CSI351 – Système d'exploitation
Tutorat 2 – Hiver 2016
Les processus - Solutionnaire

1. En classe, nous avions étudié l'exécution de processus en prenant pour acquis que tous processus sont en mémoire et peuvent être exécuté lorsque prêt. Le diagramme de transitions d'état de la figure 1 montre comment un processus change de l'état en exécution (i.e. le programme du processus se fait exécuter par l'UCT) à l'état prêt (afin de permettre à un autre processus d'exécuter). L'ordonnanceur d'UCT (ordonnanceur) à court terme est responsable de partager l'UCT parmi les processus prêts à exécuter.

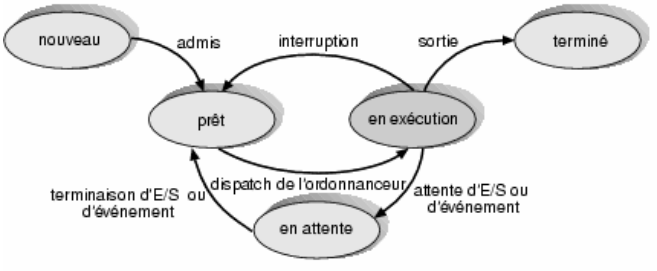


Figure 1

Rappelez-vous que l'ordonnanceur à moyen terme était responsable de faire la permutation (swapping out) des processus pour libérer de la mémoire pour faciliter l'exécution des autres processus tel que montré dans la figure 2. L'utilisation d'un tel ordonnanceur permet d'accroître le nombre de processus que peut gérer un SE.

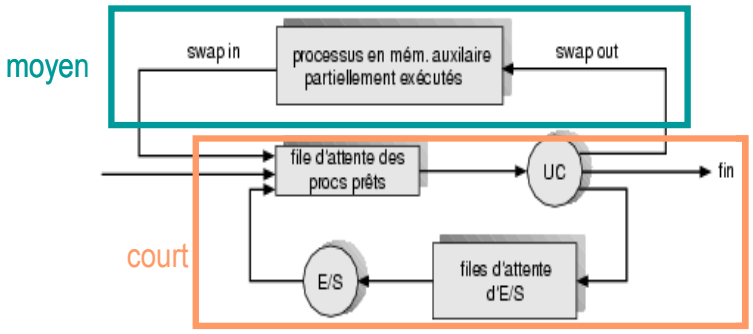
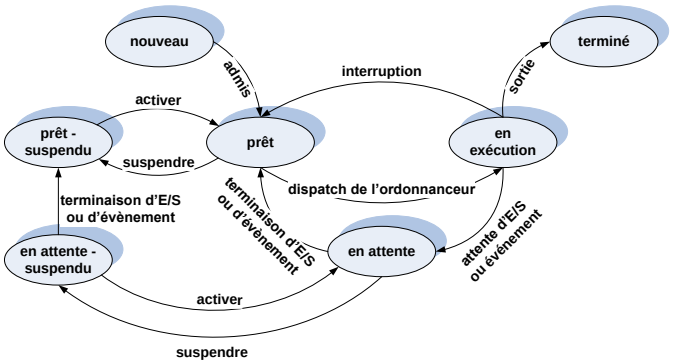


Figure 2

a) Modifiez le diagramme de transitions d'état de la figure 1 en y ajoutant 2 états qui permettra le SE de gérer les processus permutés. Ajoutez les transitions appropriées pour les nouveaux états et décrivez les événements qui engendrent les transitions.
b) Les transitions de l'état prêt ou en attente à l'état terminé sont possibles (ces transitions ne sont pas typiquement montrées dans les diagrammes de transition d'états pour qu'ils soient claires). Dans quels circonstances ont lieu ces transitions?

Un modèle de processus à 7 états



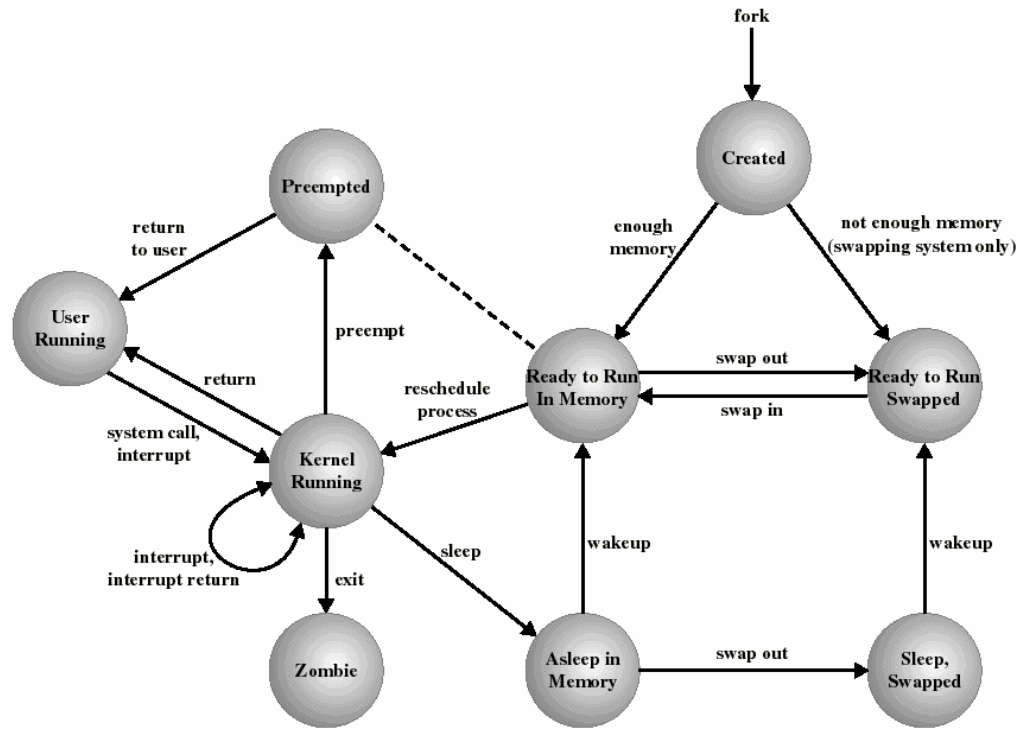
La nécessité de remplacer (swapping)

