Interactions entre système multi-tâche et processus

Michel Simatic

module CSC4508/M2

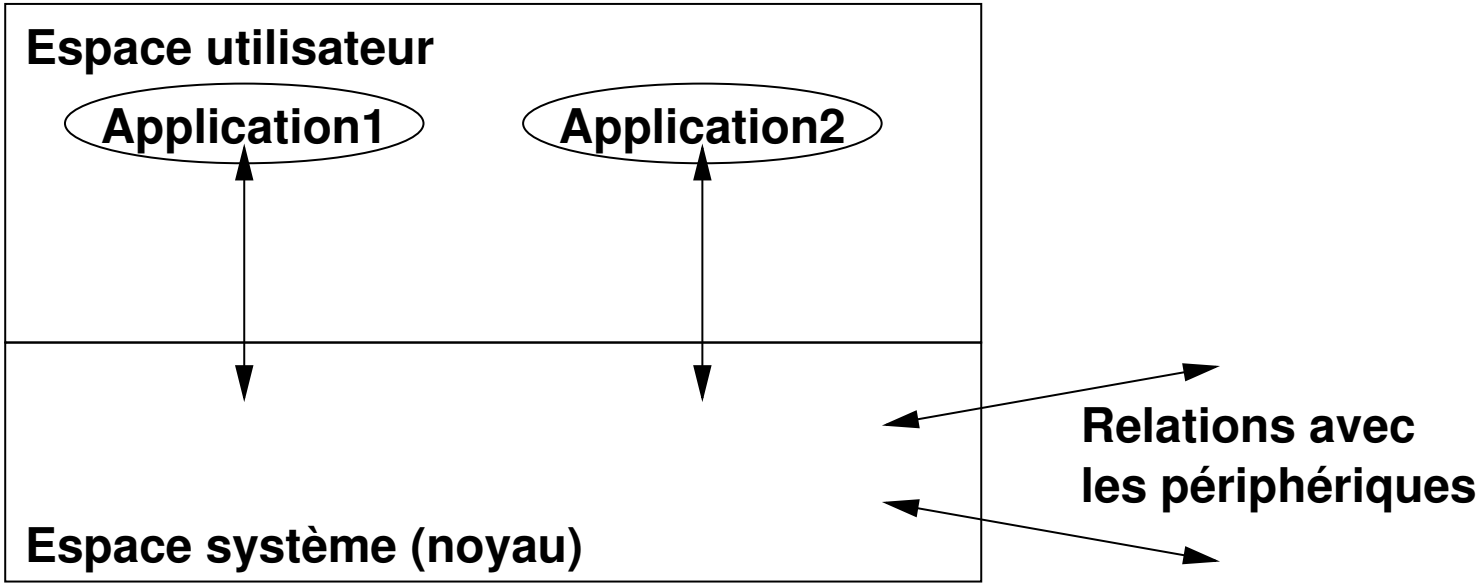
Avril 2018

Plan du document

1 Point de vue processus

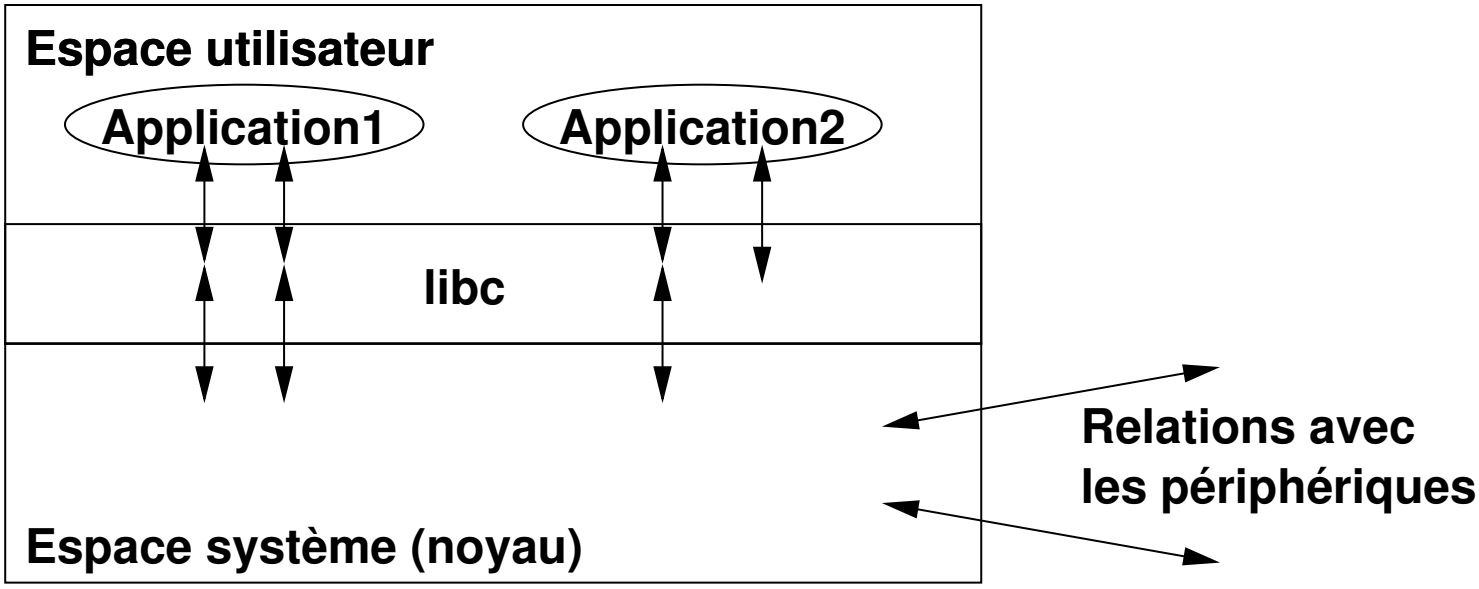
1.1 Les applications et le système

- Les applications en cours et le système résident en mémoire centrale
- La mémoire est divisée en deux parties
  - L'espace système : le noyau
  - L'espace utilisateur : où résident les applications



1.2 La programmation système

- C'est le développement d'applications en utilisant les ressources et les outils fournis par le système
  - Utilisation de fonctions standards fournies avec le langage : incluses dans la `libc`, bibliothèque standard du langage C pour Unix
  - Ces fonctions dialoguent avec le noyau et contrôlent ce dialogue
  - Les applications utilisent ainsi des ressources du noyau



1.3 Les fonctions de la libc

- Deux types fondamentaux :
  1. Les appels système
     - Ce sont les fonctions permettant la communication avec le noyau
     - Exemples : `open`, `read`, `write`, `fcntl`...
  2. Les fonctions
     - Ce sont les fonctions standard du langage C
     - Exemples : `printf`, `fopen`, `fread`, `strcmp`...

1.4 Utilisation des appels systèmes

- Travaillent en relation directe avec le noyau
- Retournent un entier positif ou nul en cas de succès et -1 en cas d'échec
- Par défaut le noyau peut bloquer les appels systèmes et ainsi bloquer l'application si la fonctionnalité demandée ne peut pas être servie immédiatement
- Ne réservent pas de la mémoire dans le noyau. Les résultats sont obligatoirement stockés dans l'espace du processus (dans l'espace utilisateur), il faut prévoir cet espace par allocation de variable (statique, pile) ou de mémoire (`malloc()...`)

