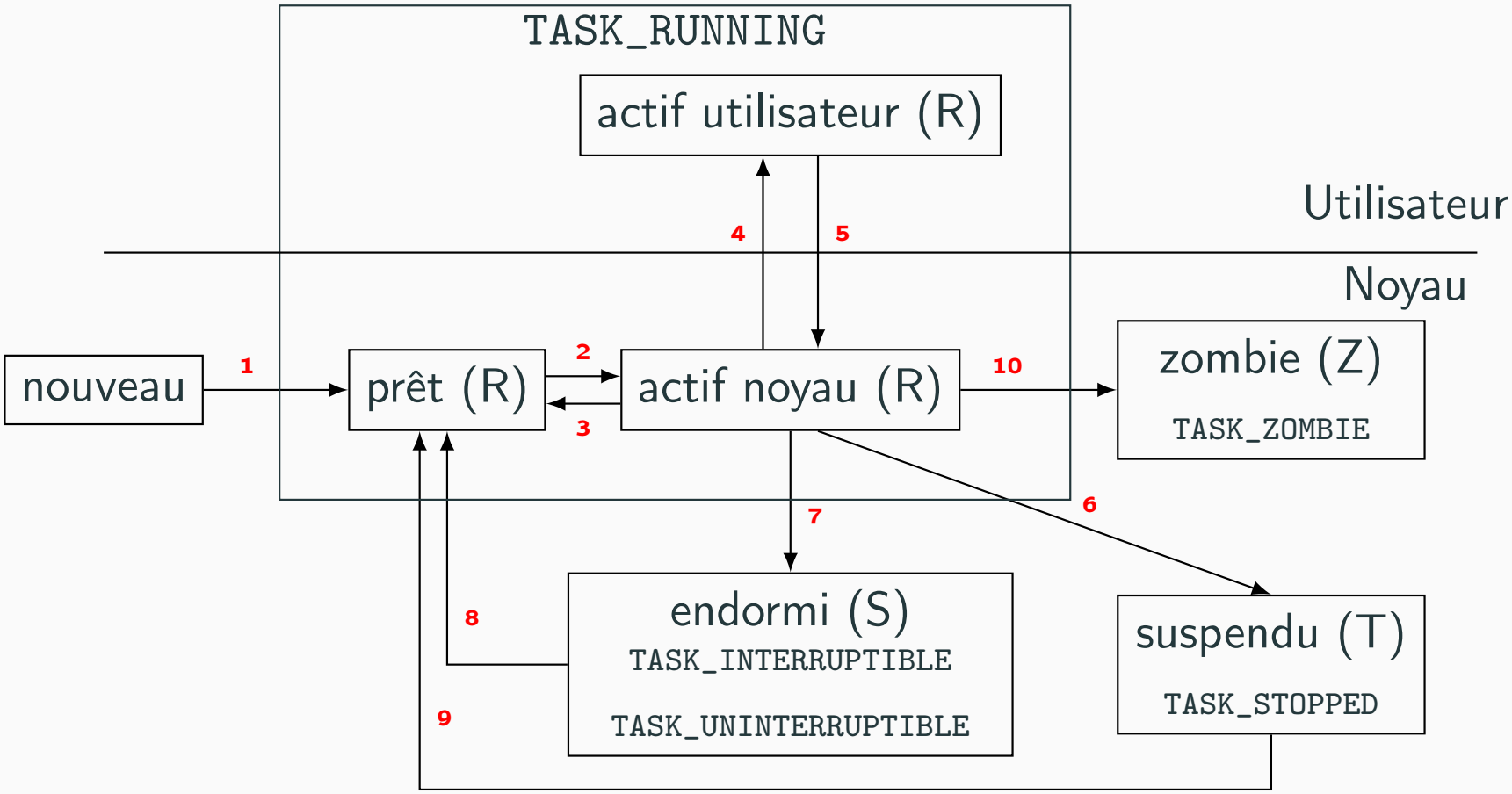
5. Le processus réalise un appel système.

Plus précisémment :



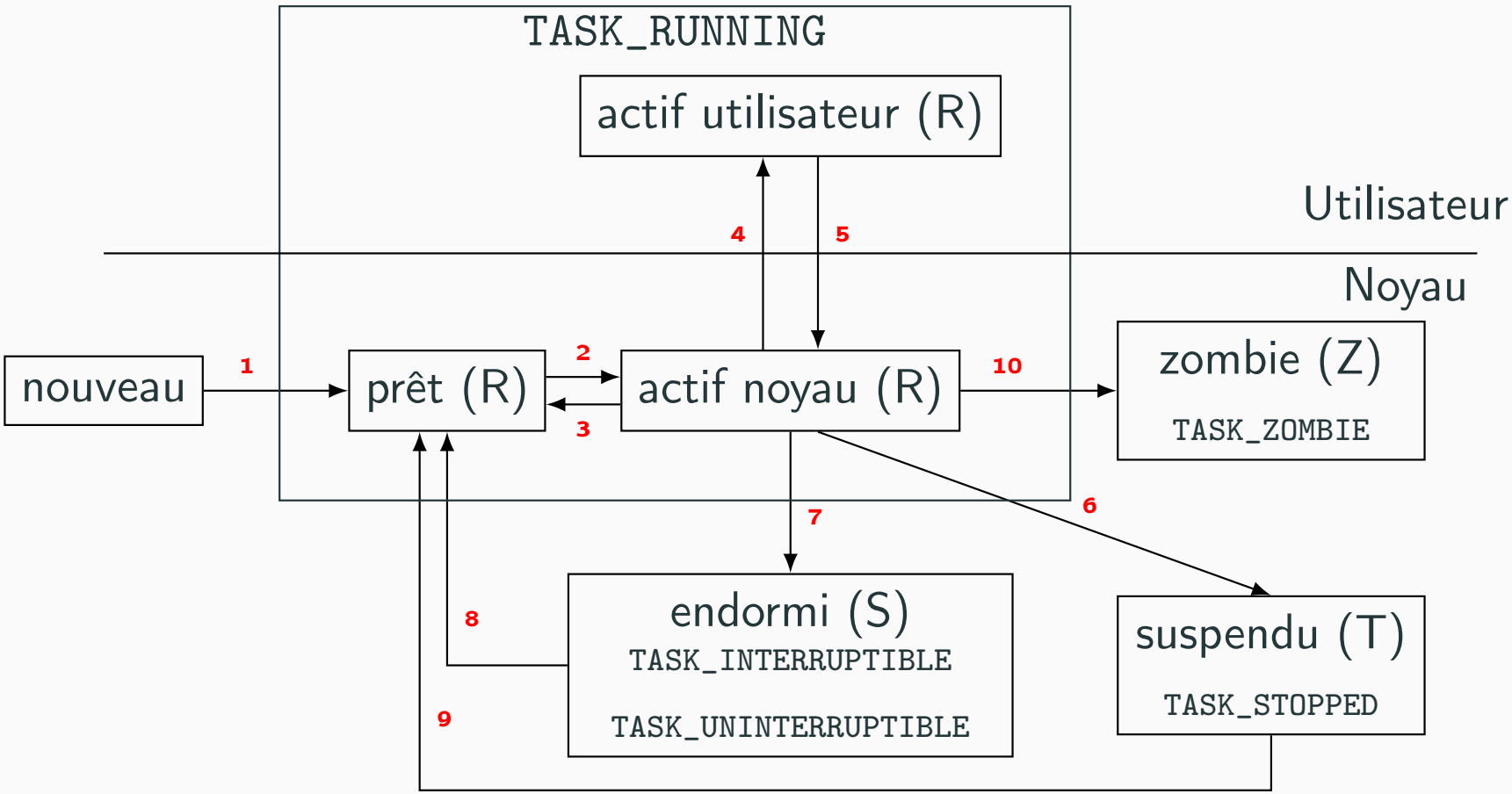
6. Délivrance d'un signal de suspension (`SIGSTOP`, `SIGTSTP`).

