

Programmation système

Ressource R3.05 - Processus

monnerat@u-pec.fr 

IUT de Fontainebleau

1. Définition
2. Cycle de vie
 - Création
 - Vie du processus
 - Terminaison
3. Recouvrement
4. Contrôle

Définition

Processus

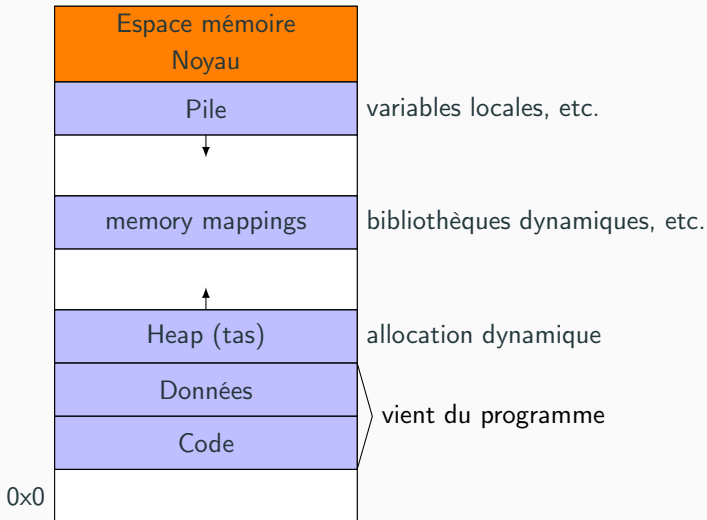
Image **dynamique** de l'exécution d'un programme.

- 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 $\Rightarrow$ **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

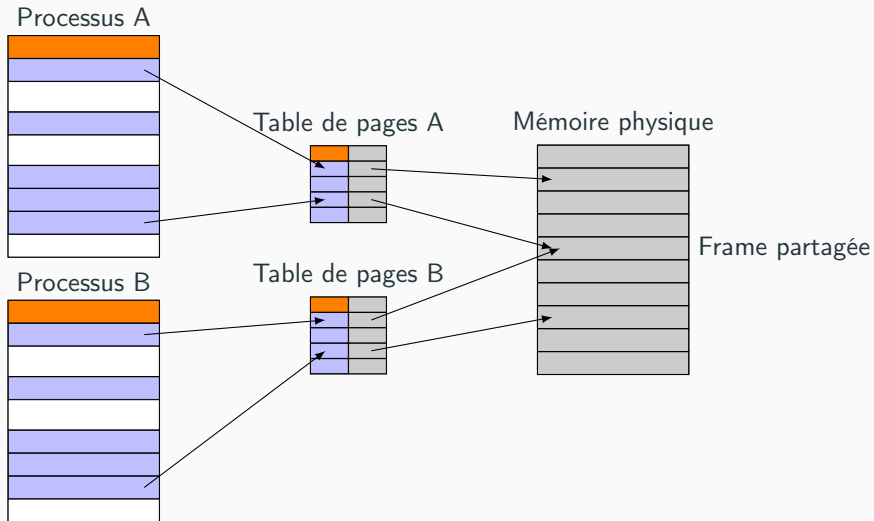


cat /proc/pid/maps

```
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]
fffffffff600000-fffffffff601000 --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 père 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);
```

```
#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()

```
#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 père 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.

```
#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 ?

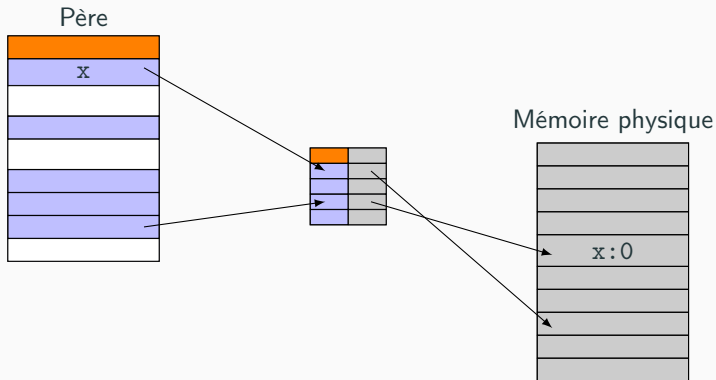
```
int main(int argc, char *argv[])
{
    pid_t p;
    int x=0;

