Chap. X : Ordonnancement

Laurent Poinsot

UMR 7030 - Université Paris 13 - Institut Galilée

Cours “Architecture et Système”

Le compilateur sépare les différents objets apparaissant dans un programme dans des zones mémoires (code, données, librairies, pile).

Précisons cela par un exemple concret.

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
/* Variables globales */
const int nbMois=12;
int gi=5;
int Tab[]={1,2,3}
char Titre[50];
```

Le compilateur sépare les différents objets apparaissant dans un programme dans des zones mémoires (code, données, librairies, pile). Précisons cela par un exemple concret.

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
/* Variables globales */
const int nbMois=12;
int gi=5;
int Tab[]={1,2,3}
char Titre[50];
```

Le compilateur sépare les différents objets apparaissant dans un programme dans des zones mémoires (code, données, librairies, pile). Précisons cela par un exemple concret.

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
/* Variables globales */
const int nbMois=12;
int gi=5;
int Tab[]={1,2,3}
char Titre[50];
```

Puis le main :

```
main (){
/*Variables locales du main ()
donc dans la pile*/
int t=2;int *pti=&i;
char *mes="il fait beau";
char *ad;/*Pour allocation dynamique*/
int T[2];
char c;
ad=(char*)malloc(512);
strcpy(Titre,"**********");
```

Puis le main :

```
main (){
/*Variables locales du main ()
donc dans la pile*/
int t=2;int *pti=&i;
char *mes="il fait beau";
char *ad;/*Pour allocation dynamique*/
int T[2];
char c;
ad=(char*)malloc(512);
strcpy(Titre,"**********");
```

Suite du main() :

```
printf("&nbMois : %p\n",&nbMois);
printf("&gi : %p, Tab : %p, Titre :
%p\n",&gi,Tab,Titre);
printf("&i : %p, pti : %p, &pti :
%p\n",&i,pti,&pti);
printf("mes : %p, &mes : %p\n, mes,&mes);
printf("ad : %p, &ad : %p\n",ad,&ad);
printf("T : %p\n",T);
printf("strcpy : %p\n,strcpy);
printf("strcmp : %p\n",strcmp);}
```

Résultat de l'exécution du programme :

```
&nbMois : 0x8048600
&gi : 0x8049810, Tab : 0x8049614, Titre :
0x8049c80
&i : 0xbffffc84, pti : 0xbffffc84, &pti :
0xbffffc80
mes : 0x80489e0, &mes : 0xbffffc7c
ad : 0x8049e50, &ad : 0xbffffc78
T : 0xbffffc70
strcpy : 0x8048788
strcmp : 0x8048708
```

Résultat de l'exécution du programme :

```
&nbMois : 0x8048600
&gi : 0x8049810, Tab : 0x8049614, Titre :
0x8049c80
&i : 0xbffffc84, pti : 0xbffffc84, &pti :
0xbffffc80
mes : 0x80489e0, &mes : 0xbffffc7c
ad : 0x8049e50, &ad : 0xbffffc78
T : 0xbffffc70
strcpy : 0x8048788
strcmp : 0x8048708
```

On schématise maintenant l'implémentation par région mémoire pour cet exemple. Il y a tout d'abord la région du **code** qui contient les constantes 12 (`nbMois`) et `"il fait beau"`. Ces données sont **protégées en écriture** : si on essaie par exemple d'exécuter l'instruction `mes[0]='*';`, alors il y a une erreur de segmentation. Puis les données globales initialisées (`gi, Tab`) se trouvent avant la donnée globale non intialisée `Titre`. Les variables du `main ()` sont implémentées dans la **pile** ; leurs adresses sont décroissantes (`i, pti, mes, ad, T, c`). `malloc()` a permis d'allouer une zone dont l'adresse est dans la variable `ad` (soit 08049e50). Cette zone allouée se trouve dans le **tas**. Les arguments du `main` (quand il y en a) et les variables d'environnement sont également mémorisés dans la pile (en bas de celle-ci). Enfin entre le tas et la pile, il y a la zone réservée aux librairies utilisées.

On schématise maintenant l'implémentation par région mémoire pour cet exemple. Il y a tout d'abord la région du **code** qui contient les constantes 12 (`nbMois`) et `"il fait beau"`. Ces données sont protégées en écriture : si on essaie par exemple d'exécuter l'instruction `mes[0]='*';`, alors il y a une erreur de segmentation. Puis les données globales initialisées (`gi, Tab`) se trouvent avant la donnée globale non intialisée `Titre`. Les variables du `main ()` sont implémentées dans la **pile** ; leurs adresses sont décroissantes (`i, pti, mes, ad, T, c`). `malloc()` a permis d'allouer une zone dont l'adresse est dans la variable `ad` (soit 08049e50). Cette zone allouée se trouve dans le **tas**. Les arguments du `main` (quand il y en a) et les variables d'environnement sont également mémorisés dans la pile (en bas de celle-ci). Enfin entre le tas et la pile, il y a la zone réservée aux librairies utilisées.

On schématise maintenant l'implémentation par région mémoire pour cet exemple. Il y a tout d'abord la région du **code** qui contient les constantes 12 (`nbMois`) et `"il fait beau"`. Ces données sont protégées en écriture : si on essaie par exemple d'exécuter l'instruction `mes[0]='*';`, alors il y a une erreur de segmentation. Puis les données globales initialisées (`gi, Tab`) se trouvent avant la donnée globale non intialisée `Titre`. Les variables du `main ()` sont implémentées dans la **pile** ; leurs adresses sont décroissantes (`i, pti, mes, ad, T, c`). `malloc()` a permis d'allouer une zone dont l'adresse est dans la variable `ad` (soit 08049e50). Cette zone allouée se trouve dans le **tas**. Les arguments du `main` (quand il y en a) et les variables d'environnement sont également mémorisés dans la pile (en bas de celle-ci). Enfin entre le tas et la pile, il y a la zone réservée aux librairies utilisées.

On schématise maintenant l'implémentation par région mémoire pour cet exemple. Il y a tout d'abord la région du **code** qui contient les constantes 12 (`nbMois`) et `"il fait beau"`. Ces données sont protégées en écriture : si on essaie par exemple d'exécuter l'instruction `mes[0]='*';`, alors il y a une erreur de segmentation. Puis les données globales initialisées (`gi, Tab`) se trouvent avant la donnée globale non intialisée `Titre`. Les variables du `main ()` sont implémentées dans la **pile** ; leurs adresses sont décroissantes (`i, pti, mes, ad, T, c`). `malloc()` a permis d'allouer une zone dont l'adresse est dans la variable `ad` (soit 08049e50). Cette zone allouée se trouve dans le **tas**. Les arguments du `main` (quand il y en a) et les variables d'environnement sont également mémorisés dans la pile (en bas de celle-ci). Enfin entre le tas et la pile, il y a la zone réservée aux librairies utilisées.

On schématise maintenant l'implémentation par région mémoire pour cet exemple. Il y a tout d'abord la région du **code** qui contient les constantes 12 (`nbMois`) et `"il fait beau"`. Ces données sont protégées en écriture : si on essaie par exemple d'exécuter l'instruction `mes[0]='*';`, alors il y a une erreur de segmentation. Puis les données globales initialisées (`gi, Tab`) se trouvent avant la donnée globale non intialisée `Titre`. Les variables du `main ()` sont implémentées dans la **pile** ; leurs adresses sont décroissantes (`i, pti, mes, ad, T, c`). `malloc()` a permis d'allouer une zone dont l'adresse est dans la variable `ad` (soit 08049e50). Cette zone allouée se trouve dans le **tas**. Les arguments du `main` (quand il y en a) et les variables d'environnement sont également mémorisés dans la pile (en bas de celle-ci). Enfin entre le tas et la pile, il y a la zone réservée aux librairies utilisées.

On schématise maintenant l'implémentation par région mémoire pour cet exemple. Il y a tout d'abord la région du **code** qui contient les constantes 12 (`nbMois`) et `"il fait beau"`. Ces données sont protégées en écriture : si on essaie par exemple d'exécuter l'instruction `mes[0]='*';`, alors il y a une erreur de segmentation. Puis les données globales initialisées (`gi, Tab`) se trouvent avant la donnée globale non intialisée `Titre`. Les variables du `main ()` sont implémentées dans la **pile** ; leurs adresses sont décroissantes (`i, pti, mes, ad, T, c`). `malloc()` a permis d'allouer une zone dont l'adresse est dans la variable `ad` (soit 08049e50). Cette zone allouée se trouve dans le **tas**. Les arguments du `main` (quand il y en a) et les variables d'environnement sont également mémorisés dans la pile (en bas de celle-ci). Enfin entre le tas et la pile, il y a la zone réservée aux librairies utilisées.