Plus précisémment :



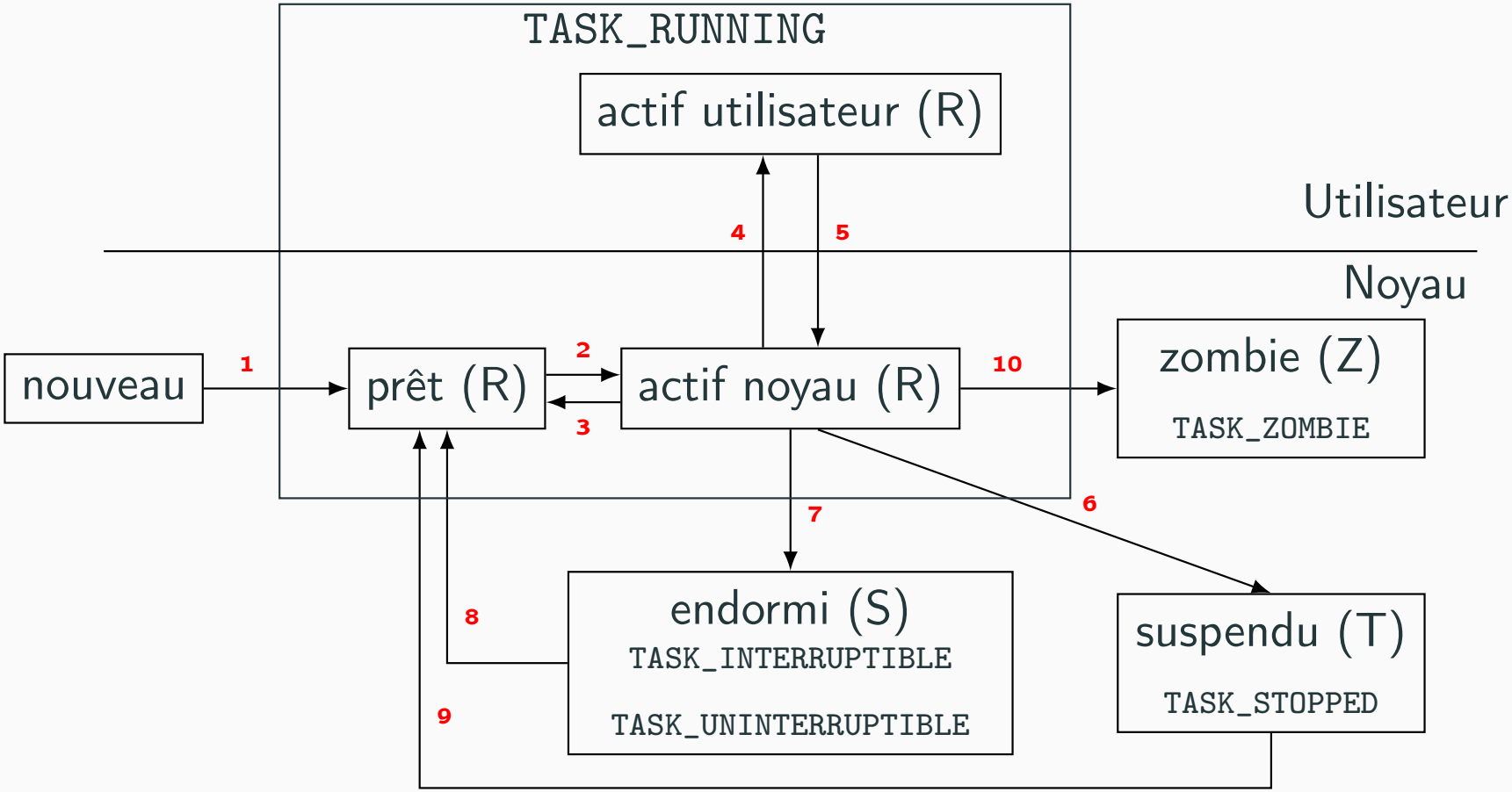
7. Attente d'un événement (`read()`,`wait()`, `etc.`).

Plus précisémment :