- **Le SE doit parfois suspendre certains processus, i.e.: les transférer au disque. Donc 2 autres états:**
  - **En attente Suspendu: processus bloqués transférés au disque**
  - **Prêt Suspendu: processus prêts transférés au disque**
- **Notez qu'un processus ne peut pas être permuté lorsqu'il est dans l'état en exécution.**
- **Ces états servent d'abords à représenté l'état des processus au moment où ils sont permuté.**
- **L'état du processus peut changer même s'il est permuté.**

Nouvelles transitions

- **Deux transitions se font lorsqu'un processus est permuté**
  - **En attente --> En attente Suspendu**
    - **Choix privilégié par le SE sont les processus en attente pour libérer de la mémoire**
  - **Prêt --> Prêt Suspendu (rare)**
    - **On doit libérer de la mémoire mais il n'y a plus de processus bloqués en mémoire**
- **En attente Suspendu --> Prêt Suspendu**
  - **Lorsque l'événement attendu se produit (info d'état est disponible au SE) pendant que le processus est permuté**
- **Prêt Suspendu --> Prêt**
  - **Lorsque le processus est ramené en mémoire principale pour se faire exécuter (par exemple, lorsqu'il n'y a plus de processus prêt)**
- **En-attente – suspendu -> en attente**
  - **Cette transition est utilisé si le SE sait que l'événement qui bloque le processus aura lieu bientôt et que le processus à une haute priorité (plus haute que tous processus dans l'état prêt-suspendu).**

Voici un exemple de diagramme de transitions d'états pour un SE réel, du UNIX SVR4 (system V, release 4 – source Stallings)



Quelques notes au sujet du diagramme de transition du système UNIX:

L'état « running » (en exécution) a été divisé en 2 états, pour distinguer les moments lorsque le processus roule en mode noyau des moments qu'il roule en mode utilisateur. Notez que le processus roule en mode noyau lorsqu'il roule du code du SE. En classe, la perspective prise était qu'au moment d'interruption, le SE roule en dehors des processus. Le diagramme de transitions d'états UNIX reflète une autre perspective – que le SE roule dans le contexte d'un processus. L'avantage de cette approche est de réduire le nombre de commutation de processus. Lorsque l'exécution du code noyau est finie et qu'aucune commutation de processus est nécessaire (e.g. la demande d'un appel système peut être satisfaite immédiatement), le contrôle est simplement retourné au code usage (en changeant de mode aussi). Donc le SE est simplement vu comme une collection de sous-programmes qui roulent en mode noyau lorsqu'une interruption matériel ou interruption logicielle a lieu.

L'état zombie correspond à l'état terminé (les processus zombies ont été discutés en classes).

L'état « asleep » correspond à l'état « en attente ».

Notez que lorsqu'il n'existe pas assez de mémoire principale pour rouler un processus à sa création, il est placé en mémoire de permutation (sur le disque rigide).

Notez que pour composer avec les interruptions de matériel en mode noyau, l'interruption est desservie et le contrôle revient au code du noyau qui roulait au moment de l'interruption.

L'état « preempted » et l'état « ready to run in memory » sont effectivement le même état (tel qu'indiqué par la ligne pointillée). Les processus dans un état ou l'autre sont placé dans la même file d'ordonnancement (la file prêt). Quand le noyau est prêt à retourner le contrôle au code d'utilisateur, il peut décider de faire la préemption du processus courant en faveur d'un autre processus avec une priorité plus élevée. Dans ce cas le processus courant est placé dans l'état « preempted ».

2. Examinez la figure 3 qui montre un exemple de files d'attentes d'ordonnancement pour le digramme de transitions de 5 états de la question 1.

   a) Premièrement, ajoutez une étiquette à chacune des transitions (sans étiquette) avec l'action/événement qui déclenche la transition.

   b) Ensuite indiquez lesquelles des transitions et files d'attente sont la responsabilité de chacun des sous-systèmes ci-dessous (c'est-à-dire, la portée de chaque sous-système dans la gestion des processus).

      - L'ordonnanceur de l'UCT (l'ordonnanceur à court terme).
      - Le sous-système de fichier (comprend un ordonnanceur du disque rigide et pilotes pour faire de l'E/S avec chaque appareil du système). Ce sous-système est responsable des demandes de service au système de fichiers tel que l'ouverture de fichiers, la lecture et l'écriture aux fichiers, et la fermeture des fichiers et de tout entrée/sortie avec le matériel de l'ordinateur.
      - Le sous-système IPC: Responsable de toutes fonctions de communications entre processus dans le SE (y compris les signaux).

   c) Modifiez la figure pour accommoder le nouveau diagramme de transition de 7 états de la question 1. Ajoutez l'ordonnanceur à moyen terme aux sous-systèmes du SE pour compléter la figure.

   d) Pour chaque file d'attente, identifier l'état ou les états qu'ont (ou peuvent avoir) les processus lorsque dans la file.