On schématise maintenant l'implémentation par région mémoire pour cet exemple. Il y a tout d'abord la région du **code** qui contient les constantes 12 (`nbMois`) et `"il fait beau"`. Ces données sont protégées en écriture : si on essaie par exemple d'exécuter l'instruction `mes[0]='*';`, alors il y a une erreur de segmentation. Puis les données globales initialisées (`gi, Tab`) se trouvent avant la donnée globale non intialisée `Titre`. Les variables du `main ()` sont implémentées dans la **pile** ; leurs adresses sont décroissantes (`i, pti, mes, ad, T, c`). `malloc()` a permis d'allouer une zone dont l'adresse est dans la variable `ad` (soit 08049e50). Cette zone allouée se trouve dans le **tas**. Les arguments du `main` (quand il y en a) et les variables d'environnement sont également mémorisés dans la pile (en bas de celle-ci). Enfin entre le tas et la pile, il y a la zone réservée aux librairies utilisées.

On schématise maintenant l'implémentation par région mémoire pour cet exemple. Il y a tout d'abord la région du **code** qui contient les constantes 12 (`nbMois`) et `"il fait beau"`. Ces données sont protégées en écriture : si on essaie par exemple d'exécuter l'instruction `mes[0]='*';`, alors il y a une erreur de segmentation. Puis les données globales initialisées (`gi, Tab`) se trouvent avant la donnée globale non intialisée `Titre`. Les variables du `main ()` sont implémentées dans la **pile** ; leurs adresses sont décroissantes (`i, pti, mes, ad, T, c`). `malloc()` a permis d'allouer une zone dont l'adresse est dans la variable `ad` (soit 08049e50). Cette zone allouée se trouve dans le **tas**. Les arguments du `main` (quand il y en a) et les variables d'environnement sont également mémorisés dans la pile (en bas de celle-ci). Enfin entre le tas et la pile, il y a la zone réservée aux librairies utilisées.

On schématise maintenant l'implémentation par région mémoire pour cet exemple. Il y a tout d'abord la région du **code** qui contient les constantes 12 (`nbMois`) et `"il fait beau"`. Ces données sont protégées en écriture : si on essaie par exemple d'exécuter l'instruction `mes[0]='*';`, alors il y a une erreur de segmentation. Puis les données globales initialisées (`gi, Tab`) se trouvent avant la donnée globale non intialisée `Titre`. Les variables du `main ()` sont implémentées dans la **pile** ; leurs adresses sont décroissantes (`i, pti, mes, ad, T, c`). `malloc()` a permis d'allouer une zone dont l'adresse est dans la variable `ad` (soit 08049e50). Cette zone allouée se trouve dans le **tas**. Les arguments du `main` (quand il y en a) et les variables d'environnement sont également mémorisés dans la pile (en bas de celle-ci). Enfin entre le tas et la pile, il y a la zone réservée aux librairies utilisées.

On schématise maintenant l'implémentation par région mémoire pour cet exemple. Il y a tout d'abord la région du **code** qui contient les constantes 12 (`nbMois`) et `"il fait beau"`. Ces données sont protégées en écriture : si on essaie par exemple d'exécuter l'instruction `mes[0]='*';`, alors il y a une erreur de segmentation. Puis les données globales initialisées (`gi, Tab`) se trouvent avant la donnée globale non intialisée `Titre`. Les variables du `main ()` sont implémentées dans la **pile** ; leurs adresses sont décroissantes (`i, pti, mes, ad, T, c`). `malloc()` a permis d'allouer une zone dont l'adresse est dans la variable `ad` (soit 08049e50). Cette zone allouée se trouve dans le **tas**. Les arguments du `main` (quand il y en a) et les variables d'environnement sont également mémorisés dans la pile (en bas de celle-ci). Enfin entre le tas et la pile, il y a la zone réservée aux librairies utilisées.

On schématise maintenant l'implémentation par région mémoire pour cet exemple. Il y a tout d'abord la région du **code** qui contient les constantes 12 (`nbMois`) et `"il fait beau"`. Ces données sont protégées en écriture : si on essaie par exemple d'exécuter l'instruction `mes[0]='*';`, alors il y a une erreur de segmentation. Puis les données globales initialisées (`gi, Tab`) se trouvent avant la donnée globale non intialisée `Titre`. Les variables du `main ()` sont implémentées dans la **pile** ; leurs adresses sont décroissantes (`i, pti, mes, ad, T, c`). `malloc()` a permis d'allouer une zone dont l'adresse est dans la variable `ad` (soit 08049e50). Cette zone allouée se trouve dans le **tas**. Les arguments du `main` (quand il y en a) et les variables d'environnement sont également mémorisés dans la pile (en bas de celle-ci). Enfin entre le tas et la pile, il y a la zone réservée aux librairies utilisées.

Unix est un O.S. de temps partagé permettant à plusieurs utilisateurs d'exécuter plusieurs processus. L'O.S. alloue l'unité centrale (U.C.) de l'ordinateur successivement (mais toujours à un seul à la fois) à plusieurs processus se trouvant dans la mémoire centrale, de façon à optimiser l'utilisation de l'U.C. L'O.S. change de programme actif à chaque fois qu'un programme est en attente (d'entrées-sorties notamment) ou que son délai d'exécution est écoulé.

Unix est un O.S. de temps partagé permettant à plusieurs utilisateurs d'exécuter plusieurs processus. L'O.S. alloue l'unité centrale (U.C.) de l'ordinateur successivement (mais toujours à un seul à la fois) à plusieurs processus se trouvant dans la mémoire centrale, de façon à optimiser l'utilisation de l'U.C. L'O.S. change de programme actif à chaque fois qu'un programme est en attente (d'entrées-sorties notamment) ou que son délai d'exécution est écoulé.

Unix est un O.S. de temps partagé permettant à plusieurs utilisateurs d'exécuter plusieurs processus. L'O.S. alloue l'unité centrale (U.C.) de l'ordinateur successivement (mais toujours à un seul à la fois) à plusieurs processus se trouvant dans la mémoire centrale, de façon à optimiser l'utilisation de l'U.C. L'O.S. change de programme actif à chaque fois qu'un programme est en attente (d'entrées-sorties notamment) ou que son délai d'exécution est écoulé.

Unix est un O.S. de temps partagé permettant à plusieurs utilisateurs d'exécuter plusieurs processus. L'O.S. alloue l'unité centrale (U.C.) de l'ordinateur successivement (mais toujours à un seul à la fois) à plusieurs processus se trouvant dans la mémoire centrale, de façon à optimiser l'utilisation de l'U.C. L'O.S. change de programme actif à chaque fois qu'un programme est en attente (d'entrées-sorties notamment) ou que son délai d'exécution est écoulé.

Unix est un O.S. de temps partagé permettant à plusieurs utilisateurs d'exécuter plusieurs processus. L'O.S. alloue l'unité centrale (U.C.) de l'ordinateur successivement (mais toujours à un seul à la fois) à plusieurs processus se trouvant dans la mémoire centrale, de façon à optimiser l'utilisation de l'U.C. L'O.S. change de programme actif à chaque fois qu'un programme est en attente (d'entrées-sorties notamment) ou que son délai d'exécution est écoulé.

Chaque processus est représenté par une structure (de type `struct task_struct`) contenant des informations relatives au processus.

Cette structure est également appelée le PCB : Processus Control Block. La structure, de l'ordre du millier d'octets pour chaque processus (sous Linux), contient directement de nombreuses informations et permet d'accéder à d'autres informations grâce à des pointeurs sur d'autres structures. Elle contient en particulier les informations suivantes.

Chaque processus est représenté par une structure (de type `struct task_struct`) contenant des informations relatives au processus. Cette structure est également appelée le PCB : Processus Control Block. La structure, de l'ordre du millier d'octets pour chaque processus (sous Linux), contient directement de nombreuses informations et permet d'accéder à d'autres informations grâce à des pointeurs sur d'autres structures. Elle contient en particulier les informations suivantes.

Chaque processus est représenté par une structure (de type `struct task_struct`) contenant des informations relatives au processus. Cette structure est également appelée le PCB : Processus Control Block. La structure, de l'ordre du millier d'octets pour chaque processus (sous Linux), contient directement de nombreuses informations et permet d'accéder à d'autres informations grâce à des pointeurs sur d'autres structures. Elle contient en particulier les informations suivantes.

Chaque processus est représenté par une structure (de type `struct task_struct`) contenant des informations relatives au processus. Cette structure est également appelée le PCB : Processus Control Block. La structure, de l'ordre du millier d'octets pour chaque processus (sous Linux), contient directement de nombreuses informations et permet d'accéder à d'autres informations grâce à des pointeurs sur d'autres structures. Elle contient en particulier les informations suivantes.

Chaque processus est représenté par une structure (de type `struct task_struct`) contenant des informations relatives au processus. Cette structure est également appelée le PCB : Processus Control Block. La structure, de l'ordre du millier d'octets pour chaque processus (sous Linux), contient directement de nombreuses informations et permet d'accéder à d'autres informations grâce à des pointeurs sur d'autres structures. Elle contient en particulier les informations suivantes.