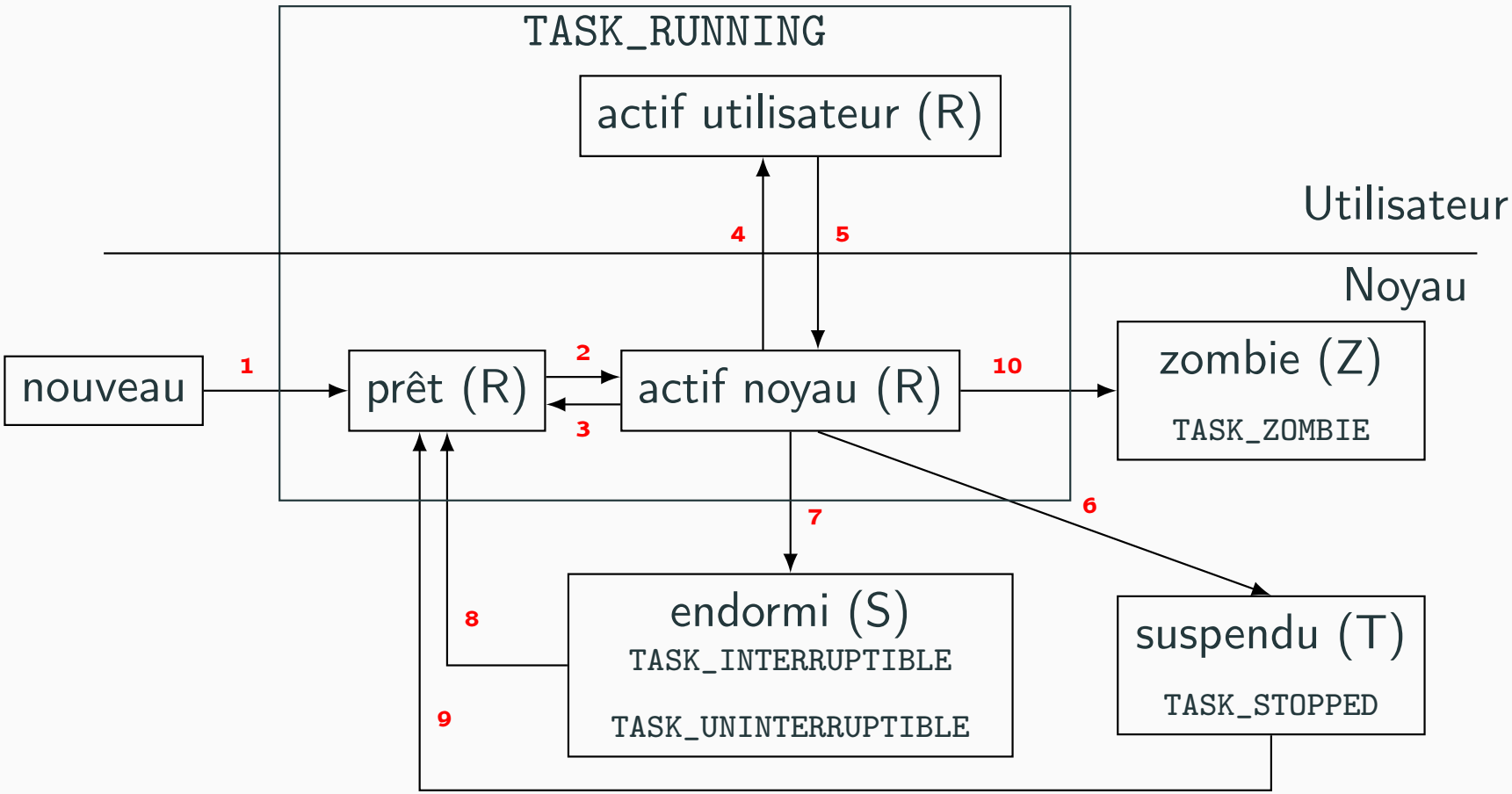
8. L'événement attendu a eu lieu.

Plus précisémment :



9. Réveil du processus par le signal `SIGCONT`.

Plus précisémment :



10. Terminaison.

Mode user vs Kernel

Le mode user a beaucoup moins de privilèges que le mode noyau. Tous les appels systèmes sont effectués en mode noyau.

En C, avec une enveloppe

```c
#include <unistd.h>
int main(int argc, char *argv[])
{
  pid_t = getpid();
  return 0;
}
```

En assembleur

```
    .text
    .globl  my_getpid
    .type   my_getpid, @function

my_getpid:
   pushq   %rbp
   movq   %rsp, %rbp
   movl     $20,%eax
   int $0x80
   popq     %rbp
   ret
```

Interruption, passage en mode noyau, sauvgarde des registres, action, restauration registre, retour en mode user ...

Cycle de vie

Terminaison

Terminaison

- Normale (suicide) : primitive `exit()`

```c
#include <stdlib.h>
void exit(int status);
```

On peut enregistrer une (des) fonction exécutée(s) à la mort normale d'un processus avec :

```c
#include <stdlib.h>
int atexit(void (*function)(void));
```

- Anormale (meurtre) : causée par un signal envoyé par
  - un utilisateur : CTRL-C, commande kill
  - un autre processus : primitive `kill()`
  - le système lui-même : problème d'exécution

Lors de la mort d'un processus,

- les ressources sont libérées.
- les fichiers ouverts sont fermés.
- ses enfants sont adoptés par `init` (pid=1).
- son père reçoit le signal `SIGCHLD`.
- il passe dans l'état zombie.
- le bloc lui correspondant n'est pas libéré ...

... tant que son père n'a pas pris connaissance de sa terminaison.

Comment ?

Primitive `wait()`

```c
#include <sys/wait.h>
pid_t wait(int *stat_loc);
```

Attend (bloquant) la terminaison d'un fils. Retourne sans attendre si un fils est déjà mort.

A la terminaison d'un fils, `wait` renvoie son pid, et place dans `*stat_loc` (si non NULL) des informations sur la terminaison du fils.

Des macros permettent d'interpréter cette valeur :

- `WIFEXITED(val)` : 1 si terminaison normale, 0 sinon.
- `WEXITSTATUS(val)` : code de retour (sur 8 bits) du processus pour une terminaison normale.
- `WIFSIGNALED(val)` : 1 si terminaison anormale (signal), 0 sinon.
- `WTERMSIG(val)` : numéro du signal qui a causé la mort du processus.

Primitive wait()

```c
int main(int argc, char *argv[])
{
  pid_t p,n;
  int status;
  assert((p=fork()) != -1);

  if (p==0){// fils
    printf("%d child exit\n",getpid());
    //kill(getpid(),SIGTERM);  <- signal
    exit(random()%256);
  }
  n=wait(&status);
  if (WIFEXITED(status)) // terminaison normale
    printf("child %d exit status %d\n",n, WEXITSTATUS(status));
  else if (WIFSIGNALED(status)) // terminaison anormale
    printf("child %d killed by signal %d\n", n,WTERMSIG(status));
}
```

Recouvrement

Primitives `exec()`

```c
#include <unistd.h>
int execl(const char *pathname, const char *arg, ...
                    ,(char  *) NULL);
```

Possibilité de charger un programme exécutable en mémoire en remplaçant l'image du processus appelant, et de l'exécuter. Famille de primitives `exec()`

En cas de succès, l'espace d'adressage du processus est écrasé avec l'exécutable, et l'exécution reprend à son point d'entrée.

Sont perdus (entre autres) :

- les signaux pendants
- les handlers, masque de signaux.
- fichiers mappés, verrous
- etc.

Sont conservés :

- pid
- priorité
- fichiers ouverts si l'attribut close-on-exec n'est pas poitionné sur le descripteur de fichier.
  (`O_CLOEXEC` à l'ouverture avec `open`, où `FD_CLOEXEC` avec `fnctl`)
- UID, etc.

```c
int main(int argc, char *argv[])
{
  pid_t p = fork();
  assert(p != -1);

  if (p==0){
    int ret=execl("/usr/bin/ls","ls","-l",NULL);
    if (ret==-1) {perror("erreur exec");
      exit(2);
    }
  }else{
    wait(NULL);
  }
}
```

Sécurité

Les primitives `execlp()` et `execvp` utilise le PATH. En modifiant la variable d'environnement, on peut injecter son propre code ...