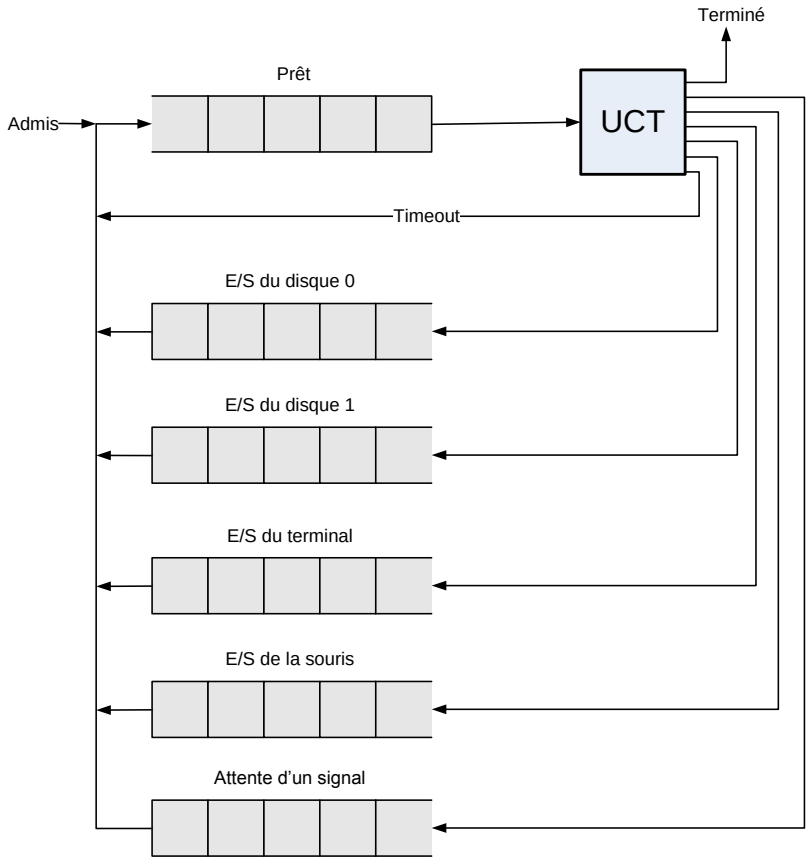
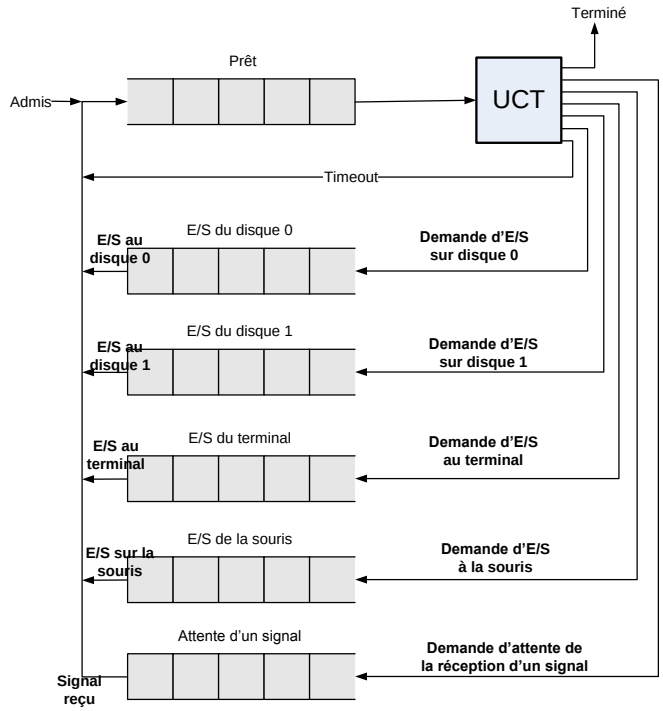
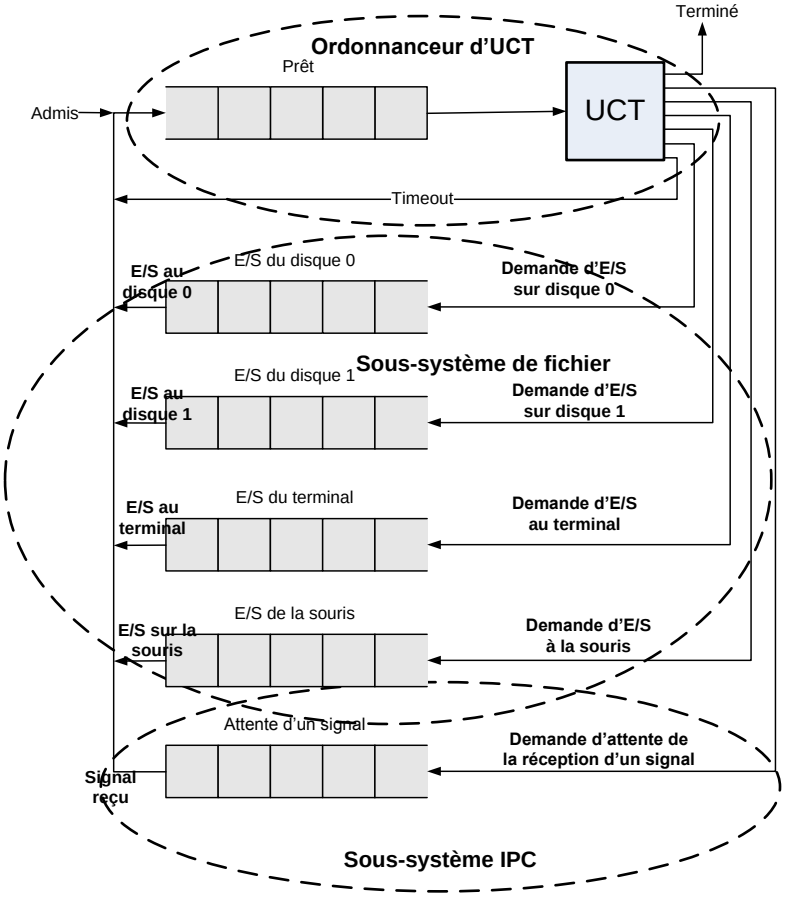