1. Des informations sur le processus

- L'état du processus (exécutable, en attente, stoppé, *etc.*), sa priorité ;
- Le numéro du processus (pid), de son père (ppid), du propriétaire (uid), du groupe (gid), les temps cpu, ...
- Une structure contenant les valeurs des registres permettant de sauver et de restituer l'état du processeur. Elle contient entre autres, le compteur ordinal qui repère la prochaine instruction à exécuter lors du passage en U.C. du processus.

1. Des informations sur le processus

- L'état du processus (exécutable, en attente, stoppé, *etc.*), sa priorité ;
- Le numéro du processus (pid), de son père (ppid), du propriétaire (uid), du groupe (gid), les temps cpu, ...
- Une structure contenant les valeurs des registres permettant de sauver et de restituer l'état du processeur. Elle contient entre autres, le compteur ordinal qui repère la prochaine instruction à exécuter lors du passage en U.C. du processus.

1. Des informations sur le processus

- L'état du processus (exécutable, en attente, stoppé, *etc.*), sa priorité ;
- Le numéro du processus (pid), de son père (ppid), du propriétaire (uid), du groupe (gid), les temps cpu, ...
- Une structure contenant les valeurs des registres permettant de sauver et de restituer l'état du processeur. Elle contient entre autres, le compteur ordinal qui repère la prochaine instruction à exécuter lors du passage en U.C. du processus.

1. Des informations sur le processus

- L'état du processus (exécutable, en attente, stoppé, *etc.*), sa priorité ;
- Le numéro du processus (pid), de son père (ppid), du propriétaire (uid), du groupe (gid), les temps cpu, ...
- Une structure contenant les valeurs des registres permettant de sauver et de restituer l'état du processeur. Elle contient entre autres, le compteur ordinal qui repère la prochaine instruction à exécuter lors du passage en U.C. du processus.

1. Des informations sur le processus

- L'état du processus (exécutable, en attente, stoppé, *etc.*), sa priorité ;
- Le numéro du processus (pid), de son père (ppid), du propriétaire (uid), du groupe (gid), les temps cpu, ...
- Une structure contenant les valeurs des registres permettant de sauver et de restituer l'état du processeur. Elle contient entre autres, le compteur ordinal qui repère la prochaine instruction à exécuter lors du passage en U.C. du processus.

1. Des informations sur le processus

- L'état du processus (exécutable, en attente, stoppé, *etc.*), sa priorité ;
- Le numéro du processus (pid), de son père (ppid), du propriétaire (uid), du groupe (gid), les temps cpu, ...
- Une structure contenant les valeurs des registres permettant de sauver et de restituer l'état du processeur. Elle contient entre autres, le compteur ordinal qui repère la prochaine instruction à exécuter lors du passage en U.C. du processus.

1. Des informations sur le processus

- L'état du processus (exécutable, en attente, stoppé, *etc.*), sa priorité ;
- Le numéro du processus (pid), de son père (ppid), du propriétaire (uid), du groupe (gid), les temps cpu, ...
- Une structure contenant les valeurs des registres permettant de sauver et de restituer l'état du processeur. Elle contient entre autres, le compteur ordinal qui repère la prochaine instruction à exécuter lors du passage en U.C. du processus.

2. Des pointeurs vers d'autres structures

- Un pointeur sur une structure répertoriant les fichiers ouverts par le processus ;
- Un pointeur sur une structure répertoriant (dans une liste) les régions utilisées par le processus (code, données, pile, librairie).

2. Des pointeurs vers d'autres structures

- Un pointeur sur une structure répertoriant les fichiers ouverts par le processus ;
- Un pointeur sur une structure répertoriant (dans une liste) les régions utilisées par le processus (code, données, pile, librairie).

3. Des pointeurs chaînant entre elles les structures

- Une liste chaînant entre eux tous les processus (`next_hash`) ;
- Une liste chaînant entre eux les processus prêts (`next_run`) ;
- Des pointeurs permettant de mémoriser la filiation (père-fils) des processus.

3. Des pointeurs chaînant entre elles les structures

- Une liste chaînant entre eux tous les processus (`next_hash`) ;
- Une liste chaînant entre eux les processus prêts (`next_run`) ;
- Des pointeurs permettant de mémoriser la filiation (père-fils) des processus.

3. Des pointeurs chaînant entre elles les structures

- Une liste chaînant entre eux tous les processus (`next_hash`) ;
- Une liste chaînant entre eux les processus prêts (`next_run`) ;
- Des pointeurs permettant de mémoriser la filiation (père-fils) des processus.

Un processus peut être dans l'état *prêt* (`TASK_RUNNING=0`) : il a toutes les ressources pour s'exécuter. Il peut être dans l'état *interrompu* (`TASK_INTERRUPTIBLE=1`) : il lui manque une ou plusieurs ressources pour continuer. Il peut être dans l'état *stoppé* (`TASK_STOPPED`), s'il a été stoppé par un `ctrl-Z`. D'autres états existent non détaillés ici (ininterruptible, zombie, swappé).

Un processus peut être dans l'état *prêt* (`TASK_RUNNING=0`) : il a toutes les ressources pour s'exécuter. Il peut être dans l'état *interrompu* (`TASK_INTERRUPTIBLE=1`) : il lui manque une ou plusieurs ressources pour continuer. Il peut être dans l'état *stoppé* (`TASK_STOPPED`), s'il a été stoppé par un `ctrl-Z`. D'autres états existent non détaillés ici (ininterruptible, zombie, swappé).

Un processus peut être dans l'état *prêt* (`TASK_RUNNING=0`) : il a toutes les ressources pour s'exécuter. Il peut être dans l'état *interrompu* (`TASK_INTERRUPTIBLE=1`) : il lui manque une ou plusieurs ressources pour continuer. Il peut être dans l'état *stoppé* (`TASK_STOPPED`), s'il a été stoppé par un `ctrl-Z`. D'autres états existent non détaillés ici (ininterruptible, zombie, swappé).

Un processus peut être dans l'état *prêt* (`TASK_RUNNING=0`) : il a toutes les ressources pour s'exécuter. Il peut être dans l'état *interrompu* (`TASK_INTERRUPTIBLE=1`) : il lui manque une ou plusieurs ressources pour continuer. Il peut être dans l'état *stoppé* (`TASK_STOPPED`), s'il a été stoppé par un `ctrl-Z`. D'autres états existent non détaillés ici (ininterruptible, zombie, swappé).

Un processus peut être dans l'état *prêt* (`TASK_RUNNING=0`) : il a toutes les ressources pour s'exécuter. Il peut être dans l'état *interrompu* (`TASK_INTERRUPTIBLE=1`) : il lui manque une ou plusieurs ressources pour continuer. Il peut être dans l'état *stoppé* (`TASK_STOPPED`), s'il a été stoppé par un `ctrl-Z`. D'autres états existent non détaillés ici (ininterruptible, zombie, swappé).

Un processus peut être dans l'état *prêt* (`TASK_RUNNING=0`) : il a toutes les ressources pour s'exécuter. Il peut être dans l'état *interrompu* (`TASK_INTERRUPTIBLE=1`) : il lui manque une ou plusieurs ressources pour continuer. Il peut être dans l'état *stoppé* (`TASK_STOPPED`), s'il a été stoppé par un `ctrl-Z`. D'autres états existent non détaillés ici (ininterruptible, zombie, swappé).

Un processus peut être dans l'état *prêt* (`TASK_RUNNING=0`) : il a toutes les ressources pour s'exécuter. Il peut être dans l'état *interrompu* (`TASK_INTERRUPTIBLE=1`) : il lui manque une ou plusieurs ressources pour continuer. Il peut être dans l'état *stoppé* (`TASK_STOPPED`), s'il a été stoppé par un `ctrl-Z`. D'autres états existent non détaillés ici (ininterruptible, zombie, swappé).

L'exécution d'un programme est une suite de passages en U.C. et d'attente d'E/S ou d'U.C. Si le programme fait beaucoup d'E/S, la durée d'un passage est courte. Si le programme fait beaucoup de calcul et peu d'E/S, la durée d'un passage est plus longue. Le *scheduler* (programme système chargé de cette gestion) explore la liste des processus prêts et attribue l'U.C. au processus prêt ayant la plus haute priorité qui est exécuté jusqu'à une entrée/sortie (appels système `read` ou `write` par exemple) ou jusqu'à ce que le quantum de temps (100 ms par exemple) soit épuisé. La liste des processus prêts est alors à nouveau explorée à la recherche du processus de plus haute priorité qui peut être le même. La priorité évolue au fil du temps et se compose d'une priorité statique (20 par défaut) et d'une priorité dynamique qui tient compte du temps de passage.