Contrôle

Sessions et groupes

Les processus sont partionnées en groupe, eux-même partionné en session. But ? job control, E/S et envoie groupé de signaux

- un processus appartient à un groupe unique.
- un groupe appartient à une session unique.
- une session peut être controllée par un terminal `tty` device.

- processs group id : pid du leader du groupe lorsqu'il existe.
- session id : pid du leader de session
- foreground/background process group. un seul au plus en avant plan.

Terminal de contrôle

- Une session peut avoir un terminal de contrôle. Il y a alors un seul groupe en avant plan, les autres en arrière plan.
- des raccourcis claviers permettent depuis le terminal d'envoyer des signaux au groupe en avant plan (ctrl-C ou ctrl-Z)
- `SIGHUP` est envoyé à tous les processus d'une session attachée à un terminal à la mort de son leader, où lors d'une déconnexion.
- Si le processus leader d'un groupe est déconnecté, `SIGHUP` est envoyé à tout le groupe.

```
$stty -a
speed 38400 baud; rows 67; columns 239; line = 0;
intr = ^C; quit = ^\; erase = ^?; kill = ^U; eof = ^D; eol = <undef>; eol2 = <undef>; sw
-parenb -parodd -cmspar cs8 -hupcl -cstopb cread -clocal -crtscts
-ignbrk -brkint -ignpar -parmrk -inpck -istrip -inlcr -igncr icrnl ixon -ixoff -iuclc -i
opost -olcuc -ocrnl onlcr -onocr -onlret -ofill -ofdel nl0 cr0 tab0 bs0 vt0 ff0
isig icanon iexten echo echoe echok -echonl -noflsh -xcase -tostop -echoprt echoctl echo
```

Groupes

Chaque session est partionnée en groupe.

- Chaque session a un groupe au plus en premier plan
- la frappe des caractères intr, quit et susp provoque respectivement l'envoi du signal `SIGINT`,`SIGQUIT` et `SIGTSTP` à tous les processus de ce groupe.
- Les groupes en arrière plan. Les processus n'ont pas accès au terminal de contrôle et ne recoivent pas les signaux précédents.
- `SIGTTIN` et `SIGTTOU` envoyé à un processus en arrière plan qui tente de lire ou écrire sur le terminal.

```
[denis@portabledenis ~]$ cat | cat | cat
```

```
[denis@portabledenis ~]$ ps -t /dev/pts/2 -o pid,ppid,pgid,tpgid,sess,stat,comm --forest
  PID  PPID  PGID TPGID  SESS STAT COMMAND
 7005  7000  7005  7297  7005 Ss   bash
 7297  7005  7297  7297  7005 S+    \_ cat
 7298  7005  7297  7297  7005 S+    \_ cat
 7299  7005  7297  7297  7005 S+    \_ cat
```

Expliquez.

Démonification

```c
int main (void)
{
  pid_t pid;
  int i;
  /* create new process */
  pid = fork ();
  if (pid == -1) return -1;
  else if (pid != 0)
    exit (EXIT_SUCCESS);
  /* create new session and process group */
  if (setsid () == -1) return -1;
  /* set the working directory to the root directory */
  if (chdir ("/") == -1) return -1;
  /* close all open files--NR_OPEN is overkill, but works */
  for (i = 0; i < NR_OPEN; i++)
    close (i);
  /* redirect fd's 0,1,2 to /dev/null */
  open ("/dev/null", O_RDWR); /* stdin */
  dup (0); /* stdout */
  dup (0); /* stderror */

  /* do its daemon thing... */
  return 0;
}
```

```c
#include <unistd.h>
int daemon(int nochdir, int noclose);
```

Ordonnancement

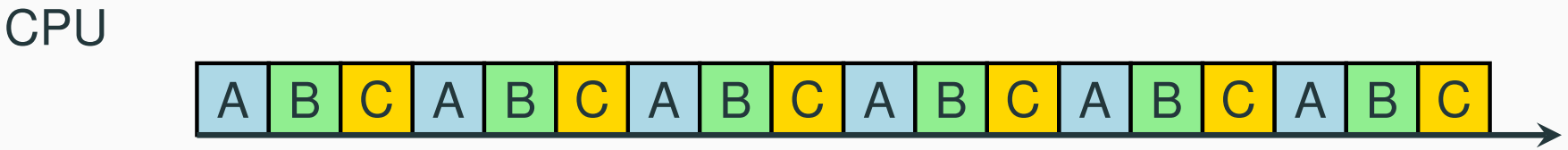
Ordonnancement

Généralités

Multitâche

- Faculté du système à exécuter plusieurs processus "à la fois".
- Pseudo prallèlisme et entrelacement

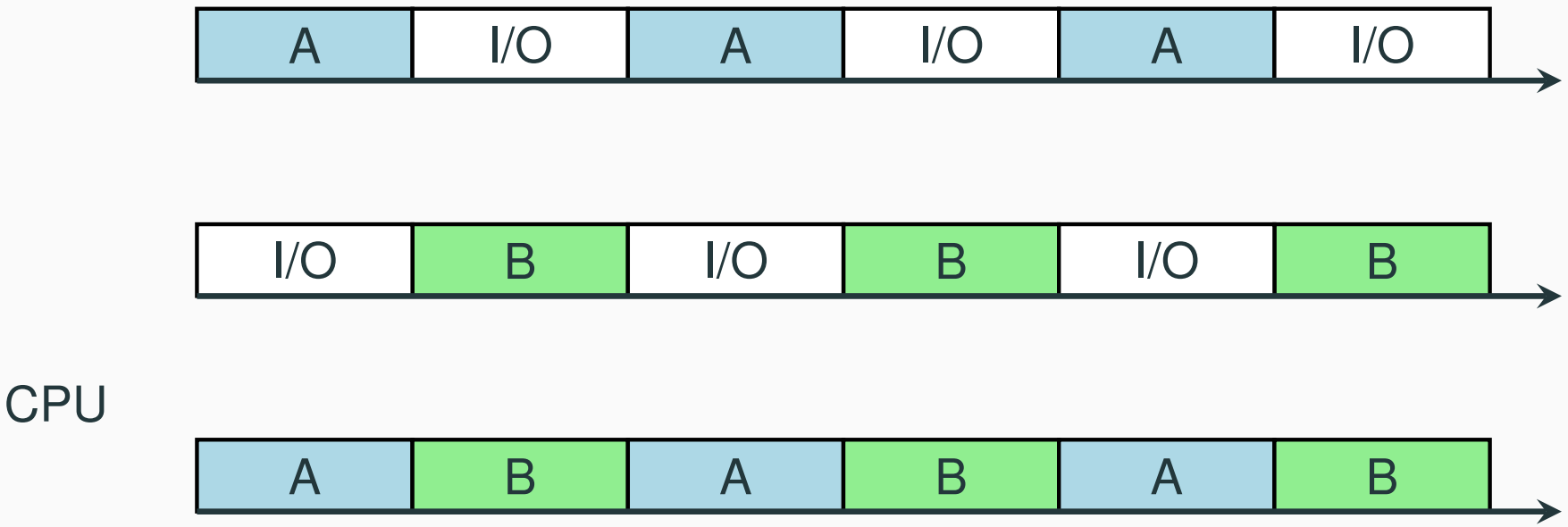
CPU time sharing



Remarque : l'entrelacement doit être "assez fin" pour ne pas se "voir".

