TP de Systèmes d'exploitation
François Cayre / francois.cayre@grenoble-inp.fr

Stratégies pour l'ordonnancement

Notions abordées : ordonnancement, signaux, communication inter-processus (IPC SYS V).

Fonctions utilisées : `ftok(3)`, `msgget(2)`, `msgsnd(2)`, `msgrcv(2)`.

1 Brève histoire de l'ordonnancement

L'ordonnancement est un problème dépassant très largement le cadre des Systèmes d'exploitation, et date de bien avant l'invention des ordinateurs. En effet, il a fallu trouver des manières d'optimiser diverses métriques sur une chaîne d'assemblage ou de réparation (d'avions[1], de voitures, *etc.*) : le nombre de réparations terminées par unité de temps, le nombre de clients servis par unité de temps, *etc.* La branche des mathématiques appliquées concernée par ce problème s'appelle la recherche opérationnelle (ou l'optimisation combinatoire). De nombreuses stratégies d'ordonnancement sont héritées de ce cadre opérationnel (FCFS, RR, SJF).

Cette histoire est à compléter par celle des ordinateurs. Les premiers ordinateurs étaient des systèmes dits de *traitement par lot* (*batch*) : on programmait un ensemble de problèmes divers (facturation, simulations, etc.) à faire résoudre par la machine, et on en attendait le résultat. Deux observations doivent être faites : (1) certaines tâches peuvent être réputées plus urgentes que d'autres, et (2) ces systèmes n'étaient pas interactifs (aucun utilisateur n'était devant un quelconque terminal pour interagir avec la machine). Ajoutons pour terminer que les programmes étaient, au tout début des systèmes de traitement par lot, exécutés *en entier* avant de passer au suivant. Ce sont ces considérations historiques successives qui doivent guider votre compréhension des stratégies d'ordonnancement évoquées dans ce TP.

Nous utiliserons deux métriques pour évaluer les politiques d'ordonnancement :

1. Le temps de réponse (*response time*), défini comme la durée entre l'arrivée d'un processus dans le système et sa première exécution ;
2. Le temps de production (*turnaround time*), défini comme la durée entre l'arrivée d'un processus dans le système et sa terminaison.

---

1 Autre exemple : pendant la Seconde Guerre Mondiale, lorsque Londres était sous le feu nazi, il a fallu trouver comment maximiser le temps des avions de combat passé en vol, avec un nombre de pilotes donné. Cela a débouché sur une solution optimale (en $O(N^3)$) pour le problème dit de *flot maximal dans un graphe bipartite*. Ajoutez les efforts de l'équipe d'Alan Turing à Bletchley Park pour déchiffrer les messages codés par Enigma (un travail débuté avec succès par le mathématicien polonais Marian Rejewski, puis poursuivi plus modestement par les Français, qui ont aussi assuré la transmission des découvertes de Rejewski à Turing), et vous aurez une idée du nombre de vies sauvées par les mathématiques et l'informatique, alors naissante.

2 Un simulateur pour l'ordonnancement

Nous nous focalisons dans ce TP sur l'ordonnancement à court terme, consistant à choisir le prochain processus auquel donner le CPU.

Le problème est double car (1) nous voulons pouvoir *simuler* différentes politiques d'ordonnancement, et (2) nous devons le faire sur une machine elle-même déjà régie par l'ordonnanceur du système d'exploitation hôte (celui de votre machine de TP).

Notre simulateur devra être suffisamment flexible pour simuler des politiques simples, comme le tourniquet, ou plus avancées, comme par exemple un SJF préemptif. La plupart des ordonnanceurs préemptifs utilisent une interruption pour prévoir de déclencher le prochain changement de contexte. Nous devrons donc prévoir un mécanisme permettant de déclencher un nouvel ordonnancement dans le futur.

Un ordonnanceur est régi par des événements (cf. *infra*) qui déterminent, parmi les processus prêts à être exécutés à un instant donné (dans la *Ready queue*), celui auquel donner le[2] CPU (celui de la machine hôte sur laquelle s'exécutera votre TP). Une simulation plus fine nécessiterait l'utilisation d'une *Waiting queue*, contenant les processus en attente de pouvoir être exécutés à nouveau. Le problème est donc celui d'une allocation concurrente (les processus prêts étant les concurrents) de ressource (le temps à passer sur le CPU). Bien évidemment, nous allons utiliser les signaux pour émuler les événements de l'ordonnanceur.