L'exécution d'un programme est une suite de passages en U.C. et d'attente d'E/S ou d'U.C. Si le programme fait beaucoup d'E/S, la durée d'un passage est courte. Si le programme fait beaucoup de calcul et peu d'E/S, la durée d'un passage est plus longue. Le *scheduler* (programme système chargé de cette gestion) explore la liste des processus prêts et attribue l'U.C. au processus prêt ayant la plus haute priorité qui est exécuté jusqu'à une entrée/sortie (appels système `read` ou `write` par exemple) ou jusqu'à ce que le quantum de temps (100 ms par exemple) soit épuisé. La liste des processus prêts est alors à nouveau explorée à la recherche du processus de plus haute priorité qui peut être le même. La priorité évolue au fil du temps et se compose d'une priorité statique (20 par défaut) et d'une priorité dynamique qui tient compte du temps de passage.

L'exécution d'un programme est une suite de passages en U.C. et d'attente d'E/S ou d'U.C. Si le programme fait beaucoup d'E/S, la durée d'un passage est courte. Si le programme fait beaucoup de calcul et peu d'E/S, la durée d'un passage est plus longue. Le *scheduler* (programme système chargé de cette gestion) explore la liste des processus prêts et attribue l'U.C. au processus prêt ayant la plus haute priorité qui est exécuté jusqu'à une entrée/sortie (appels système `read` ou `write` par exemple) ou jusqu'à ce que le quantum de temps (100 ms par exemple) soit épuisé. La liste des processus prêts est alors à nouveau explorée à la recherche du processus de plus haute priorité qui peut être le même. La priorité évolue au fil du temps et se compose d'une priorité statique (20 par défaut) et d'une priorité dynamique qui tient compte du temps de passage.

L'exécution d'un programme est une suite de passages en U.C. et d'attente d'E/S ou d'U.C. Si le programme fait beaucoup d'E/S, la durée d'un passage est courte. Si le programme fait beaucoup de calcul et peu d'E/S, la durée d'un passage est plus longue. Le *scheduler* (programme système chargé de cette gestion) explore la liste des processus prêts et attribue l'U.C. au processus prêt ayant la plus haute priorité qui est exécuté jusqu'à une entrée/sortie (appels système `read` ou `write` par exemple) ou jusqu'à ce que le quantum de temps (100 ms par exemple) soit épuisé. La liste des processus prêts est alors à nouveau explorée à la recherche du processus de plus haute priorité qui peut être le même. La priorité évolue au fil du temps et se compose d'une priorité statique (20 par défaut) et d'une priorité dynamique qui tient compte du temps de passage.

L'exécution d'un programme est une suite de passages en U.C. et d'attente d'E/S ou d'U.C. Si le programme fait beaucoup d'E/S, la durée d'un passage est courte. Si le programme fait beaucoup de calcul et peu d'E/S, la durée d'un passage est plus longue. Le *scheduler* (programme système chargé de cette gestion) explore la liste des processus prêts et attribue l'U.C. au processus prêt ayant la plus haute priorité qui est exécuté jusqu'à une entrée/sortie (appels système `read` ou `write` par exemple) ou jusqu'à ce que le quantum de temps (100 ms par exemple) soit épuisé. La liste des processus prêts est alors à nouveau explorée à la recherche du processus de plus haute priorité qui peut être le même. La priorité évolue au fil du temps et se compose d'une priorité statique (20 par défaut) et d'une priorité dynamique qui tient compte du temps de passage.

L'exécution d'un programme est une suite de passages en U.C. et d'attente d'E/S ou d'U.C. Si le programme fait beaucoup d'E/S, la durée d'un passage est courte. Si le programme fait beaucoup de calcul et peu d'E/S, la durée d'un passage est plus longue. Le *scheduler* (programme système chargé de cette gestion) explore la liste des processus prêts et attribue l'U.C. au processus prêt ayant la plus haute priorité qui est exécuté jusqu'à une entrée/sortie (appels système `read` ou `write` par exemple) ou jusqu'à ce que le quantum de temps (100 ms par exemple) soit épuisé. La liste des processus prêts est alors à nouveau explorée à la recherche du processus de plus haute priorité qui peut être le même. La priorité évolue au fil du temps et se compose d'une priorité statique (20 par défaut) et d'une priorité dynamique qui tient compte du temps de passage.

L'exécution d'un programme est une suite de passages en U.C. et d'attente d'E/S ou d'U.C. Si le programme fait beaucoup d'E/S, la durée d'un passage est courte. Si le programme fait beaucoup de calcul et peu d'E/S, la durée d'un passage est plus longue. Le *scheduler* (programme système chargé de cette gestion) explore la liste des processus prêts et attribue l'U.C. au processus prêt ayant la plus haute priorité qui est exécuté jusqu'à une entrée/sortie (appels système `read` ou `write` par exemple) ou jusqu'à ce que le quantum de temps (100 ms par exemple) soit épuisé. La liste des processus prêts est alors à nouveau explorée à la recherche du processus de plus haute priorité qui peut être le même. La priorité évolue au fil du temps et se compose d'une priorité statique (20 par défaut) et d'une priorité dynamique qui tient compte du temps de passage.

Les processus bloqués en attente d'un événement sont dans la file des processus interrompus ou endormis (`state=1`). Quand l'événement se produit, ils sont remis dans la file des processus prêts. Si le processus était en attente d'une lecture au clavier (`scanf`) par exemple, il est inséré dans la liste des processus prêts lorsque l'entrée est terminée et peut donc être choisi s'il a la plus haute priorité.

Les processus bloqués en attente d'un événement sont dans la file des processus interrompus ou endormis (`state=1`). Quand l'événement se produit, ils sont remis dans la file des processus prêts. Si le processus était en attente d'une lecture au clavier (`scanf`) par exemple, il est inséré dans la liste des processus prêts lorsque l'entrée est terminée et peut donc être choisi s'il a la plus haute priorité.

Les processus bloqués en attente d'un événement sont dans la file des processus interrompus ou endormis (`state=1`). Quand l'événement se produit, ils sont remis dans la file des processus prêts. Si le processus était en attente d'une lecture au clavier (`scanf`) par exemple, il est inséré dans la liste des processus prêts lorsque l'entrée est terminée et peut donc être choisi s'il a la plus haute priorité.

Lors d'un changement de processus en U.C., il faut minimiser les déplacements d'informations pour pouvoir exécuter le nouveau processus. Ce changement de contexte s'effectue en rangeant dans la structure du processus courant, les informations du processus (valeurs des registres, compteur ordinal repérant la prochaine instruction, en langage machine, à exécuter, *etc.*), et en les remplaçant par les valeurs similaires contenues dans le nouveau processus à exécuter.

Lors d'un changement de processus en U.C., il faut minimiser les déplacements d'informations pour pouvoir exécuter le nouveau processus. Ce changement de contexte s'effectue en rangeant dans la structure du processus courant, les informations du processus (valeurs des registres, compteur ordinal repérant la prochaine instruction, en langage machine, à exécuter, *etc.*), et en les remplaçant par les valeurs similaires contenues dans le nouveau processus à exécuter.

Lors d'un changement de processus en U.C., il faut minimiser les déplacements d'informations pour pouvoir exécuter le nouveau processus. Ce changement de contexte s'effectue en rangeant dans la structure du processus courant, les informations du processus (valeurs des registres, compteur ordinal repérant la prochaine instruction, en langage machine, à exécuter, *etc.*), et en les remplaçant par les valeurs similaires contenues dans le nouveau processus à exécuter.

La priorité statique normale est de 20. Elle peut être modifiée à l'aide de la commande `nice`. La priorité dynamique évolue de 20 à 0 cycliquement. Voyons l'exemple suivant : supposons que l'on ait trois processus 635, 636 et 637. Chaque processus fait une simple boucle sans entrée-sortie. Les trois processus ont une priorité de 20 et peuvent s'excuter pendant cinq unités de 20 ms (100 ms) sans être interrompus. Le processus 635 début son exécution pendant 5 unités. Sa priorité dynamique décroît de 20 à 16. Il est interrompu. L'examen de la liste des processus prêts donne l'U.C. au processus 636 pendant 5 unités ; sa priorité est alors de 16. Un nouvel examen de la liste des processus prêts permet à 637 d'être choisi. À la fin du temps d'exécution de 637, tous les processus ont une priorité de 16. Le premier de la liste est choisi : il s'exécute et sa priorité passe de 16 à 11, *etc.* Lorsque les processus ont tous une priorité dynamique égale à 0, on réinitialise celle-ci avec la priorité statique et on recommance. Dans la réalité, les processus sont souvent interrompus et les priorités ne sont pas aussi régulières que l'exemple peut le laisser penser.