Figure 3

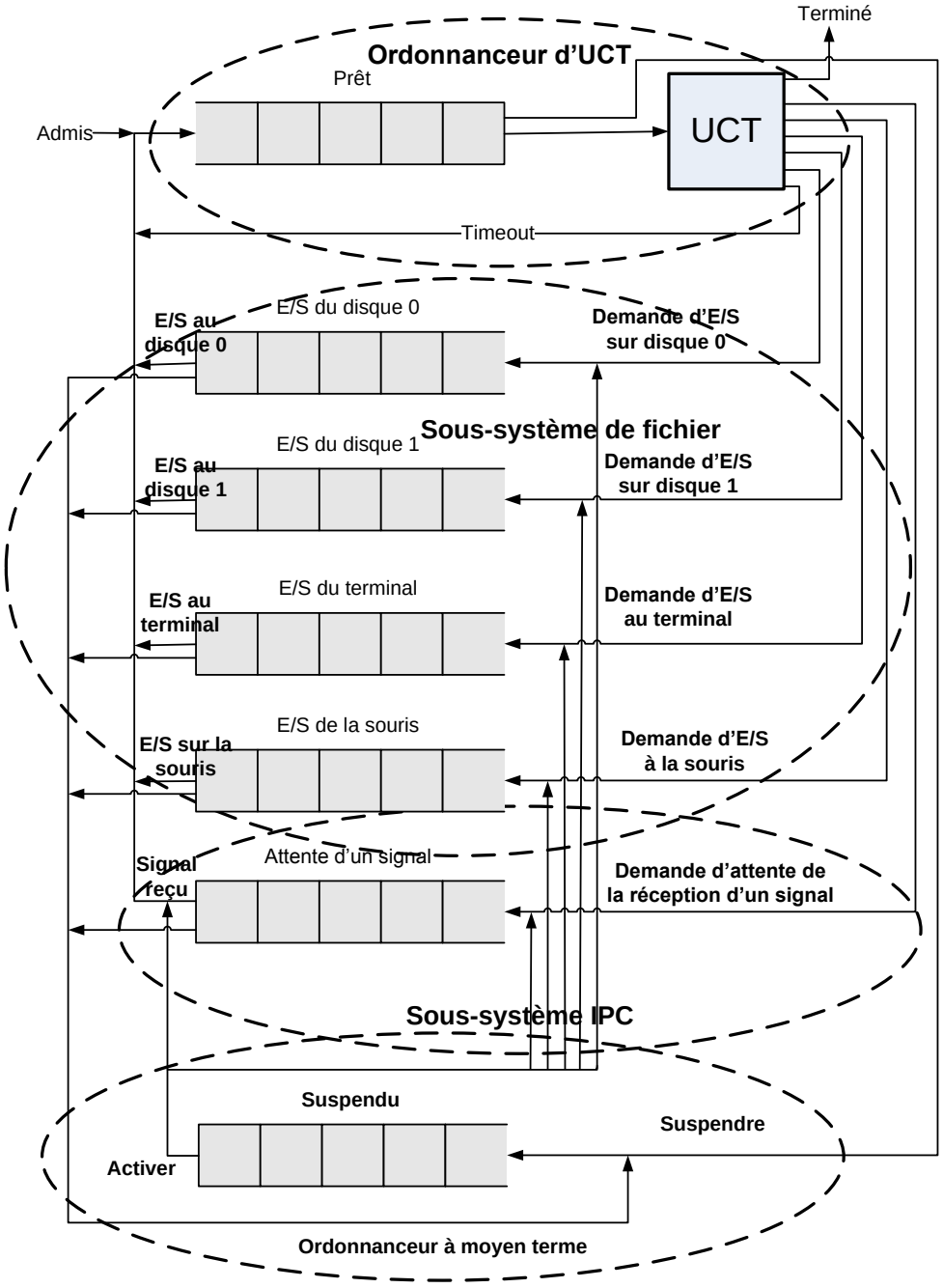
Part a)


Terminé
Prêt
Admis
UCT
Timeout
E/S du disque 0
E/S au disque 0
Demande d'E/S sur disque 0
E/S du disque 1
E/S au disque 1
Demande d'E/S sur disque 1
E/S du terminal
E/S au terminal
Demande d'E/S au terminal
E/S de la souris
E/S sur la souris
Demande d'E/S à la souris
Attente d'un signal
Signal reçu
Demande d'attente de la réception d'un signal

Part b)


Terminé
Ordonnanceur d'UCT
Prêt
Admis
UCT
Timeout
E/S du disque 0
E/S au disque 0
Demande d'E/S sur disque 0
Sous-système de fichier
E/S du disque 1
E/S au disque 1
Demande d'E/S sur disque 1
E/S du terminal
E/S au terminal
Demande d'E/S au terminal
E/S de la souris
E/S sur la souris
Demande d'E/S à la souris
Attente d'un signal
Signal reçu
Demande d'attente de la réception d'un signal
Sous-système IPC

Part c)



Part d)

File d'attente	État
Prêt	**Prêt**
E/S du disque 0	**En attente**
E/S du disque 1	**En attente**
E/S du terminal	**En attente**
E/S de la souris	**En attente**
Attente d'un signal	**En attente**
Suspendu	**En attente/suspendu ou Prêt/suspendu**

À notez qu'il serait possible d'avoir deux files d'attente au line d'une seule file « suspendu » - une file pour chacun des états « en attente/suspendu » et « prêt/suspendu ».