Notons tout de suite que le but de ce TP est double : s'il s'agit bien évidemment d'illustrer la notion d'ordonnancement, on en profitera aussi pour mettre en place une IPC, et surtout pour expliquer comment se servir (presque !) proprement des signaux.

3 Compléments sur les signaux

Notre gestion des signaux dans le TP sur le démon, si elle avait pour but de vous en faire découvrir l'interface de base, pouvait se justifier par la structure même de notre démon. Elle n'en reste pas moins touchante de naïveté et n'est pas à privilégier dans un cadre plus général : lorsque plusieurs processus peuvent s'envoyer des signaux les uns aux autres à n'importe quel instant (dans notre démon, les signaux ne concernaient qu'un seul processus).

En effet, on rappelle que les signaux sont des événements *asynchrones*. En particulier, si un signal est intercepté, son gestionnaire peut être interrompu par la réception d'un autre signal… Pire que cela : si le gestionnaire en question appelle, par exemple, `printf(3)`, c'est le code même de `printf(3)` qui peut être interrompu[3]. Se pose alors la question suivante : que peut-on exécuter dans un gestionnaire de signal, sous contraintes (1) de prendre en compte tous les signaux, et (2) d'y réagir effectivement ?

---

2 Remarquez que si vos machines ont plusieurs cœurs physiques d'exécution, nous n'en utiliserons pourtant qu'un seul dans ce TP. Un ordonnanceur réel doit évidemment pouvoir gérer plusieurs coeurs physiques d'exécution, mais cela obscurcirait assez inutilement notre propos.

3 Si `printf(3)` n'en fait pas partie, POSIX définit néanmoins une liste de fonctions (*async-safe functions*) pouvant être appelées de manière sûre dans un gestionnaire de signal (voir `signal(7)`). Il n'en reste pas moins vrai que l'on risque de ne pas pouvoir prendre en compte un signal qui arriverait pendant l'exécution d'un gestionnaire de signal…

La réponse apportée par la norme du langage C est radicale : seul le changement d'état d'une variable de type `sig_atomic_t` est garanti. Et encore, à condition en pratique que cette variable soit déclarée avec le qualifieur `volatile` (pour forcer un accès à la donnée réelle, pas à une de ses éventuelles copies, pas forcément à jour au moment de la réception du signal, dans la hiérarchie mémoire). Idéalement, un gestionnaire de signal devrait donc toujours s'écrire :

```
volatile sig_atomic_t state = 0 ;

void sig_handler_flip( int signum ) {

     state = 1 ;

}

void sig_handler_flap( int signum ) {

     state = 0 ;

}
```

Cela nous garantit de respecter la première contrainte ci-dessus : celle consistant à voir passer tous les signaux. Notons tout de suite une première entorse dans le code fourni, puisqu'il contient des gestionnaires écrits ainsi :

```
void sig_handler( int signum ) {

     signal( signum, sig_handler ) ;

     state = 1 ;

}
```

La fonction `signal(2)` est définie comme *async-safe* par POSIX, mais nous courrons toujours le risque d'être interrompu entre l'appel à `signal( signum, sig_handler )` (pour recharger le gestionnaire) et la mise à jour de la variable d'état `state`[4]. C'est là où nous devons nous arranger pour que le problème, s'il se produisait, ait des conséquences (très) limitées. Dans ce qui suit, nous ferons donc l'hypothèse que nous ne serons jamais interrompus dans un gestionnaire de signal. Plus exactement, tout se passera comme si c'était effectivement le cas.

En pratique, nous utiliserons une échelle de temps bien plus lente que celle de la machine hôte. Cela vous permettra en outre de régler la vitesse de la simulation pour pouvoir la suivre à l'œil nu. Notre simulateur fonctionnera avec des fausses millisecondes (une fausse milliseconde valant `DURATION_SCALE` millisecondes réelles, typiquement 80), alors que la machine hôte peut traîter les signaux en quelques dizaines de microsecondes réelles.