La priorité statique normale est de 20. Elle peut être modifiée à l'aide de la commande `nice`. La priorité dynamique évolue de 20 à 0 cycliquement. Voyons l'exemple suivant : supposons que l'on ait trois processus 635, 636 et 637. Chaque processus fait une simple boucle sans entrée-sortie. Les trois processus ont une priorité de 20 et peuvent s'exécuter pendant cinq unités de 20 ms (100 ms) sans être interrompus. Le processus 635 début son exécution pendant 5 unités. Sa priorité dynamique décroît de 20 à 16. Il est interrompu. L'examen de la liste des processus prêts donne l'U.C. au processus 636 pendant 5 unités ; sa priorité est alors de 16. Un nouvel examen de la liste des processus prêts permet à 637 d'être choisi. À la fin du temps d'exécution de 637, tous les processus ont une priorité de 16. Le premier de la liste est choisi : il s'exécute et sa priorité passe de 16 à 11, *etc.* Lorsque les processus ont tous une priorité dynamique égale à 0, on réinitialise celle-ci avec la priorité statique et on recommance. Dans la réalité, les processus sont souvent interrompus et les priorités ne sont pas aussi régulières que l'exemple peut le laisser penser.

La priorité statique normale est de 20. Elle peut être modifiée à l'aide de la commande `nice`. La priorité dynamique évolue de 20 à 0 cycliquement. Voyons l'exemple suivant : supposons que l'on ait trois processus 635, 636 et 637. Chaque processus fait une simple boucle sans entrée-sortie. Les trois processus ont une priorité de 20 et peuvent s'excuter pendant cinq unités de 20 ms (100 ms) sans être interrompus. Le processus 635 début son exécution pendant 5 unités. Sa priorité dynamique décroît de 20 à 16. Il est interrompu. L'examen de la liste des processus prêts donne l'U.C. au processus 636 pendant 5 unités ; sa priorité est alors de 16. Un nouvel examen de la liste des processus prêts permet à 637 d'être choisi. À la fin du temps d'exécution de 637, tous les processus ont une priorité de 16. Le premier de la liste est choisi : il s'exécute et sa priorité passe de 16 à 11, *etc.* Lorsque les processus ont tous une priorité dynamique égale à 0, on réinitialise celle-ci avec la priorité statique et on recommance. Dans la réalité, les processus sont souvent interrompus et les priorités ne sont pas aussi régulières que l'exemple peut le laisser penser.

La priorité statique normale est de 20. Elle peut être modifiée à l'aide de la commande `nice`. La priorité dynamique évolue de 20 à 0 cycliquement. Voyons l'exemple suivant : supposons que l'on ait trois processus 635, 636 et 637. Chaque processus fait une simple boucle sans entrée-sortie. Les trois processus ont une priorité de 20 et peuvent s'excuter pendant cinq unités de 20 ms (100 ms) sans être interrompus. Le processus 635 début son exécution pendant 5 unités. Sa priorité dynamique décroît de 20 à 16. Il est interrompu. L'examen de la liste des processus prêts donne l'U.C. au processus 636 pendant 5 unités ; sa priorité est alors de 16. Un nouvel examen de la liste des processus prêts permet à 637 d'être choisi. À la fin du temps d'exécution de 637, tous les processus ont une priorité de 16. Le premier de la liste est choisi : il s'exécute et sa priorité passe de 16 à 11, *etc.* Lorsque les processus ont tous une priorité dynamique égale à 0, on réinitialise celle-ci avec la priorité statique et on recommance. Dans la réalité, les processus sont souvent interrompus et les priorités ne sont pas aussi régulières que l'exemple peut le laisser penser.

La priorité statique normale est de 20. Elle peut être modifiée à l'aide de la commande `nice`. La priorité dynamique évolue de 20 à 0 cycliquement. Voyons l'exemple suivant : supposons que l'on ait trois processus 635, 636 et 637. Chaque processus fait une simple boucle sans entrée-sortie. Les trois processus ont une priorité de 20 et peuvent s'excuter pendant cinq unités de 20 ms (100 ms) sans être interrompus. Le processus 635 début son exécution pendant 5 unités. Sa priorité dynamique décroît de 20 à 16. Il est interrompu. L'examen de la liste des processus prêts donne l'U.C. au processus 636 pendant 5 unités ; sa priorité est alors de 16. Un nouvel examen de la liste des processus prêts permet à 637 d'être choisi. À la fin du temps d'exécution de 637, tous les processus ont une priorité de 16. Le premier de la liste est choisi : il s'exécute et sa priorité passe de 16 à 11, *etc.* Lorsque les processus ont tous une priorité dynamique égale à 0, on réinitialise celle-ci avec la priorité statique et on recommance. Dans la réalité, les processus sont souvent interrompus et les priorités ne sont pas aussi régulières que l'exemple peut le laisser penser.

La priorité statique normale est de 20. Elle peut être modifiée à l'aide de la commande `nice`. La priorité dynamique évolue de 20 à 0 cycliquement. Voyons l'exemple suivant : supposons que l'on ait trois processus 635, 636 et 637. Chaque processus fait une simple boucle sans entrée-sortie. Les trois processus ont une priorité de 20 et peuvent s'excuter pendant cinq unités de 20 ms (100 ms) sans être interrompus. Le processus 635 début son exécution pendant 5 unités. Sa priorité dynamique décroît de 20 à 16. Il est interrompu. L'examen de la liste des processus prêts donne l'U.C. au processus 636 pendant 5 unités ; sa priorité est alors de 16. Un nouvel examen de la liste des processus prêts permet à 637 d'être choisi. À la fin du temps d'exécution de 637, tous les processus ont une priorité de 16. Le premier de la liste est choisi : il s'exécute et sa priorité passe de 16 à 11, *etc.* Lorsque les processus ont tous une priorité dynamique égale à 0, on réinitialise celle-ci avec la priorité statique et on recommance. Dans la réalité, les processus sont souvent interrompus et les priorités ne sont pas aussi régulières que l'exemple peut le laisser penser.

La priorité statique normale est de 20. Elle peut être modifiée à l'aide de la commande `nice`. La priorité dynamique évolue de 20 à 0 cycliquement. Voyons l'exemple suivant : supposons que l'on ait trois processus 635, 636 et 637. Chaque processus fait une simple boucle sans entrée-sortie. Les trois processus ont une priorité de 20 et peuvent s'excuter pendant cinq unités de 20 ms (100 ms) sans être interrompus. Le processus 635 début son exécution pendant 5 unités. Sa priorité dynamique décroît de 20 à 16. Il est interrompu. L'examen de la liste des processus prêts donne l'U.C. au processus 636 pendant 5 unités ; sa priorité est alors de 16. Un nouvel examen de la liste des processus prêts permet à 637 d'être choisi. À la fin du temps d'exécution de 637, tous les processus ont une priorité de 16. Le premier de la liste est choisi : il s'exécute et sa priorité passe de 16 à 11, *etc.* Lorsque les processus ont tous une priorité dynamique égale à 0, on réinitialise celle-ci avec la priorité statique et on recommance. Dans la réalité, les processus sont souvent interrompus et les priorités ne sont pas aussi régulières que l'exemple peut le laisser penser.

La priorité statique normale est de 20. Elle peut être modifiée à l'aide de la commande `nice`. La priorité dynamique évolue de 20 à 0 cycliquement. Voyons l'exemple suivant : supposons que l'on ait trois processus 635, 636 et 637. Chaque processus fait une simple boucle sans entrée-sortie. Les trois processus ont une priorité de 20 et peuvent s'excuter pendant cinq unités de 20 ms (100 ms) sans être interrompus. Le processus 635 début son exécution pendant 5 unités. Sa priorité dynamique décroît de 20 à 16. Il est interrompu. L'examen de la liste des processus prêts donne l'U.C. au processus 636 pendant 5 unités ; sa priorité est alors de 16. Un nouvel examen de la liste des processus prêts permet à 637 d'être choisi. À la fin du temps d'exécution de 637, tous les processus ont une priorité de 16. Le premier de la liste est choisi : il s'exécute et sa priorité passe de 16 à 11, *etc.* Lorsque les processus ont tous une priorité dynamique égale à 0, on réinitialise celle-ci avec la priorité statique et on recommance. Dans la réalité, les processus sont souvent interrompus et les priorités ne sont pas aussi régulières que l'exemple peut le laisser penser.

La priorité statique normale est de 20. Elle peut être modifiée à l'aide de la commande `nice`. La priorité dynamique évolue de 20 à 0 cycliquement. Voyons l'exemple suivant : supposons que l'on ait trois processus 635, 636 et 637. Chaque processus fait une simple boucle sans entrée-sortie. Les trois processus ont une priorité de 20 et peuvent s'excuter pendant cinq unités de 20 ms (100 ms) sans être interrompus. Le processus 635 début son exécution pendant 5 unités. Sa priorité dynamique décroît de 20 à 16. Il est interrompu. L'examen de la liste des processus prêts donne l'U.C. au processus 636 pendant 5 unités ; sa priorité est alors de 16. Un nouvel examen de la liste des processus prêts permet à 637 d'être choisi. À la fin du temps d'exécution de 637, tous les processus ont une priorité de 16. Le premier de la liste est choisi : il s'exécute et sa priorité passe de 16 à 11, *etc.* Lorsque les processus ont tous une priorité dynamique égale à 0, on réinitialise celle-ci avec la priorité statique et on recommance. Dans la réalité, les processus sont souvent interrompus et les priorités ne sont pas aussi régulières que l'exemple peut le laisser penser.