    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);
}
```

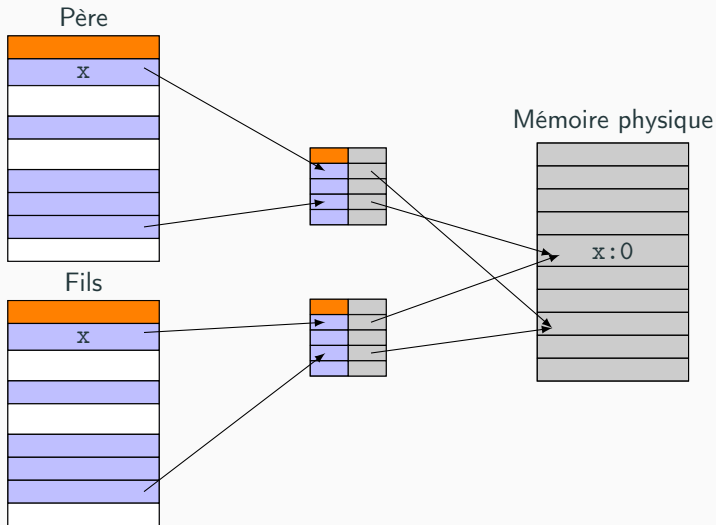
Copy on write

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



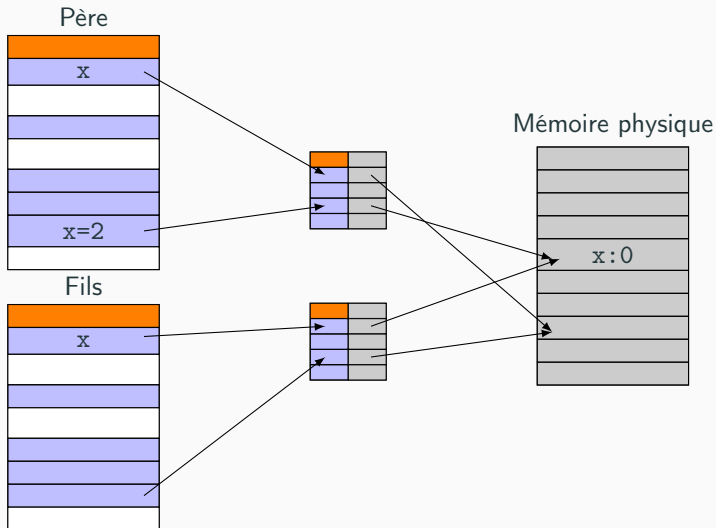
Copy on write

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



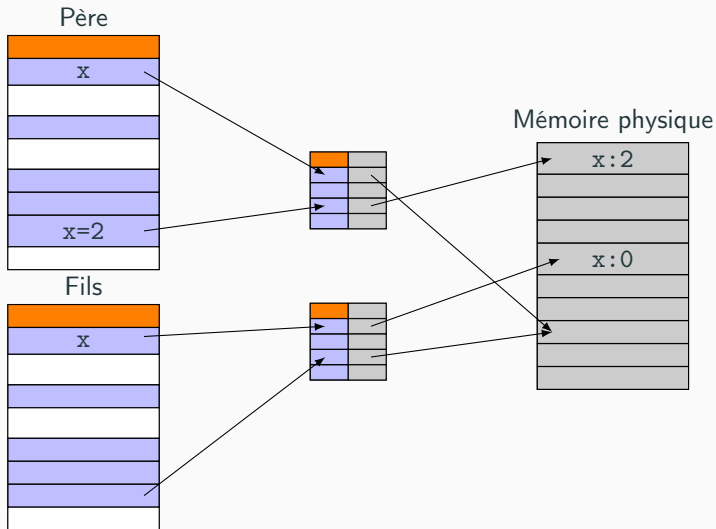
Copy on write

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



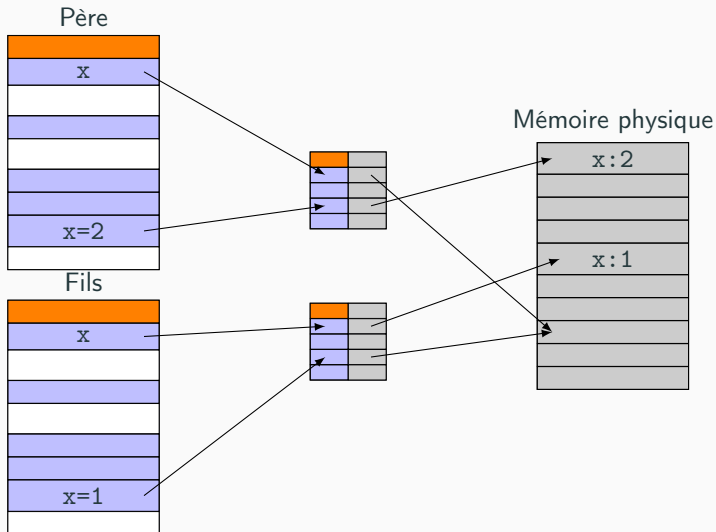
Copy on write

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

[...]

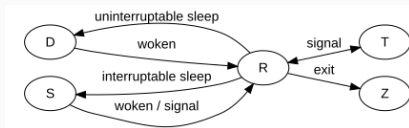
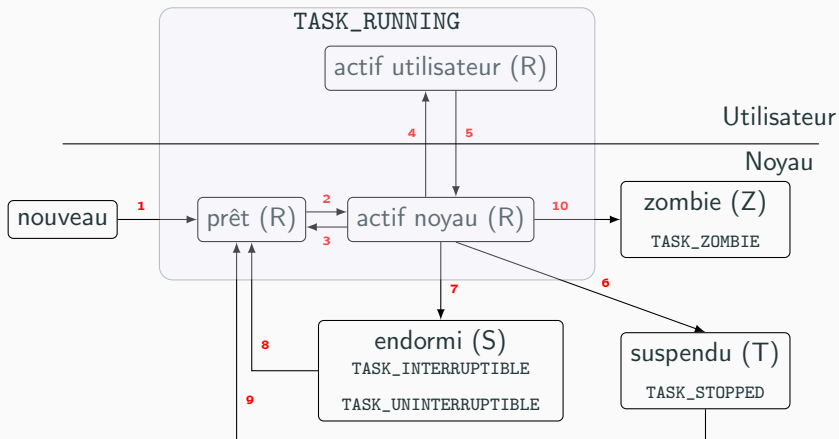
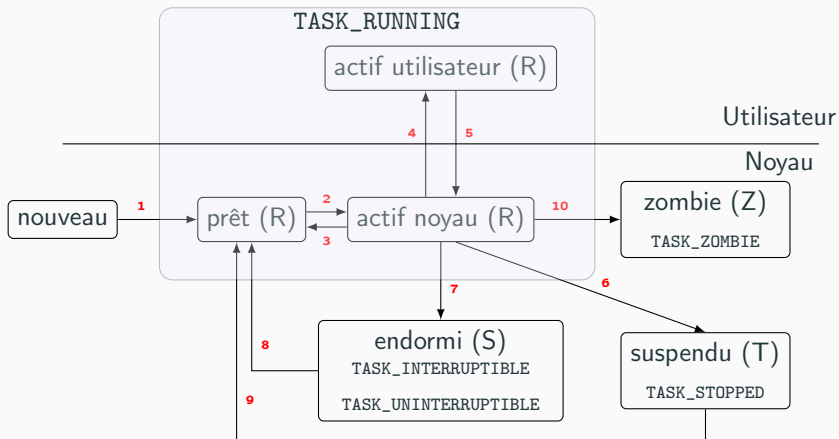


Figure 1: États d'un processus

Plus précisément :