4 C'est, entre autres, ce problème qui a suscité l'évolution de l'interface édictée par POSIX, qui propose de gérer les signaux par l'intermédiaire de la fonction `sigaction(2)`. Cette fonction permet de spécifier le rechargement automatique du gestionnaire *au moment* de son association au signal concerné. Ainsi, le risque d'interruption mentionné dans le texte disparaît-il avec l'appel, devenu inutile, à `signal( signum, sig_handler )`. Aussi curieux que cela puisse paraître, la norme du langage C ne définit pas (encore ?) `sigaction(2)`. Ce qui se passe dans les comités de normalisation relève parfois du mystère... Bien que d'un maniement parfois ardu à l'occasion (la raison pour laquelle ce cours d'introduction ne l'utilise pas), préférez-la à l'avenir lorsqu'elle est disponible sur votre architecture cible !

4 Un protocole de communication

Les développements ci-dessus nous garantissent de *pouvoir* prendre en compte tous les signaux (la variable d'état sera toujours correctement mise à jour), mais ne nous disent cependant rien de la manière dont on va pouvoir y *réagir effectivement*. La réponse à ce problème dépend de ce que l'on veut faire, et nous devons donc préciser maintenant les modalités de communication entre l'ordonnanceur et les processus qu'il contrôle.

Toute communication un peu digne d'intérêt entre deux interlocuteurs nécessite (1) un *canal de communication* pour faire transiter l'information utile, et (2) un *protocole commun de signalisation* des événements associés à la vie de ce canal (pour signaler à notre interlocuteur, par exemple, qu'un nouveau message est disponible dans le canal de communication).

Les processus vont utiliser une file de messages[5] comme canal de communication pour envoyer des messages à l'ordonnanceur (la signalisation autour du canal étant donc naturellement implantée par des… signaux !)[6]. L'ordonnanceur n'envoit jamais de message aux processus, seulement des signaux. On rappelle qu'une file de messages impose que les messages soient *typés*, en plus de contenir éventuellement des *données utiles*.

Nous utiliserons la syntaxe `{type|donnée}` pour décrire les messages envoyés par les processus à l'ordonnanceur. Les seuls messages possibles sont :

- `{PROC_CREATE|pid,TTL,next_burst}` : Le processus de PID `pid` demande sa création dans l'ordonnanceur pour une durée de vie de `TTL` millisecondes, avec un premier burst de `next_burst` ms ;
- `{PROC_DELETE|pid}` : Le processus de PID `pid` demande sa suppression de l'ordonnanceur ;
- `{PROC_BURST|pid,next_burst}` : Le processus de PID `pid` informe l'ordonnanceur qu'il prévoit un prochain *burst* CPU de `next_burst` ms[7] ;
- `{PROC_YIELD|pid}` : Le processus de PID `pid` informe l'ordonnanceur qu'il ne peut plus utiliser le CPU (pour simuler, par exemple, qu'il a demandé à l'OS de procéder pour lui à une opération d'entrée/sortie et qu'il en attend le résultat avant de pouvoir continuer à s'exécuter).

Précisons maintenant la sémantique associée aux signaux :

- Ordonnanceur
  - `SIGINT`, `SIGTERM` : Mettre fin à la simulation ;
  - `SIGUSR2` : Au moins un nouveau message est disponible dans la file de messages ;

5 Dans ce TP, nous utilisons les files de messages SYS V. POSIX définit aussi, depuis une date plus récente, des files de messages aux caractéristiques extrêmement proches. Comme vous l'avez compris, il existe souvent plusieurs interfaces pour faire la même chose, et un ingénieur doit, au minimum, être capable d'utiliser l'une ou l'autre indifféremment suivant son choix, leurs caractéristiques, ou leur disponibilité sur l'architecture cible.

6 On aurait évidemment pu faire d'autres choix de conception, mais nous avons quelques objectifs pédagogiques à atteindre !