La priorité statique normale est de 20. Elle peut être modifiée à l'aide de la commande `nice`. La priorité dynamique évolue de 20 à 0 cycliquement. Voyons l'exemple suivant : supposons que l'on ait trois processus 635, 636 et 637. Chaque processus fait une simple boucle sans entrée-sortie. Les trois processus ont une priorité de 20 et peuvent s'excuter pendant cinq unités de 20 ms (100 ms) sans être interrompus. Le processus 635 début son exécution pendant 5 unités. Sa priorité dynamique décroît de 20 à 16. Il est interrompu. L'examen de la liste des processus prêts donne l'U.C. au processus 636 pendant 5 unités ; sa priorité est alors de 16. Un nouvel examen de la liste des processus prêts permet à 637 d'être choisi. À la fin du temps d'exécution de 637, tous les processus ont une priorité de 16. Le premier de la liste est choisi : il s'exécute et sa priorité passe de 16 à 11, *etc.* Lorsque les processus ont tous une priorité dynamique égale à 0, on réinitialise celle-ci avec la priorité statique et on recommance. Dans la réalité, les processus sont souvent interrompus et les priorités ne sont pas aussi régulières que l'exemple peut le laisser penser.

La priorité statique normale est de 20. Elle peut être modifiée à l'aide de la commande `nice`. La priorité dynamique évolue de 20 à 0 cycliquement. Voyons l'exemple suivant : supposons que l'on ait trois processus 635, 636 et 637. Chaque processus fait une simple boucle sans entrée-sortie. Les trois processus ont une priorité de 20 et peuvent s'excuter pendant cinq unités de 20 ms (100 ms) sans être interrompus. Le processus 635 début son exécution pendant 5 unités. Sa priorité dynamique décroît de 20 à 16. Il est interrompu. L'examen de la liste des processus prêts donne l'U.C. au processus 636 pendant 5 unités ; sa priorité est alors de 16. Un nouvel examen de la liste des processus prêts permet à 637 d'être choisi. À la fin du temps d'exécution de 637, tous les processus ont une priorité de 16. Le premier de la liste est choisi : il s'exécute et sa priorité passe de 16 à 11, *etc.* Lorsque les processus ont tous une priorité dynamique égale à 0, on réinitialise celle-ci avec la priorité statique et on recommance. Dans la réalité, les processus sont souvent interrompus et les priorités ne sont pas aussi régulières que l'exemple peut le laisser penser.

La priorité statique normale est de 20. Elle peut être modifiée à l'aide de la commande `nice`. La priorité dynamique évolue de 20 à 0 cycliquement. Voyons l'exemple suivant : supposons que l'on ait trois processus 635, 636 et 637. Chaque processus fait une simple boucle sans entrée-sortie. Les trois processus ont une priorité de 20 et peuvent s'excuter pendant cinq unités de 20 ms (100 ms) sans être interrompus. Le processus 635 début son exécution pendant 5 unités. Sa priorité dynamique décroît de 20 à 16. Il est interrompu. L'examen de la liste des processus prêts donne l'U.C. au processus 636 pendant 5 unités ; sa priorité est alors de 16. Un nouvel examen de la liste des processus prêts permet à 637 d'être choisi. À la fin du temps d'exécution de 637, tous les processus ont une priorité de 16. Le premier de la liste est choisi : il s'exécute et sa priorité passe de 16 à 11, *etc.* Lorsque les processus ont tous une priorité dynamique égale à 0, on réinitialise celle-ci avec la priorité statique et on recommance. Dans la réalité, les processus sont souvent interrompus et les priorités ne sont pas aussi régulières que l'exemple peut le laisser penser.

La priorité statique normale est de 20. Elle peut être modifiée à l'aide de la commande `nice`. La priorité dynamique évolue de 20 à 0 cycliquement. Voyons l'exemple suivant : supposons que l'on ait trois processus 635, 636 et 637. Chaque processus fait une simple boucle sans entrée-sortie. Les trois processus ont une priorité de 20 et peuvent s'excuter pendant cinq unités de 20 ms (100 ms) sans être interrompus. Le processus 635 début son exécution pendant 5 unités. Sa priorité dynamique décroît de 20 à 16. Il est interrompu. L'examen de la liste des processus prêts donne l'U.C. au processus 636 pendant 5 unités ; sa priorité est alors de 16. Un nouvel examen de la liste des processus prêts permet à 637 d'être choisi. À la fin du temps d'exécution de 637, tous les processus ont une priorité de 16. Le premier de la liste est choisi : il s'exécute et sa priorité passe de 16 à 11, *etc.* Lorsque les processus ont tous une priorité dynamique égale à 0, on réinitialise celle-ci avec la priorité statique et on recommance. Dans la réalité, les processus sont souvent interrompus et les priorités ne sont pas aussi régulières que l'exemple peut le laisser penser.

La priorité statique normale est de 20. Elle peut être modifiée à l'aide de la commande `nice`. La priorité dynamique évolue de 20 à 0 cycliquement. Voyons l'exemple suivant : supposons que l'on ait trois processus 635, 636 et 637. Chaque processus fait une simple boucle sans entrée-sortie. Les trois processus ont une priorité de 20 et peuvent s'excuter pendant cinq unités de 20 ms (100 ms) sans être interrompus. Le processus 635 début son exécution pendant 5 unités. Sa priorité dynamique décroît de 20 à 16. Il est interrompu. L'examen de la liste des processus prêts donne l'U.C. au processus 636 pendant 5 unités ; sa priorité est alors de 16. Un nouvel examen de la liste des processus prêts permet à 637 d'être choisi. À la fin du temps d'exécution de 637, tous les processus ont une priorité de 16. Le premier de la liste est choisi : il s'exécute et sa priorité passe de 16 à 11, *etc*. Lorsque les processus ont tous une priorité dynamique égale à 0, on réinitialise celle-ci avec la priorité statique et on recommance. Dans la réalité, les processus sont souvent interrompus et les priorités ne sont pas aussi régulières que l'exemple peut le laisser penser.

La priorité statique normale est de 20. Elle peut être modifiée à l'aide de la commande `nice`. La priorité dynamique évolue de 20 à 0 cycliquement. Voyons l'exemple suivant : supposons que l'on ait trois processus 635, 636 et 637. Chaque processus fait une simple boucle sans entrée-sortie. Les trois processus ont une priorité de 20 et peuvent s'excuter pendant cinq unités de 20 ms (100 ms) sans être interrompus. Le processus 635 début son exécution pendant 5 unités. Sa priorité dynamique décroît de 20 à 16. Il est interrompu. L'examen de la liste des processus prêts donne l'U.C. au processus 636 pendant 5 unités ; sa priorité est alors de 16. Un nouvel examen de la liste des processus prêts permet à 637 d'être choisi. À la fin du temps d'exécution de 637, tous les processus ont une priorité de 16. Le premier de la liste est choisi : il s'exécute et sa priorité passe de 16 à 11, *etc.* Lorsque les processus ont tous une priorité dynamique égale à 0, on réinitialise celle-ci avec la priorité statique et on recommance. Dans la réalité, les processus sont souvent interrompus et les priorités ne sont pas aussi régulières que l'exemple peut le laisser penser.

La priorité statique normale est de 20. Elle peut être modifiée à l'aide de la commande `nice`. La priorité dynamique évolue de 20 à 0 cycliquement. Voyons l'exemple suivant : supposons que l'on ait trois processus 635, 636 et 637. Chaque processus fait une simple boucle sans entrée-sortie. Les trois processus ont une priorité de 20 et peuvent s'excuter pendant cinq unités de 20 ms (100 ms) sans être interrompus. Le processus 635 début son exécution pendant 5 unités. Sa priorité dynamique décroît de 20 à 16. Il est interrompu. L'examen de la liste des processus prêts donne l'U.C. au processus 636 pendant 5 unités ; sa priorité est alors de 16. Un nouvel examen de la liste des processus prêts permet à 637 d'être choisi. À la fin du temps d'exécution de 637, tous les processus ont une priorité de 16. Le premier de la liste est choisi : il s'exécute et sa priorité passe de 16 à 11, *etc.* Lorsque les processus ont tous une priorité dynamique égale à 0, on réinitialise celle-ci avec la priorité statique et on recommance.

Dans la réalité, les processus sont souvent interrompus et les priorités ne sont pas aussi régulières que l'exemple peut le laisser penser.