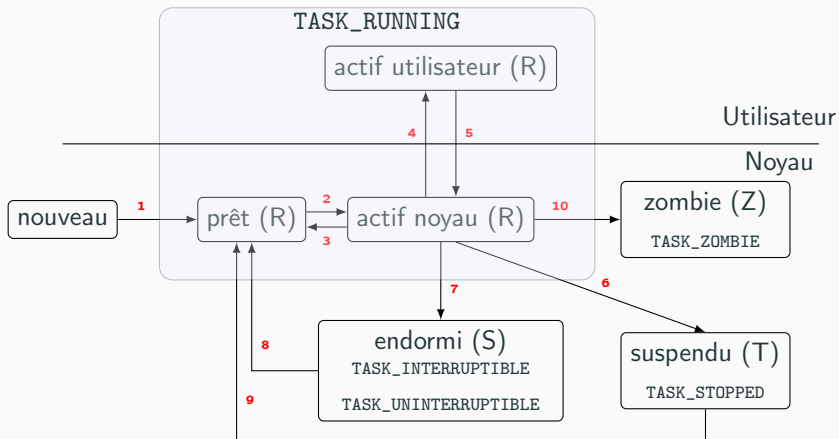


Plus précisément :



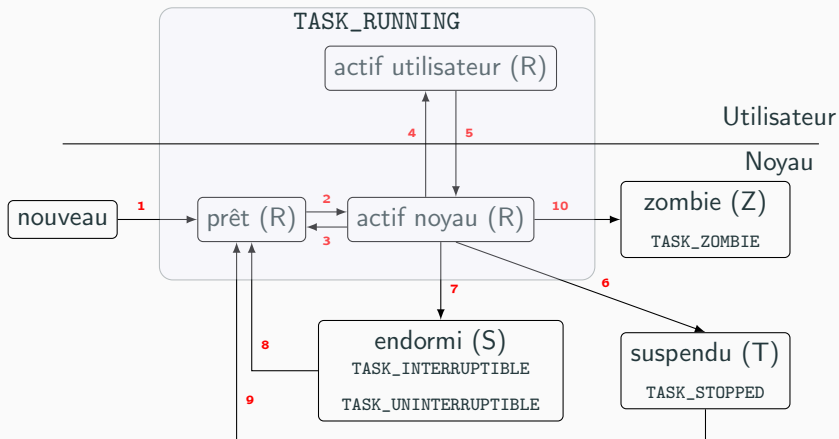
1 Création du processus.

Plus précisément :



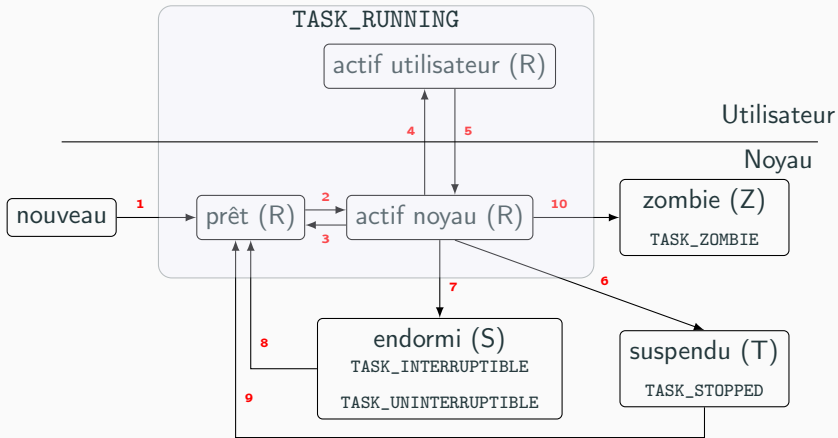
2 Election : le processus est élu par l'ordonnanceur.

Plus précisément :



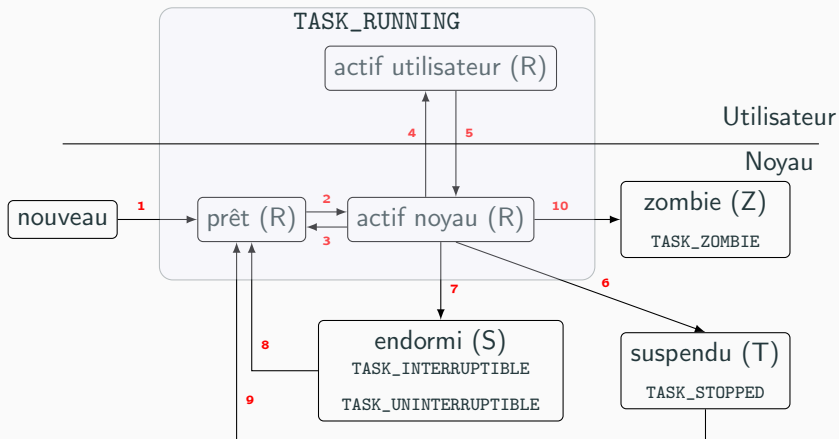
3 Prémption : le processus élu est remplacé dans le file des prêts.

Plus précisément :



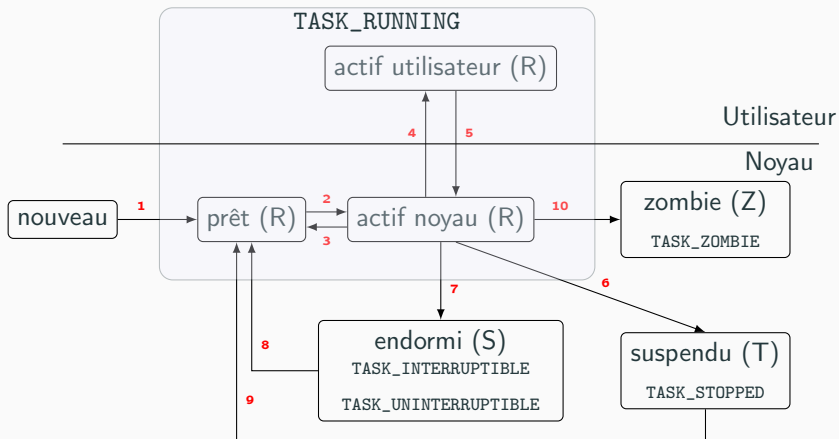
4 Le processus revient d'un appel système, ou d'une interruption.

Plus précisément :



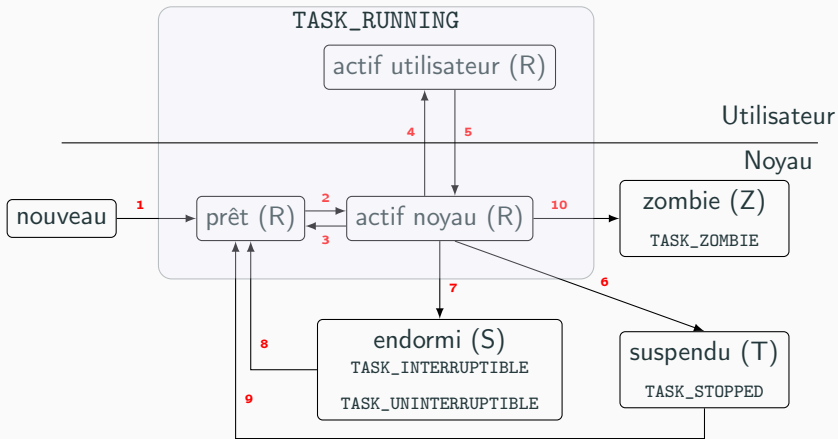
5 Le processus réalise un appel système.

Plus précisément :



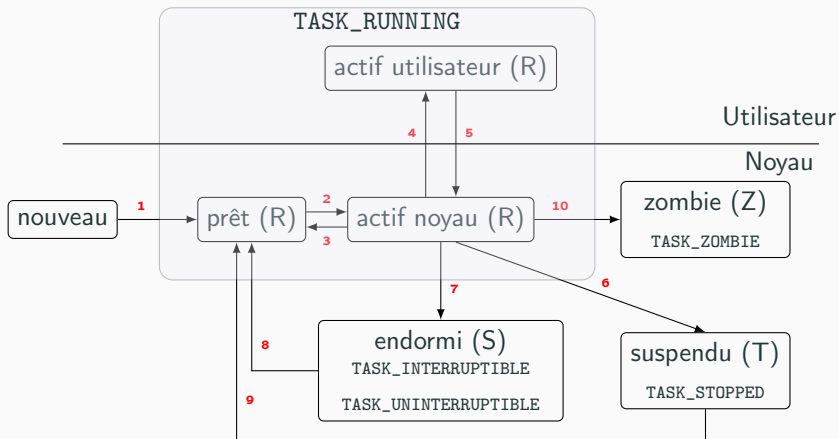
6 Délivrance d'un signal de suspension (SIGSTOP, SIGTSTP).