Multitâche

- Empiriquement, un processus alterne des phases de calculs (CPU burst) et des phases d'E/S (I/O burst).



- Evidemment, on peut avoir des processus déséquilibrés (plus de calculs ou d'E/S).

E/S

E/S

- latence d'accès au matériel : disque, réseau...
- lenteur de l'utilisateur d'un programme interactif
- synchronisation avec d'autres programmes

Mauvaise solution : attente active (polling)

- difficile pour le programmeur d'application
- temps processeur gâché à attendre

Bonne solution : attente passive

- plus facile pour le développeur
- meilleur taux d'utilisation du CPU
- masquage (= recouvrement) des latences

besoin d'un mécanisme pour se partager le(s) processeur(s)

Commutation de Contexte

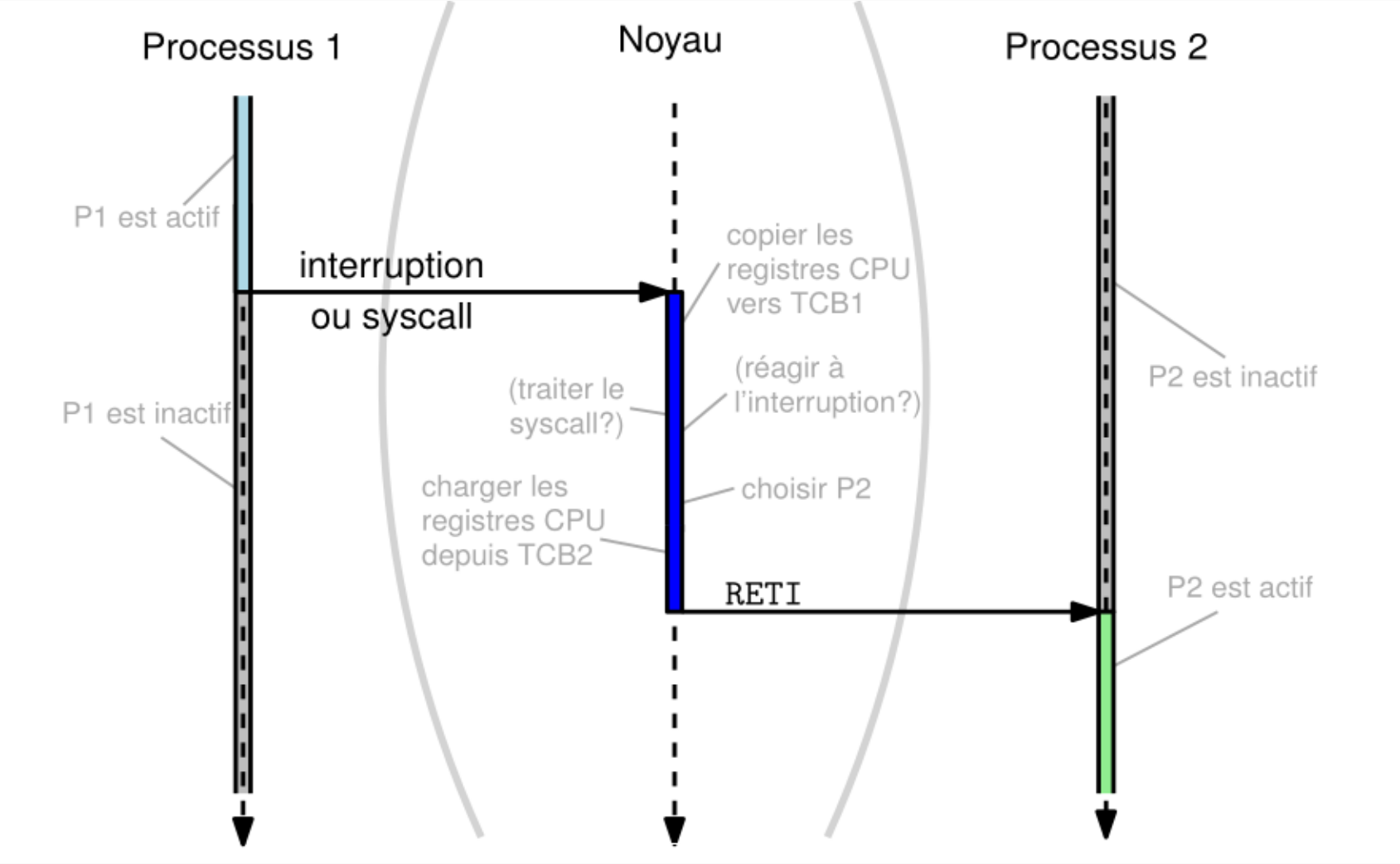


Figure 2: Commutation de contexte

Le remplacement du processus en exécution a un coût !

Commutation de contexte

- Exécution de la routine d'ordonnancement
- Sauvegarde du contexte (registres + PC)
- Chargement d'un nouveau contexte

Structures du noyau

- ProcessCB : identité, mémoire, fichiers ouverts, statistiques, etc.
- ThreadCB : fil d'exécution lié à un processus : registres, PC, SR, pile, etc

Ordonnanceur

Les objectifs d'un ordonnanceur d'un système multi-utilisateur sont, entre autres :

- S'assurer que chaque processus en attente d'exécution reçoive sa part de temps processeur.
- Minimiser le temps de réponse.
- Utiliser le processeur à 100%.
- Utilisation équilibrée des ressources.
- Prendre en compte des priorités.
- Être prédictible.

Les objectifs ne sont pas les mêmes selon qu le système est intéractif, temps réel, traitement par lots, etc.

Critères d'évaluation

- CPU utilization rate : fraction du temps où le CPU est productif i.e. occupé à exécuter du code applicatif (vs noyau, ou CPU idle)
- Throughput = débit : nombre de tâches terminées par unité de temps.
  attention : n'a de sens que si les «tâches» peuvent «terminer»
- Non-Starvation : absence garantie de risque de famine
- Turnaround time (tps de séjour) : durée entre arrivée et terminaison
  n'a de sens que si les «tâches» peuvent «terminer»
- Waiting time (temps d'attente) : durée passée dans la ready queue
- Response time : durée entre arrivée et «réponse»

Ordonnancer

Formulation

- étant donnés K processus prêts,
- étant donnés N processeurs disponibles,

décider quel processus exécuter sur chaque processeur.