La priorité statique normale est de 20. Elle peut être modifiée à l'aide de la commande `nice`. La priorité dynamique évolue de 20 à 0 cycliquement. Voyons l'exemple suivant : supposons que l'on ait trois processus 635, 636 et 637. Chaque processus fait une simple boucle sans entrée-sortie. Les trois processus ont une priorité de 20 et peuvent s'excuter pendant cinq unités de 20 ms (100 ms) sans être interrompus. Le processus 635 début son exécution pendant 5 unités. Sa priorité dynamique décroît de 20 à 16. Il est interrompu. L'examen de la liste des processus prêts donne l'U.C. au processus 636 pendant 5 unités ; sa priorité est alors de 16. Un nouvel examen de la liste des processus prêts permet à 637 d'être choisi. À la fin du temps d'exécution de 637, tous les processus ont une priorité de 16. Le premier de la liste est choisi : il s'exécute et sa priorité passe de 16 à 11, *etc.* Lorsque les processus ont tous une priorité dynamique égale à 0, on réinitialise celle-ci avec la priorité statique et on recommance. Dans la réalité, les processus sont souvent interrompus et les priorités ne sont pas aussi régulières que l'exemple peut le laisser penser.

La table des processus doit aussi enregistrer la filiation entre les processus. Sous Linux, cette filiation est enregistrée à l'aide de 4 pointeurs de structures de processus (de type `struct struct_task*`). Supposons que l'on ait la filiation suivante : 446 est le père de 495. 495 est le père de 528, 529, 530, 531, et 545. 545 est le père de 548, qui est le père de 575, qui est le père de 576, qui est le père de 588. Cet arbre n-aire est mémorisé sous la forme d'un arbre binaire en utilisant pour chaque nœud de cet arbre, un pointeur vers le père, un pointeur sur le dernier fils (le plus jeune), un pointeur sur le frère immédiatement plus jeune et un pointeur sur le frère immédiatement plus vieux.

La table des processus doit aussi enregistrer la filiation entre les processus. Sous Linux, cette filiation est enregistrée à l'aide de 4 pointeurs de structures de processus (de type `struct struct_task*`). Supposons que l'on ait la filiation suivante : 446 est le père de 495. 495 est le père de 528, 529, 530, 531, et 545. 545 est le père de 548, qui est le père de 575, qui est le père de 576, qui est le père de 588. Cet arbre n-aire est mémorisé sous la forme d'un arbre binaire en utilisant pour chaque nœud de cet arbre, un pointeur vers le père, un pointeur sur le dernier fils (le plus jeune), un pointeur sur le frère immédiatement plus jeune et un pointeur sur le frère immédiatement plus vieux.

La table des processus doit aussi enregistrer la filiation entre les processus. Sous Linux, cette filiation est enregistrée à l'aide de 4 pointeurs de structures de processus (de type `struct struct_task*`). Supposons que l'on ait la filiation suivante : 446 est le père de 495. 495 est le père de 528, 529, 530, 531, et 545. 545 est le père de 548, qui est le père de 575, qui est le père de 576, qui est le père de 588. Cet arbre n-aire est mémorisé sous la forme d'un arbre binaire en utilisant pour chaque nœud de cet arbre, un pointeur vers le père, un pointeur sur le dernier fils (le plus jeune), un pointeur sur le frère immédiatement plus jeune et un pointeur sur le frère immédiatement plus vieux.

La table des processus doit aussi enregistrer la filiation entre les processus. Sous Linux, cette filiation est enregistrée à l'aide de 4 pointeurs de structures de processus (de type `struct struct_task*`). Supposons que l'on ait la filiation suivante : 446 est le père de 495. 495 est le père de 528, 529, 530, 531, et 545. 545 est le père de 548, qui est le père de 575, qui est le père de 576, qui est le père de 588. Cet arbre n-aire est mémorisé sous la forme d'un arbre binaire en utilisant pour chaque nœud de cet arbre, un pointeur vers le père, un pointeur sur le dernier fils (le plus jeune), un pointeur sur le frère immédiatement plus jeune et un pointeur sur le frère immédiatement plus vieux.

La table des processus doit aussi enregistrer la filiation entre les processus. Sous Linux, cette filiation est enregistrée à l'aide de 4 pointeurs de structures de processus (de type `struct struct_task*`). Supposons que l'on ait la filiation suivante : 446 est le père de 495. 495 est le père de 528, 529, 530, 531, et 545. 545 est le père de 548, qui est le père de 575, qui est le père de 576, qui est le père de 588. Cet arbre n-aire est mémorisé sous la forme d'un arbre binaire en utilisant pour chaque nœud de cet arbre, un pointeur vers le père, un pointeur sur le dernier fils (le plus jeune), un pointeur sur le frère immédiatement plus jeune et un pointeur sur le frère immédiatement plus vieux.

La table des processus doit aussi enregistrer la filiation entre les processus. Sous Linux, cette filiation est enregistrée à l'aide de 4 pointeurs de structures de processus (de type `struct struct_task*`). Supposons que l'on ait la filiation suivante : 446 est le père de 495. 495 est le père de 528, 529, 530, 531, et 545. 545 est le père de 548, qui est le père de 575, qui est le père de 576, qui est le père de 588. Cet arbre n-aire est mémorisé sous la forme d'un arbre binaire en utilisant pour chaque nœud de cet arbre, un pointeur vers le père, un pointeur sur le dernier fils (le plus jeune), un pointeur sur le frère immédiatement plus jeune et un pointeur sur le frère immédiatement plus vieux.

La table des processus doit aussi enregistrer la filiation entre les processus. Sous Linux, cette filiation est enregistrée à l'aide de 4 pointeurs de structures de processus (de type `struct struct_task*`). Supposons que l'on ait la filiation suivante : 446 est le père de 495. 495 est le père de 528, 529, 530, 531, et 545. 545 est le père de 548, qui est le père de 575, qui est le père de 576, qui est le père de 588. Cet arbre n-aire est mémorisé sous la forme d'un arbre binaire en utilisant pour chaque nœud de cet arbre, un pointeur vers le père, un pointeur sur le dernier fils (le plus jeune), un pointeur sur le frère immédiatement plus jeune et un pointeur sur le frère immédiatement plus vieux.

La table des processus doit aussi enregistrer la filiation entre les processus. Sous Linux, cette filiation est enregistrée à l'aide de 4 pointeurs de structures de processus (de type `struct struct_task*`). Supposons que l'on ait la filiation suivante : 446 est le père de 495. 495 est le père de 528, 529, 530, 531, et 545. 545 est le père de 548, qui est le père de 575, qui est le père de 576, qui est le père de 588. Cet arbre n-aire est mémorisé sous la forme d'un arbre binaire en utilisant pour chaque nœud de cet arbre, un pointeur vers le père, un pointeur sur le dernier fils (le plus jeune), un pointeur sur le frère immédiatement plus jeune et un pointeur sur le frère immédiatement plus vieux.

La table des processus doit aussi enregistrer la filiation entre les processus. Sous Linux, cette filiation est enregistrée à l'aide de 4 pointeurs de structures de processus (de type `struct struct_task*`). Supposons que l'on ait la filiation suivante : 446 est le père de 495. 495 est le père de 528, 529, 530, 531, et 545. 545 est le père de 548, qui est le père de 575, qui est le père de 576, qui est le père de 588. Cet arbre n-aire est mémorisé sous la forme d'un arbre binaire en utilisant pour chaque nœud de cet arbre, un pointeur vers le père, un pointeur sur le dernier fils (le plus jeune), un pointeur sur le frère immédiatement plus jeune et un pointeur sur le frère immédiatement plus vieux.

La table des processus doit aussi enregistrer la filiation entre les processus. Sous Linux, cette filiation est enregistrée à l'aide de 4 pointeurs de structures de processus (de type `struct struct_task*`). Supposons que l'on ait la filiation suivante : 446 est le père de 495. 495 est le père de 528, 529, 530, 531, et 545. 545 est le père de 548, qui est le père de 575, qui est le père de 576, qui est le père de 588. Cet arbre n-aire est mémorisé sous la forme d'un arbre binaire en utilisant pour chaque nœud de cet arbre, un pointeur vers le père, un pointeur sur le dernier fils (le plus jeune), un pointeur sur le frère immédiatement plus jeune et un pointeur sur le frère immédiatement plus vieux.

La table des processus doit aussi enregistrer la filiation entre les processus. Sous Linux, cette filiation est enregistrée à l'aide de 4 pointeurs de structures de processus (de type `struct struct_task*`). Supposons que l'on ait la filiation suivante : 446 est le père de 495. 495 est le père de 528, 529, 530, 531, et 545. 545 est le père de 548, qui est le père de 575, qui est le père de 576, qui est le père de 588. Cet arbre n-aire est mémorisé sous la forme d'un arbre binaire en utilisant pour chaque nœud de cet arbre, un pointeur vers le père, un pointeur sur le dernier fils (le plus jeune), un pointeur sur le frère immédiatement plus jeune et un pointeur sur le frère immédiatement plus vieux.