7 Dans un vrai ordonnanceur, les processus ne déclarent évidemment pas de durée de prochain *burst* CPU. Mais nous construisons un *simulateur*…

    - SIGALRM : Le moment *prévu par l'ordonnanceur lors du dernier changement de contexte* pour procéder au prochain est arrivé. C'est ainsi que nous simulons une interruption pour la préemption. En plus de la réception de SIGALRM, un nouvel ordonnancement peut être causé par l'arrivée d'un message de type PROC_YIELD ou PROC_DELETE.
- Processus
    - SIGTERM : Détruire le processus ;
    - SIGINT : Le processus a terminé son travail, ou l'utilisateur veut détruire le processus ;
    - SIGUSR1 : L'ordonnanceur demande de se préparer à laisser le CPU[8] ;
    - SIGALRM : Le processus simule un appel système (pendant lequel il est réputé ne pas pouvoir utiliser le CPU). Le processus peut choisir de s'envoyer SIGALRM à un moment différent de ce qu'il avait annoncé à l'ordonnanceur en lui envoyant son dernier next_burst en date ;
    - SIGCONT : L'ordonnanceur redonne le CPU au processus ;
    - SIGSTOP.

Comme nous verrons, l'ordonnanceur enverra SIGSTOP à un processus *après* lui avoir envoyé SIGUSR1 (pour l'arrêter *effectivement*). Et l'ordonnanceur fera reprendre le processus là où il en était en lui envoyant SIGCONT, qui est le seul signal auquel réagira un processus ayant reçu SIGSTOP. Pour un processus, l'interception de SIGCONT lui sert à déterminer à partir de quand continuer à décrémenter sa durée de vie restante.

On peut maintenant décrire le protocole régissant les étapes de la vie d'un processus dans l'ordonnanceur, depuis sa création jusqu'à sa terminaison, provoquée par l'utilisateur ou non. Il s'agit d'un simple protocole à base de poignées de main (*handshakes*), comme peut par exemple l'être TCP en Réseaux. Ce protocole est spécifié en Annexe pour les besoins de l'exhaustivité.

Le code fourni implante ce protocole, pour l'ordonnanceur et chacun des processus, sous la forme d'une machine à états qui scrute continuellement la valeur de sa variable d'état pour détecter qu'un signal a été reçu. Lorsqu'un signal est reçu, une fonction est exécutée pour réaliser le protocole.

Pendant l'exécution de ces fonctions, par construction, on ne peut pas prendre en compte l'arrivée d'un signal.

Cela est particulièrement important pour l'ordonnanceur, susceptible de recevoir SIGUSR2 en rafale. Pour réagir au mieux à SIGSUR2, la fonction process_messages() va lire tous les messages disponibles dans la file de messages. Ainsi, si plusieurs processus ont voulu envoyer simultanément un message à l'ordonnanceur, ils vont tous parvenir à écrire dans la file de messages[9], et seront tous pris en compte, même s'il est possible que l'ordonnanceur ne voie pas passer tous les signaux SIGUSR2 correspondant.

---

8 Là encore, le but de *simulation* que nous nous fixons nous impose cette manière de faire. Dans un vrai système, un processus n'est évidemment pas prévenu qu'il va être arrêté. Ici, cela nous permet de faire remonter une valeur pour next_burst en interceptant SIGUSR1, puisque l'on rappelle que SIGSTOP ne peut pas être intercepté…

9 Les files de messages SYSV permettent d'enfiler et de défiler des messages de manière atomique.

5 Travail à réaliser

5.1 Mise en place de la file de messages

Complétez le code fourni. C'est indispensable pour continuer le TP.

5.2 Traitement par lot : tourniquet (RR) vs. SJF

Cette comparaison doit se faire dans le contexte des premiers systèmes de traitement par lot. Notre politique SJF consiste en réalité à sélectionner comme prochain processus à ordonnancer celui avec le plus court *burst* CPU[10]. Du point de vue de l'attente de clients aux caisses d'un magasin, SJF représente donc l'équivalent de l'introduction d'une file spéciale pour les clients voulant acheter peu d'articles. Et dans une chaîne de maintenance, cela consiste à s'occuper d'abord des opérations nécessitant le moins de ressources (pièces, temps, main-d'œuvre).