3. La figure 4 est un diagramme simplifié qui montre comment trois processus occupent la mémoire dans un système actif (prenez pour acquis que tous les processus sont prêt à exécuter). Notez qu'une partie de la mémoire est réservé au système d'exploitation. Dans ce problème, deux sous-systèmes du SE sont utilisés : l'ordonnanceur à court terme (le dispatcher) qui sélectionne un processus pour rouler sur l'UCT, et le sous-système de fichier qui s'occupe des demandes d'E/S.

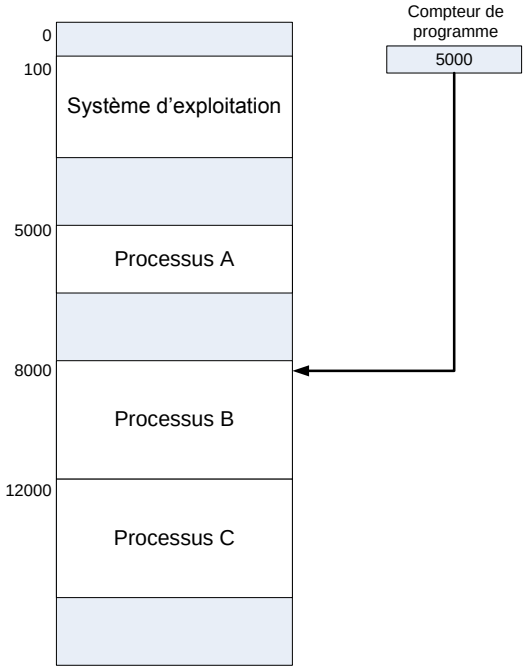


Figure 4

Tableau 1							
Cycle d'instruction	**Adresse**	**Cycle d'instruction**	**Adresse**	**Cycle d'instruction**	**Adresse**	**Cycle d'instruction**	**Adresse**
1	5000	27	8002	53	12058	79	5022
2	5001	28	8003	54	12059	80	5100
3	5002	29	8004	55	12060	81	5101
4	5000	30	8005	56	12100	82	5102
5	5001	31	8006	57	12101	83	300
6	5003	32	400	58	12102	84	301
7	5004	33	401	59	300	85	302
8	5005	34	402	60	301	86	303
9	5006	35	300	61	302	87	12103
10	5010	36	301	62	303	88	12061
11	5011	37	302	63	5101	89	12062
12	5012	38	303	64	5102	90	12063
13	5010	39	12000	65	5103	91	12074
14	5011	40	12001	66	5014	92	12075
15	5012	41	12002	67	5015	93	12076
16	5010	42	12003	68	5015	94	12077
17	5011	43	12004	69	5020	95	12074
18	5012	44	12053	70	5021	96	12075
19	5013	45	12054	71	5022	97	12076
20	5100	46	12055	72	5100	99	12077
21	300	47	12056	73	5101	99	12080
22	301	48	12057	74	5102	100	12081
23	302	49	12100	75	5103	101	12082
24	303	50	12101	76	5022	102	12100
25	8000	51	12102	77	5020	103	12101
26	8001	52	12103	78	5021	104	12102

Le tableau 1 donne le portrait de l'exécution de 104 cycles d'instruction du système actif. L'adresse des instructions est donnée pour chaque cycle (notez que pour simplifier l'exemple chaque instruction occupe une seule adresse). Prenez pour acquis que chaque cycle ne prend qu'une unité de temps pour son exécution. Les dépassements de temps et les demandes d'E/S ont lieu aux temps suivants :

a) Après le cycle 20: Dépassement de temps de processus (le processus a dépassé son temps alloué avec l'UCT).
b) Après le cycle 31: Le processus demande un E/S sur le disque rigide
c) Après le cycle 58: Dépassement de temps de processus
d) Après le cycle 82: Dépassement de temps de processus

Complétez la figure 5 qui devra montrer les changements d'états pour chaque processus durant le roulement du système. Pour chaque processus remplissez la barre pour montrer l'état du processus aux différents temps en vous servant de la légende suivante :

Pour le système d'exploitation, montrez les temps que son code exécute. (Pour simplifier les choses, prenez pour acquis qu'un processus assume l'état prêt ou en attente aussitôt qu'il a fini d'exécuter, i.e., quand le SE commence son exécution, et à l'état prêt aussitôt que le SE a complété l'exécution de son code).