La table des processus doit aussi enregistrer la filiation entre les processus. Sous Linux, cette filiation est enregistrée à l'aide de 4 pointeurs de structures de processus (de type `struct struct_task*`). Supposons que l'on ait la filiation suivante : 446 est le père de 495. 495 est le père de 528, 529, 530, 531, et 545. 545 est le père de 548, qui est le père de 575, qui est le père de 576, qui est le père de 588. Cet arbre n-aire est mémorisé sous la forme d'un arbre binaire en utilisant pour chaque nœud de cet arbre, un pointeur vers le père, un pointeur sur le dernier fils (le plus jeune), un pointeur sur le frère immédiatement plus jeune et un pointeur sur le frère immédiatement plus vieux.

La table des processus doit aussi enregistrer la filiation entre les processus. Sous Linux, cette filiation est enregistrée à l'aide de 4 pointeurs de structures de processus (de type `struct struct_task*`). Supposons que l'on ait la filiation suivante : 446 est le père de 495. 495 est le père de 528, 529, 530, 531, et 545. 545 est le père de 548, qui est le père de 575, qui est le père de 576, qui est le père de 588. Cet arbre n-aire est mémorisé sous la forme d'un arbre binaire en utilisant pour chaque nœud de cet arbre, un pointeur vers le père, un pointeur sur le dernier fils (le plus jeune), un pointeur sur le frère immédiatement plus jeune et un pointeur sur le frère immédiatement plus vieux.

Dans notre exemple, on a l'arbre suivant : 446 pointe vers son fils 495, celui-ci pointe sur son père (446), sur son fils le plus jeune (545), 545 pointer vers son père (446), vers son fils (548), vers son frère immédiatement plus vieux (531), 531 pointe sur son père, sur son frère immédiatement plus jeune (545) et sur son frère immédiatement plus vieux (530), à son tour 530 pointe vers son père, vers son frère immédiatement plus jeune (531) et vers son frère immédiatement plus jeune (529), *etc.*

Dans notre exemple, on a l'arbre suivant : 446 pointe vers son fils 495, celui-ci pointe sur son père (446), sur son fils le plus jeune (545), 545 pointer vers son père (446), vers son fils (548), vers son frère immédiatement plus vieux (531), 531 pointe sur son père, sur son frère immédiatement plus jeune (545) et sur son frère immédiatement plus vieux (530), à son tour 530 pointe vers son père, vers son frère immédiatement plus jeune (531) et vers son frère immédiatement plus jeune (529), *etc.*

Dans notre exemple, on a l'arbre suivant : 446 pointe vers son fils 495, celui-ci pointe sur son père (446), sur son fils le plus jeune (545), 545 pointer vers son père (446), vers son fils (548), vers son frère immédiatement plus vieux (531), 531 pointe sur son père, sur son frère immédiatement plus jeune (545) et sur son frère immédiatement plus vieux (530), à son tour 530 pointe vers son père, vers son frère immédiatement plus jeune (531) et vers son frère immédiatement plus jeune (529), *etc.*

Dans notre exemple, on a l'arbre suivant : 446 pointe vers son fils 495, celui-ci pointe sur son père (446), sur son fils le plus jeune (545), 545 pointer vers son père (446), vers son fils (548), vers son frère immédiatement plus vieux (531), 531 pointe sur son père, sur son frère immédiatement plus jeune (545) et sur son frère immédiatement plus vieux (530), à son tour 530 pointe vers son père, vers son frère immédiatement plus jeune (531) et vers son frère immédiatement plus jeune (529), *etc.*

Dans notre exemple, on a l'arbre suivant : 446 pointe vers son fils 495, celui-ci pointe sur son père (446), sur son fils le plus jeune (545), 545 pointer vers son père (446), vers son fils (548), vers son frère immédiatement plus vieux (531), 531 pointe sur son père, sur son frère immédiatement plus jeune (545) et sur son frère immédiatement plus vieux (530), à son tour 530 pointe vers son père, vers son frère immédiatement plus jeune (531) et vers son frère immédiatement plus jeune (529), *etc.*

Dans notre exemple, on a l'arbre suivant : 446 pointe vers son fils 495, celui-ci pointe sur son père (446), sur son fils le plus jeune (545), 545 pointer vers son père (446), vers son fils (548), vers son frère immédiatement plus vieux (531), 531 pointe sur son père, sur son frère immédiatement plus jeune (545) et sur son frère immédiatement plus vieux (530), à son tour 530 pointe vers son père, vers son frère immédiatement plus jeune (531) et vers son frère immédiatement plus jeune (529), *etc.*

Dans notre exemple, on a l'arbre suivant : 446 pointe vers son fils 495, celui-ci pointe sur son père (446), sur son fils le plus jeune (545), 545 pointer vers son père (446), vers son fils (548), vers son frère immédiatement plus vieux (531), 531 pointe sur son père, sur son frère immédiatement plus jeune (545) et sur son frère immédiatement plus vieux (530), à son tour 530 pointe vers son père, vers son frère immédiatement plus jeune (531) et vers son frère immédiatement plus jeune (529), *etc.*

Dans notre exemple, on a l'arbre suivant : 446 pointe vers son fils 495, celui-ci pointe sur son père (446), sur son fils le plus jeune (545), 545 pointer vers son père (446), vers son fils (548), vers son frère immédiatement plus vieux (531), 531 pointe sur son père, sur son frère immédiatement plus jeune (545) et sur son frère immédiatement plus vieux (530), à son tour 530 pointe vers son père, vers son frère immédiatement plus jeune (531) et vers son frère immédiatement plus jeune (529), *etc.*

Dans notre exemple, on a l'arbre suivant : 446 pointe vers son fils 495, celui-ci pointe sur son père (446), sur son fils le plus jeune (545), 545 pointer vers son père (446), vers son fils (548), vers son frère immédiatement plus vieux (531), 531 pointe sur son père, sur son frère immédiatement plus jeune (545) et sur son frère immédiatement plus vieux (530), à son tour 530 pointe vers son père, vers son frère immédiatement plus jeune (531) et vers son frère immédiatement plus jeune (529), *etc.*

Résumé

La gestion des processus et le choix par le scheduler du processus à élire parmi les candidats au passage en U.C. constituent le cœur d'un système d'exploitation. Chaque processus est caractérisé par une structure qui contient des informations sur le processus ou des pointeurs vers d'autres structures qui caractérisent des processus. On y trouve ainsi des pointeurs vers des structures de données mémorisant les fichiers ouverts par le processus, les zones mémoires allouées au processus, la sauvegarde de l'état du processeur (essentiellement les valeurs des registres) lors du changement de contexte.

Résumé

La gestion des processus et le choix par le scheduler du processus à élire parmi les candidats au passage en U.C. constituent le cœur d'un système d'exploitation. Chaque processus est caractérisé par une structure qui contient des informations sur le processus ou des pointeurs vers d'autres structures qui caractérisent des processus. On y trouve ainsi des pointeurs vers des structures de données mémorisant les fichiers ouverts par le processus, les zones mémoires allouées au processus, la sauvegarde de l'état du processeur (essentiellement les valeurs des registres) lors du changement de contexte.

Résumé

La gestion des processus et le choix par le scheduler du processus à élire parmi les candidats au passage en U.C. constituent le cœur d'un système d'exploitation. Chaque processus est caractérisé par une structure qui contient des informations sur le processus ou des pointeurs vers d'autres structures qui caractérisent des processus. On y trouve ainsi des pointeurs vers des structures de données mémorisant les fichiers ouverts par le processus, les zones mémoires allouées au processus, la sauvegarde de l'état du processeur (essentiellement les valeurs des registres) lors du changement de contexte.

Résumé

La gestion des processus et le choix par le scheduler du processus à élire parmi les candidats au passage en U.C. constituent le cœur d'un système d'exploitation. Chaque processus est caractérisé par une structure qui contient des informations sur le processus ou des pointeurs vers d'autres structures qui caractérisent des processus. On y trouve ainsi des pointeurs vers des structures de données mémorisant les fichiers ouverts par le processus, les zones mémoires allouées au processus, la sauvegarde de l'état du processeur (essentiellement les valeurs des registres) lors du changement de contexte.

Résumé

La gestion des processus et le choix par le scheduler du processus à élire parmi les candidats au passage en U.C. constituent le cœur d'un système d'exploitation. Chaque processus est caractérisé par une structure qui contient des informations sur le processus ou des pointeurs vers d'autres structures qui caractérisent des processus. On y trouve ainsi des pointeurs vers des structures de données mémorisant les fichiers ouverts par le processus, les zones mémoires allouées au processus, la sauvegarde de l'état du processeur (essentiellement les valeurs des registres) lors du changement de contexte.