Remarque : quand ordonnancer ?

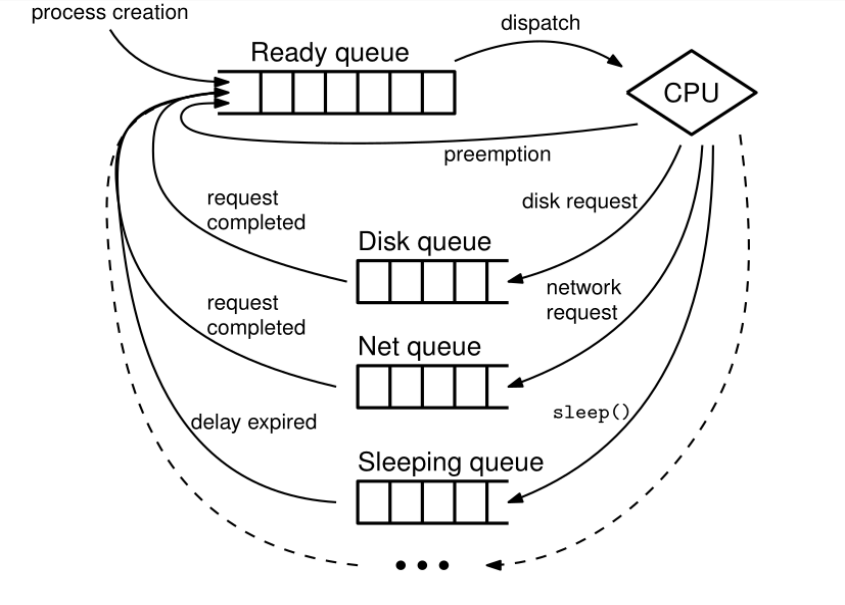
- le processus qui s'exécute se termine ou se bloque.
- un processus passe dans l'état prêt.
- le processus en exécution a eu le processeur "trop longtemps".

Principes généraux

Préemptif/Coopératif

- Préemptif : le noyau peut remplacer un processus en exécution par un autre processus prêt.
  - interruption de timer régulière pour assurer la préemption.
  - le noyau garde le "contrôle de la machine".
  - coûteux.
- Coopératif : les processus rendent explicitement la main au noyau.

File d'attentes/priorités



- Les PCB des processus prêts forment la Ready Queue
- Rôle du scheduler : choisir un PCB parmi la Ready Queue
- Processus bloqués : transférés dans une autre file d'attente
  - une Device Queue par périphérique d'entrées-sorties
  - une file d'attente pour les processus endormis
  - ... une file pour chaque autre raison d'être Blocked
- Priorité
  - Type de processus (système ou utilisateur, E/S ou calcul)
  - Priorité fixée par l'utilisateur
  - Variable au cours de l'exécution (règles de l'OS)

Ordonnancement

Algorithmes non prémptifs

FCFS

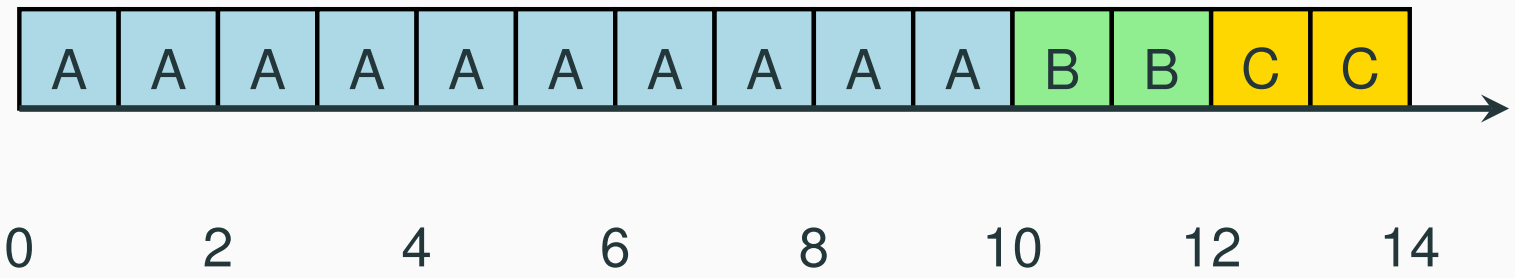
First Come, First Served

Principe

Premier arrivé, premier servi

Proc	Date	Durée
A	0	10
B	1	2
C	1	2

CPU



SJF

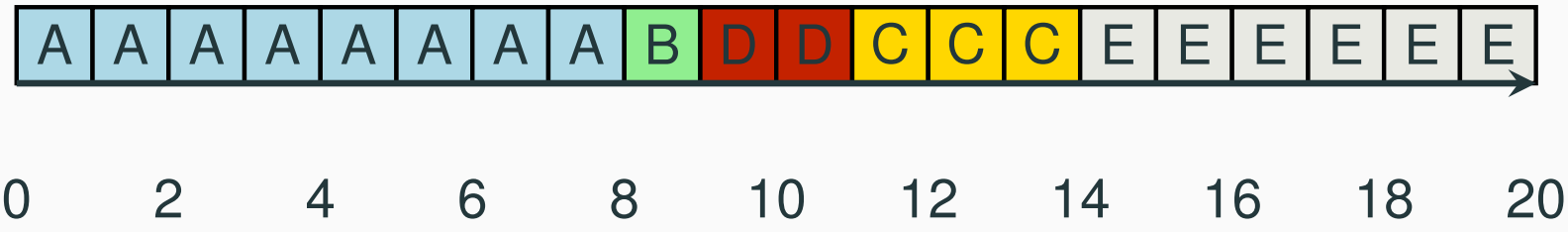
Shortest Job First

Principe

On exécute le plus court d'abord (version non préemptive)

Proc	Date	Durée
A	0	8
B	1	1
C	2	3
D	3	2
E	4	6

CPU



Ordonnancement

Algorithmes préemptifs

SRTF(Q)

Shortest Remaining Time First

Principe

C'est la version préemptive de SJF. Tous Q quantum de temps, on élit la tâche la plus courte.