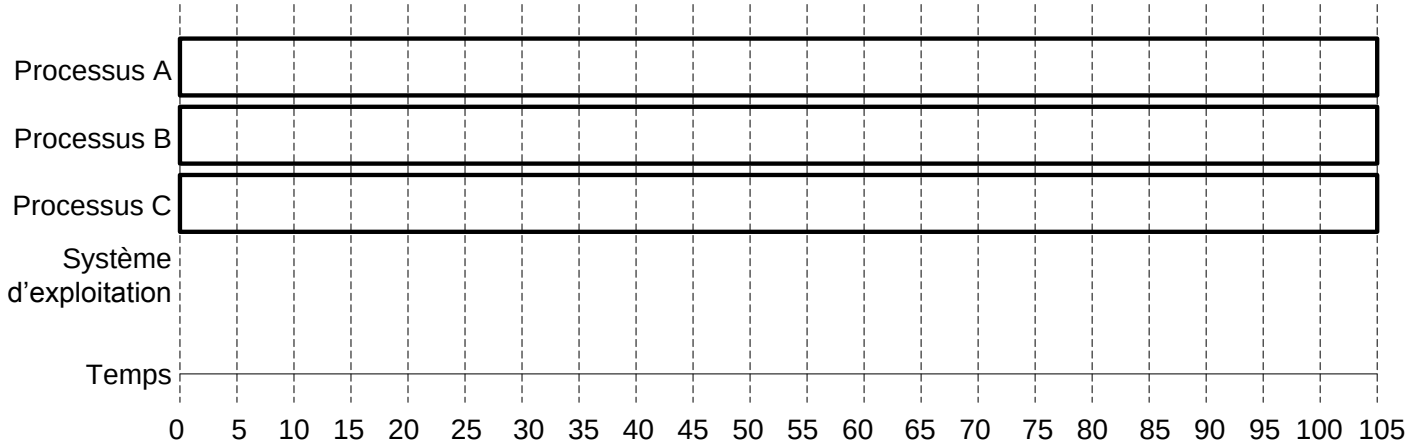
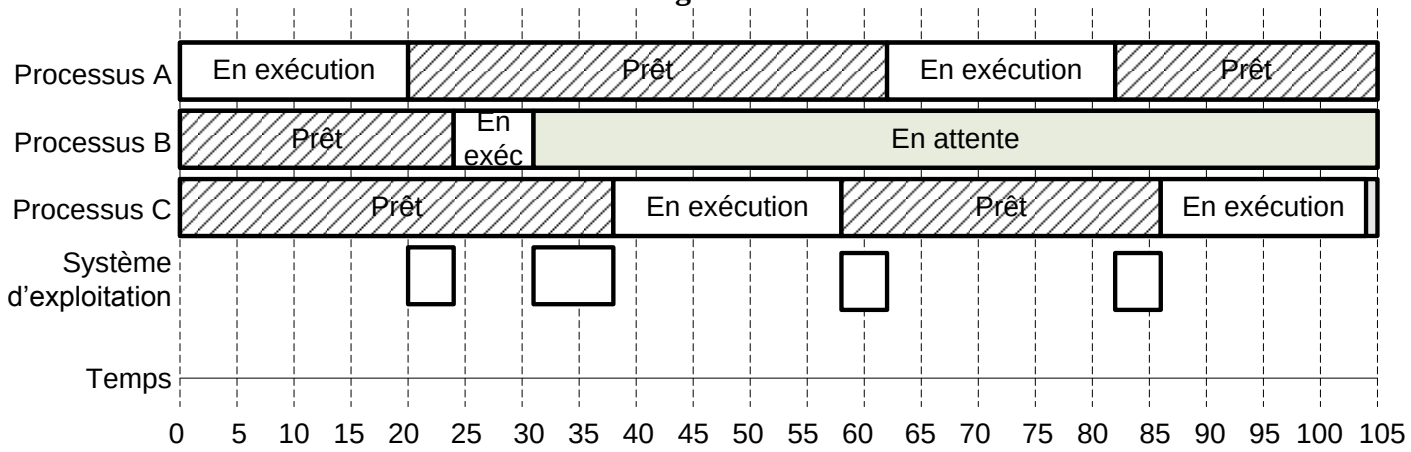


Figure 5



Déterminer le pourcentage de temps que chaque processus et le système d'exploitation exécute avec l'UCT. Est-ce que ces valeurs sont raisonnables?

Temps total d'exécution est 105 unités de temps.
Processus A roule pour 20+20 = 40 unités de temps, et donc 40/104 = 38% du temps
Processus B roule pour 7 unités de temps, et donc 7/104 = 6.7 % du temps

Processus C roule pour 20 + 18 = 38/104 unités de temps, et donc = 36% du temps (dans les faits ceci devrait être plus haut car le processus n'as pas complété son temps alloué avec l'UCT)
Le système d'exploitation roule pour 4+7+4+4 = 19 unités de temps, et donc 18% du temps.

Un système d'exploitation qui roule pour 18% du temps n'est pas bien conçu. On devrait espérer de garder la partie d'exécution du SE en bas de 10%. Notez que dans un vrai système, un processus (et SE) exécute beaucoup plus d'instructions durant une tranche de temps que dans cette exemple.

4. Quand un processus crée un nouveau processus avec fork(), laquelle ou lesquelles des ressources suivantes sont partagée(s) entre les processus parent et enfant.
    - **a.** Pile
    - **b.** Tas
    - **c.** Mémoire partagée

Seulement la mémoire partagée sera commune entre les deux processus. La pile et le tas seront copiés dans le nouveau processus.