Vous lancerez l'une ou l'autre simulation avec :

```
$ make launch_session_rr
```

et :

```
$ make launch_session_sjf
```

Chaque simulation comporte deux processus (réputés rapides) avec de courts *bursts* CPU et un processus (réputé lent) avec de longs *bursts* CPU.

Déterminez la métrique pertinente pour mesurer l'éventuelle amélioration apportée par SJF par rapport à RR. Vérifiez votre intuition à l'aide du simulateur, au besoin en exécutant plusieurs réalisations des simulations (les durées de *bursts* CPU sont tirées uniformément aléatoirement). Vous êtes évidemment encouragés à faire varier les paramètres de la simulation (dans le `Makefile` donc).

5.3 Systèmes interactifs : introduction de la politique MLFQ

En cours, nous avons fait la différence entre les processus limités par le CPU (compilateurs, simulations numériques) et les processus limités par les entrées/sorties (serveur web, suite bureautique), soulignant les nouvelles contraintes induites par l'apparition des systèmes interactifs. Un système interactif doit minimiser l'angoisse de ses utilisateurs, qui sont plongés dans les affres de la déréliction lorsque plus de trois secondes s'écoulent avant de voir les prochains progrès dans l'avancement de leur(s) processus.

Dans notre simulateur, cette distinction est implantée en faisant varier la probabilité pour un processus de laisser le CPU avant la fin de son quantum de temps, et donc de procéder, par exemple, à un appel système pour une entrée/sortie au cours de son exécution.

Les systèmes d'exploitation généralistes modernes sont interactifs, et doivent donc prendre en compte cette distinction (entre processus limités par le CPU ou par les entrées/sorties) dans leur ordonnancement. L'idée est donc d'*apprendre le profil d'exécution des processus au cours de leur exécution*, afin de donner

10 Une version de ce que l'on appelle parfois une politique STCF : *Shortest-Time-to-Completion-First*.

préférentiellement le CPU aux processus limités par les entrées/sorties, laissant, *grosso modo*, ceux limités par le CPU s'exécuter lorsqu'il n'y a rien de plus urgent à faire.

Dans notre simulateur, cette nouvelle stratégie porte le nom de *Multi-Level-Feedback-Queue* (MLFQ), car elle reprend quelques-unes des grandes lignes de cette politique, implantée sous une manière ou une autre dans les systèmes d'exploitation généralistes modernes (Linux, *Fenêtres*, *etc.*) Dans ce scénario de simulation, un processus annonce vouloir le CPU pour `next_burst` ms, *mais peut choisir de le rendre avant* (en émettant un message `PROC_YIELD`).

Dans notre implantation de MLFQ, nous allons apprendre le profil d'utilisation du CPU en calculant la *moyenne de la proportion de temps effectivement passé sur le CPU d'une exécution sur l'autre*. La *Ready queue* contient alors les processus prêts à être exécutés, *classés par utilisation croissante du CPU*.

Ainsi, au fur et à mesure de l'apprentissage du profil d'exécution des processus, on privilégiera les processus effectuant davantage d'entrées/sorties que les autres. On retrouve ici la notion de retour (*feedback*) sur l'exécution effective des processus dans MLFQ.

La notion de multi-niveau (*multi-level*) n'est pas implantée explicitement dans notre simulateur : elle consisterait à mettre dans un tourniquet les processus présentant la même utilisation du CPU.

Enfin, lorsqu'un processus entre dans le système, il est exécuté une première fois en priorité absolue. Expliquez pourquoi.

Observez et discutez la pertinence de cette politique dans une simulation comportant deux processus interactifs (limités par les entrées/sorties) et un processus limité par le CPU :