- On choisit le processus dans la file d'attente selon le plus court SRT.
- Le processus est exécuté au plus pendant un quantum de temps.
- Si, après ce quantum, il n'est pas terminé, il est ajouté à la file d'attente, et son nouveaux SRT' est calculé selon la formule : SRT' = SRT - quantum

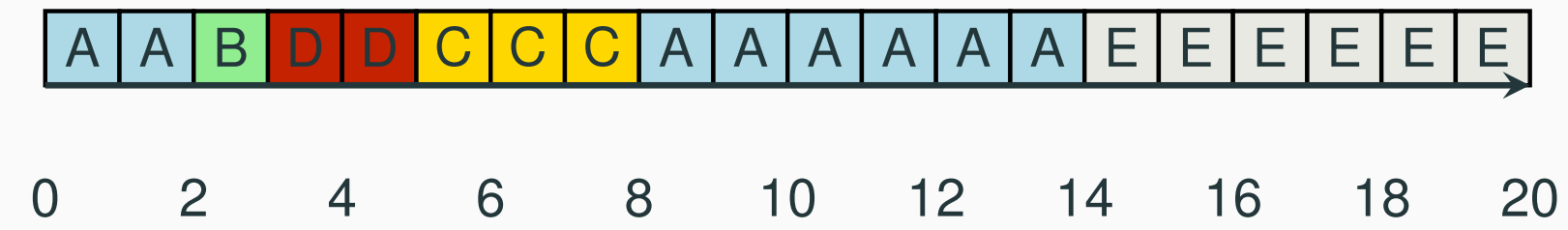
Paramètre : le quantum du temps.
Analogue du SJF, algorithme théorique car les temps d'éxécution n'est pas en général connu.

Exemple

Avec Q=2

Proc	Date	Durée
A	0	8
B	1	1
C	2	3
D	3	2
E	4	6

CPU

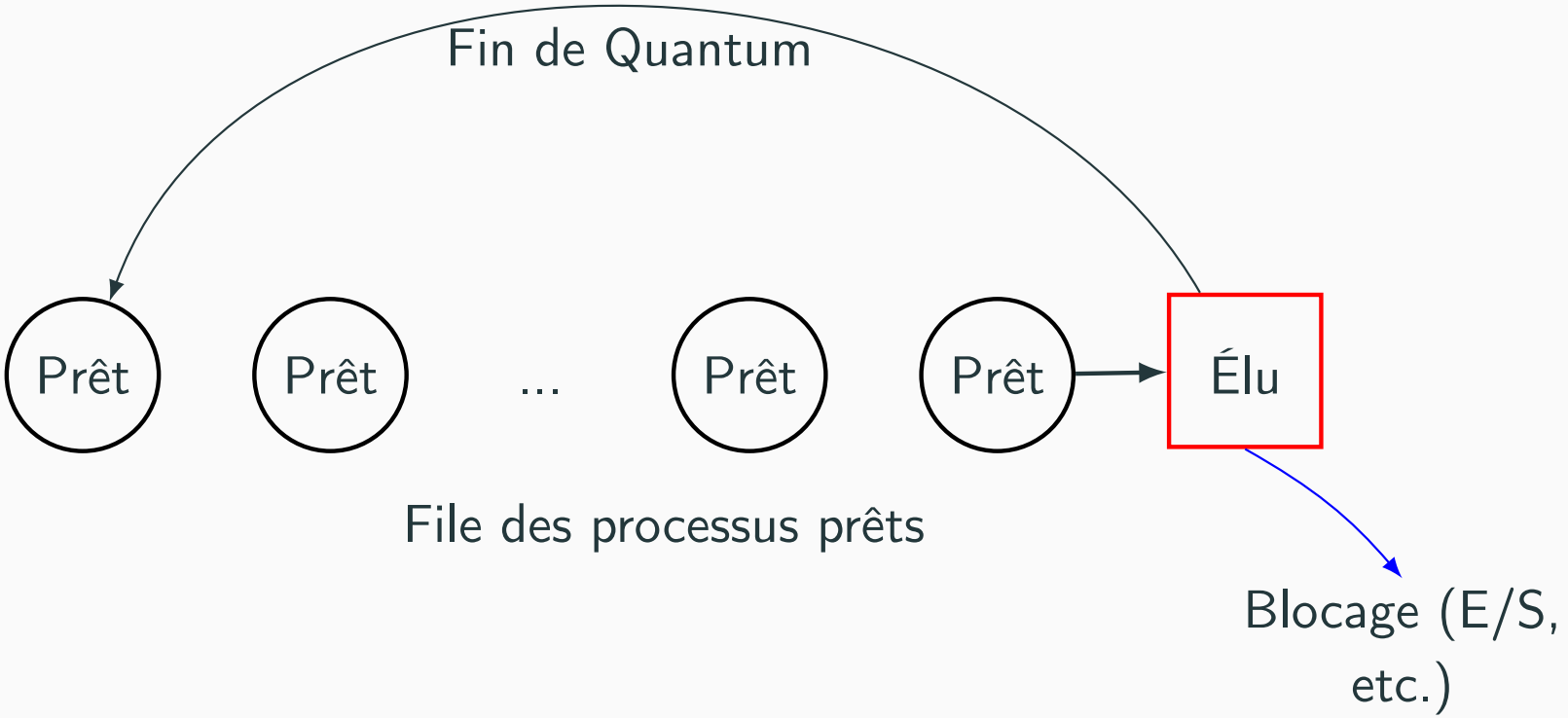


Round-robin

- Quantum = durée maximale d'activité continue.
- Préemption du processeur au profit d'un autre processeus :
  - soit en fin de quantum ;
  - soit sur blocage du processus actif.
- Adaptation possible de la durée du quantum au profil d'éxécution (calcul ou E/S).
- À l'expiration du quantum, mise en fin de file des prêts.

Choix du Quantum ?

- bref : overhead
- long : temps de réponse imporant


Fin de Quantum
Prêt
Prêt
...
Prêt
Prêt
Élu
File des processus prêts
Blocage (E/S, etc.)

Exemple

Avec Q=2

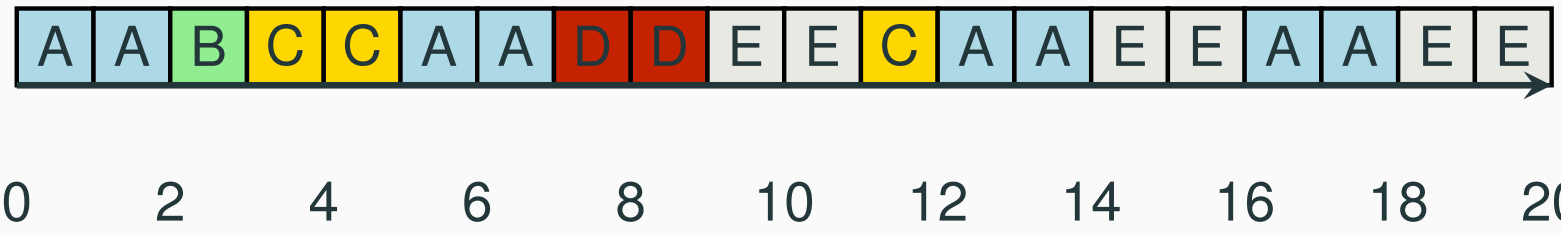
Proc	Date	Durée	Déb	Fin	Tps Sej	Att	Pénal
A	0	8					
B	1	1					
C	2	3					
D	3	2					
E	4	6					