5. Que sera la sortie générée à l'écran par la LIGNE A du programme ci-dessous? Expliquez votre réponse.

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int valeur=5;

int main()
{
   pid_t pid;

   pid = fork();
   if(pid == 0) /* processus enfant */
   {
      valeur = valeur + 15;
   }
   else if(pid > 0)  /* processus parent */
   {
      wait(NULL);
      printf("PARENT: valeur = %d\n", valeur); /* LIGNE A */
      exit(0);
   }

}
```

La sortie est:

PARENT: valeur = 5

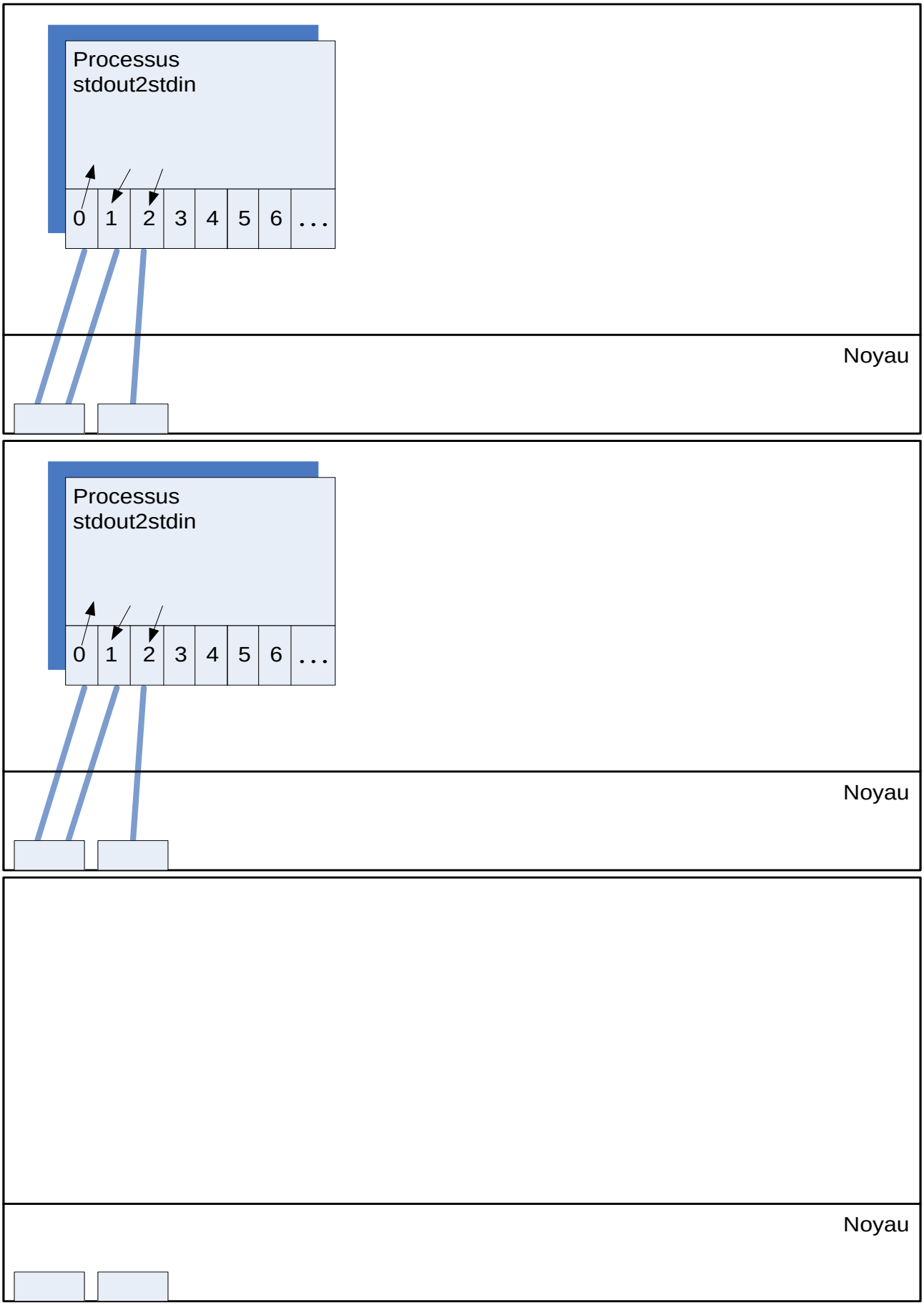
L'exécution de l'enfant n'affecte pas la valeur de la variable « valeur » dans le processus parent puisqu'il aura sa propre copie. À partir du point où fork est exécuté, les deux processus roulent le programme ci-dessus avec leur propre copie du programme (et donc ont chacun leur propre copie de la variable). Puisque fork retourne le PID de l'enfant dans le processus parent, la partie « else » de l'instruction « if » est exécuté dans le parent. Dans l'enfant, fork retourne 0, et donc la partie « if » de l'instruction « if » est exécutée, mais la valeur de la variable « valeur » n'est modifiée que dans l'enfant.

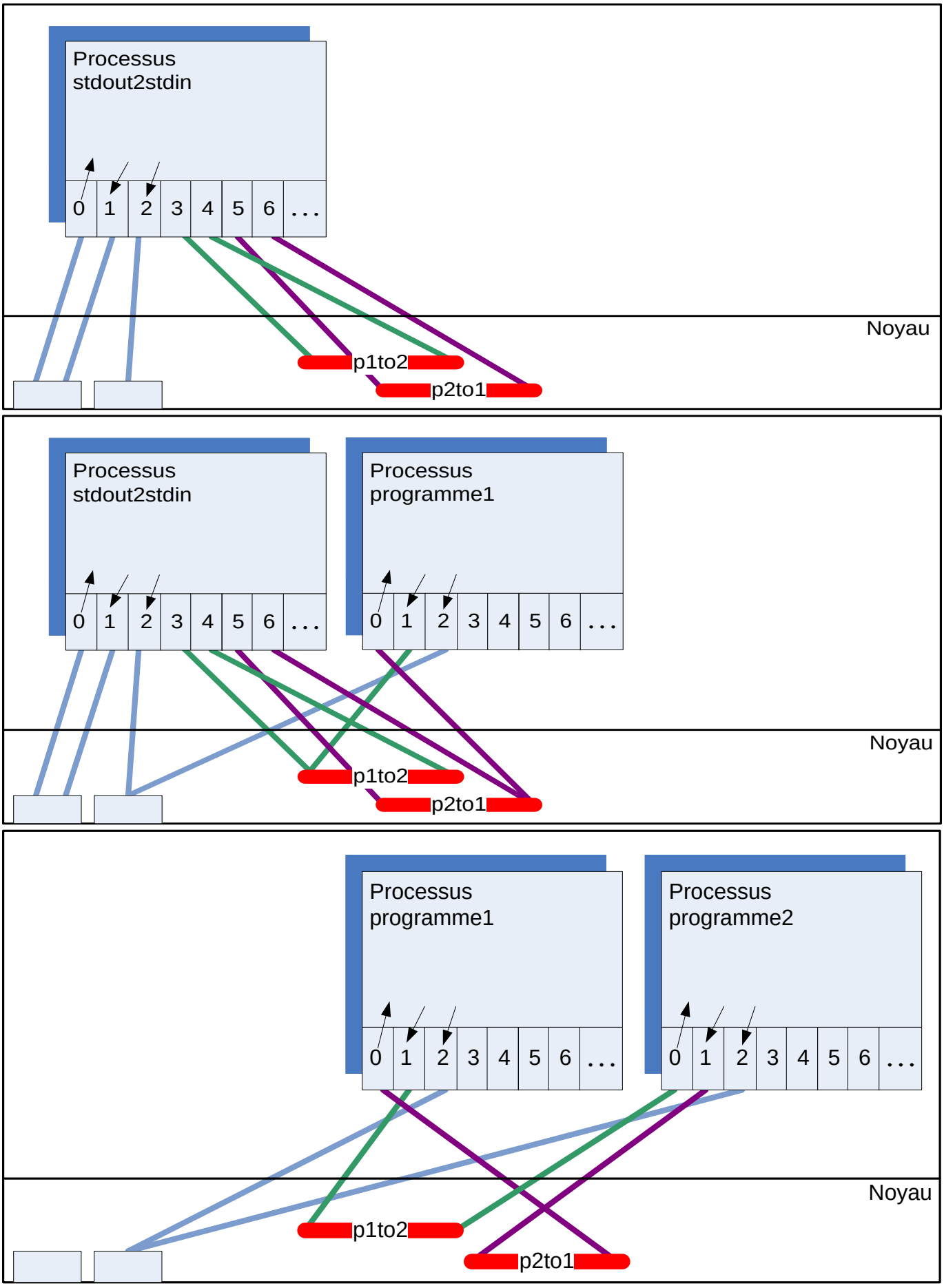
6. Examinez le code C suivant du program stdout2stdin.