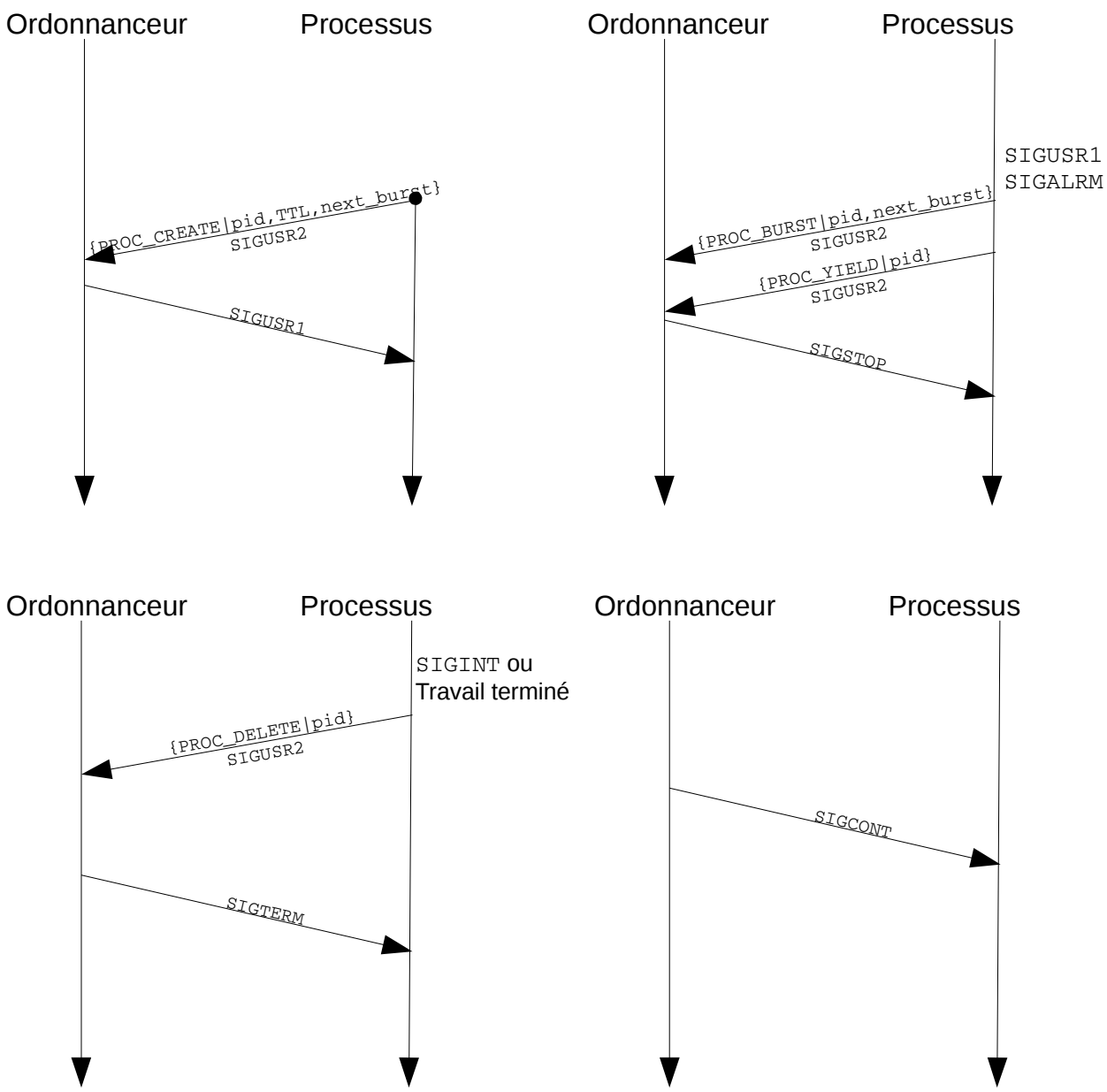
```
$ make launch_session_mlfq
```

Quel est le problème pouvant surgir en cas de multiples lancements de processus interactifs ? Proposez une modification mineure de notre implantation de MLFQ pour y remédier[11].

Indice : Nous n'avons considéré que des métriques de *performance*, nous pourrions maintenant vouloir prendre en compte l'*équité* de l'ordonnancement…

11 Mais ne l'implantez pas forcément ! L'examen du code fourni devrait vous faire sentir assez rapidement la quantité d'embûches pouvant être rencontrées en pratique… Encore que ce ne serait sans doute pas trop difficile !

6 Annexe



Vie d’un processus : création et terminaison (gauche), arrêt et reprise (droite).