Plus précisément :



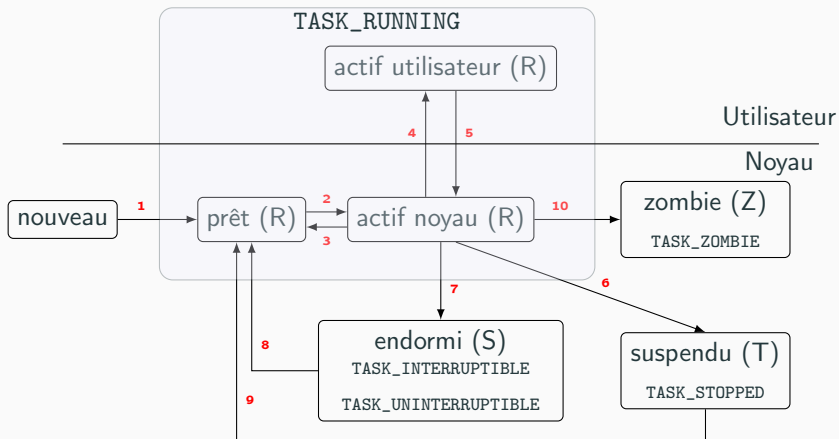
7 Attente d'un événement (`read()`, `wait()`, etc.).

Plus précisément :



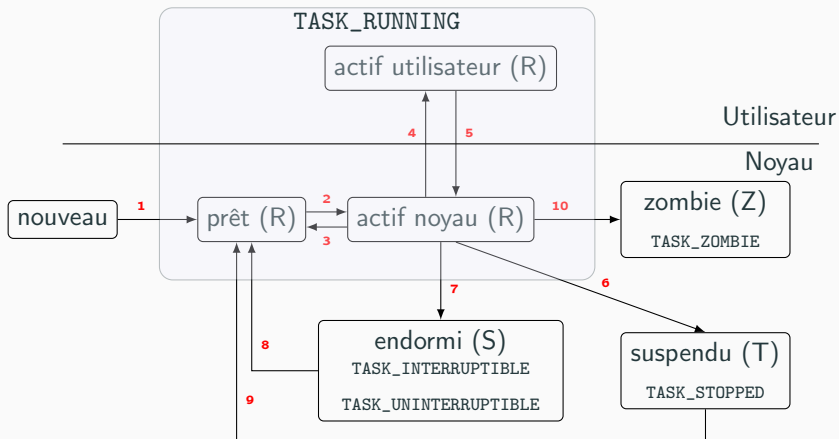
8 L'événement attendu a eu lieu.

Plus précisément :



9 Réveil du processus par le signal SIGCONT.

Plus précisément :



10 Terminaison.

Mode user vs Kernel

Le mode user a beaucoup moins de privilèges que le mode noyau.
(presque, cf vdso) Tous les appels systèmes sont effectués en mode noyau.

En C, avec une enveloppe

```
#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, sauvegarde des registres, action, restauration registre, retour en mode user ...

Cycle de vie

Terminaison

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

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

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

```
#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()`

```
#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()

```
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()

```
#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 positionné sur le descripteur de fichier.
(O_CLOEXEC à l'ouverture avec open, où FD_CLOEXEC avec fnctl)
- UID, etc.

```
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);
    }
}
```

Les primitives `exec1p()` et `execvp` utilise le PATH. En modifiant la variable d'environnement, on peut injecter son propre code ...

Contrôle

Les processus sont partitionnés en groupe, eux-même partitionné en session. But ? job control, E/S et envoi groupé de signaux

- un processus appartient à un groupe unique.
- un groupe appartient à une session unique.
- une session peut être contrôlée par un terminal tty device.
- process 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 -echoprtr echoctl echo
```

Chaque session est partitionnée 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 reçoivent 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

```
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); /* stderr */

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

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