```
int main(int argc, char *argv[])
{

   char *pgrm1;         /* pointeur au premier programme */
   char *pgrm2;         /* pointeur au deuxième programme*/
   int p1to2fd[2];  /* tuyau de prc1 au prc2 */
   int p2to1fd[2];  /* tuyau de prc2 au prc1 */
   int pid;

   if(argc != 3)
   {
     printf("Usage: stdout2stdin <pgrm1> <pgrm2> \n");
     exit(1);
   }

   /* cherchons les noms de programmes */
   pgrm1 = argv[1];
   pgrm2 = argv[2];

   /* création des tuyaux */
   pipe(p1to2fd);
   pipe(p2to1fd);
      /* Compléter la première figure lorsque le programme arrive à ce
         point*/
   /* création du processus 1 */
   pid = fork();
   if(pid == 0)
   {
      dup2(p1to2fd[1], 1);
      dup2(p2to1fd[0], 0);
      close(p1to2fd[0]);
      close(p1to2fd[1]);
      close(p2to1fd[0]);
      close(p2to1fd[1]);
      execlp(pgrm1, pgrm1, NULL);
   }
      /* Compléter le deuxième figure lorsque le programme arrive à ce
         point*/

   /* création du processus 2 */
   pid = fork();
   if(pid == 0)
   {
      dup2(p2to1fd[1], 1);
      dup2(p1to2fd[0], 0);
      close(p1to2fd[0]);
      close(p1to2fd[1]);
      close(p2to1fd[0]);
      close(p2to1fd[1]);
      execlp(pgrm2, pgrm2, NULL);
   }
}
      /* Compléter la troisième figure lorsque le programme aura terminé */
```

Complétez les figures de la prochaine page pour montrer comment les tuyaux et processus sont créés et branches entre eux lorsque la commande suivante est exécutée:

```
stdout2stdin programme1 programme2
```


Processus
stdout2stdin
0
1
2
3
4
5
6
...
Noyau
Processus
stdout2stdin
0
1
2
3
4
5
6
...
Noyau
Noyau


Processus
stdout2stdin
0 1 2 3 4 5 6 ...
Noyau
p1to2
p2to1
Processus
stdout2stdin
Processus
programme1
0 1 2 3 4 5 6 ...
0 1 2 3 4 5 6 ...
Noyau
p1to2
p2to1
Processus
programme1
Processus
programme2
0 1 2 3 4 5 6 ...
0 1 2 3 4 5 6 ...
Noyau
p1to2
p2to1