Exemple

Avec Q=2

Proc	Date	Durée	Déb	Fin	Tps Sej	Att	Pénal
A	0	8	0	18	18	10	2.25
B	1	1	2	3	2	1	2.00
C	2	3	3	12	10	7	3.33
D	3	2	7	9	6	4	3.00
E	4	6	9	20	16	10	2.67

CPU



Ordonnancement

Linux

Classes d'ordonnancement

```
$ chrt -m
propriété min/max SCHED_OTHER  : 0/0
propriété min/max SCHED_FIFO  : 1/99
propriété min/max SCHED_RR  : 1/99
propriété min/max SCHED_BATCH  : 0/0
propriété min/max SCHED_IDLE  : 0/0
propriété min/max SCHED_DEADLINE  : 0/0
```

- Stratégies temps réel
  - `SCHED_FIFO` (FIFO real-time)
  - `SCHED_RR` (Round-Robin real-time)
- Stratégies normales
  - `SCHED_OTHER`
  - `SCHED_BATCH`
  - `SCHED_IDLE`

Stratégies temps réels

Les processus en temps réel sont ordonnancés en premier et les processus normaux sont ordonnancés une fois que tous les threads en temps réel ont été ordonnancés.

Les stratégies en temps réel sont utilisées pour des tâches pour lesquelles le temps est critique, des tâches devant être effectuées sans interruption.

- `SCHED_FIFO` : le processus avec la plus grande priorité (1-99) est exécuté jusqu'à ce qu'il se termine, se bloque, ou soit preempté par un processus de plus haute priorité.
- `SCHED_RR` : version round-robin du précédent. Chaque priorité (1-99) est organisé en file FIFO.

- Les processus "temps réels" sont prioritaires sur tous les autres.
- Un mécanisme permet de limiter la monopolisation du cpu par le temps réel : `/proc/sys/kernel/sched_rt_runtime_us` et `/proc/sys/kernel/sched_rt_period_us`

Stratégies normales

Principalement `SCHED_OTHER` "Historiquement" (noyau 2.6) :

- chaque processus a un timeslice de temps d'exécution
- deux listes de processus actifs (timeslice > 0) / expirés (timeslice $= 0$) par priorité (1-140).
- lorsque qu'un processus expire son timeslice, celui-ci est recalculé, et la processus intègre la file des expirés.
- quand le file des actifs est vide, on switche les deux files.

- L'ordonnancement est basé sur une priorité dynamique : calcul qui fait intervenir la priorité statique (nice entre -20 et 10, 0 par défaut) et une pénalité/bonus si le processus est considéré comme interactif (dort souvent) ou non.
- Calcul du timeslice : initialement, la moitié de son père, sinon fonction de la priorité dynamique.

Après le noyau 2.6 : Completely Fair Scheduler (CFS). Idées :

- Quand un processus est exécuté, il augmente son `vruntime` : temps d'exécution pondéré par sa priorité.
- L'ordonnanceur utilise un arbre binaire de recherche coloré (red-black tree) pour trier les processus suivant leur `vruntime`

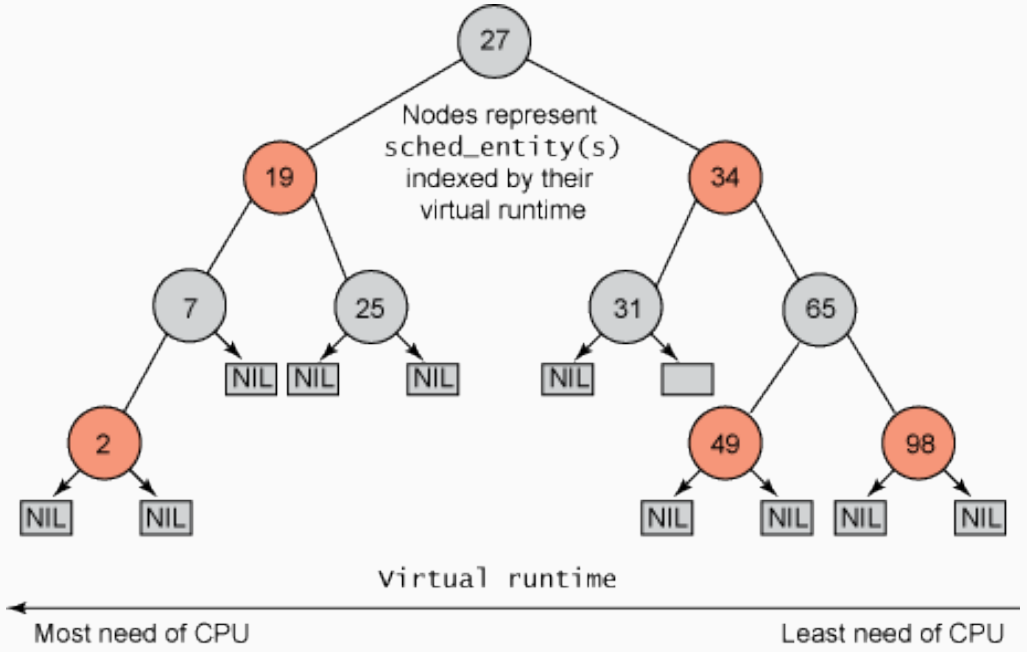


Figure 3: CFS (scheduler)

- On choisit toujours le noeud le plus à gauche (le plus petit `vruntime`)

Le temps d'exécution est pondéré par la priorité du processus (nice)

Une priorité faible accèlere le temps, une priorité forte le ralentit.

- I/O bound processus n'ont pas besoin de beaucoup de temps processeur. Leur `vruntime` sera faible $\Rightarrow$ on leur attribue une priorité importante.
- CPU bound processus au contraire ont besoin de beaucoup de temps processeurs $\Rightarrow$ on leur attribue une priorité faible.

Appels systèmes

```
int nice(int inc);
int getpriority(int which, id_t who);
int setpriority(int which, id_t who, int value);
int sched_get_priority_max(int policy);
int sched_get_priority_min(int policy);
int sched_getscheduler(pid_t pid);
int sched_setscheduler(pid_t pid, int policy,
    const struct sched_param *param);
int sched_getparam(pid_t pid, struct sched_param *param);
int sched_setparam(pid_t pid, const struct sched_param *param);
int sched_yield(void);
int sched_rr_get_interval(pid_t pid, struct timespec *interval);
int sched_getcpu(void);
int sched_setaffinity(pid_t pid, size_t cpusetsize,
    const cpu_set_t *mask);
int sched_getaffinity(pid_t pid, size_t cpusetsize,
    cpu_set_t *mask);
```

Commandes

```
chrt - manipulate the real-time attributes of a process

nice - run a program with modified scheduling priority

taskset - set or retrieve a process's CPU affinity
```