1.5 Utilisation des fonctions

- Retournent une valeur de type divers (entier, caractère, pointeur...). Voir le manuel de référence pour chacune d'entre elles
- Lorsqu'elles rendent un pointeur, celui-ci est le pointeur NULL en cas d'échec
- Certaines peuvent utiliser un appel système (`fopen` s'appuie sur `open`, fread sur `read`...)
- Les fonctions rendant un pointeur ont généralement alloué de la mémoire dans l'espace du processus et le pointeur rendu y donne accès

1.6 Test du retour des appels système et des fonctions

- Il faut *toujours* tester la valeur de retour d'un appel système

  Si valeur rendue est égale à -1

  - Il faut gérer le problème
  - Une variable externe de nom `errno` est positionnée à une valeur indiquant l'erreur (cette variable vaut 0 s'il n'y a pas d'erreur)

- Il faut *presque toujours* tester la valeur de retour d'une fonction

  Pour les fonctions rendant un pointeur, si la valeur rendue est NULL

  - Il faut gérer le problème

- Envoi de messages d'erreurs à l'aide des fonctions
  - `perror()` (ou `strerror()`)
  - `fprintf()` (ou `fputs`)

1.7 Que faire en cas d'erreur système ?

- Utiliser `perror("message")` pour afficher le message indiqué suivi de « : » et du message système correspondant à l'erreur
- Utiliser la macro `assert()`

```
rc = appelSysteme(...);
assert(rc >= 0);
```

- (*Spécifique BSD et non standard, mais OK Linux et MacOS*) Utiliser les fonctions `err` et `errx`
- (*Spécifique compilateur gcc*) Utiliser les fonctions `error` et `error_at_line`
- Définir et utiliser une macro `ERROR_AT_LINE`

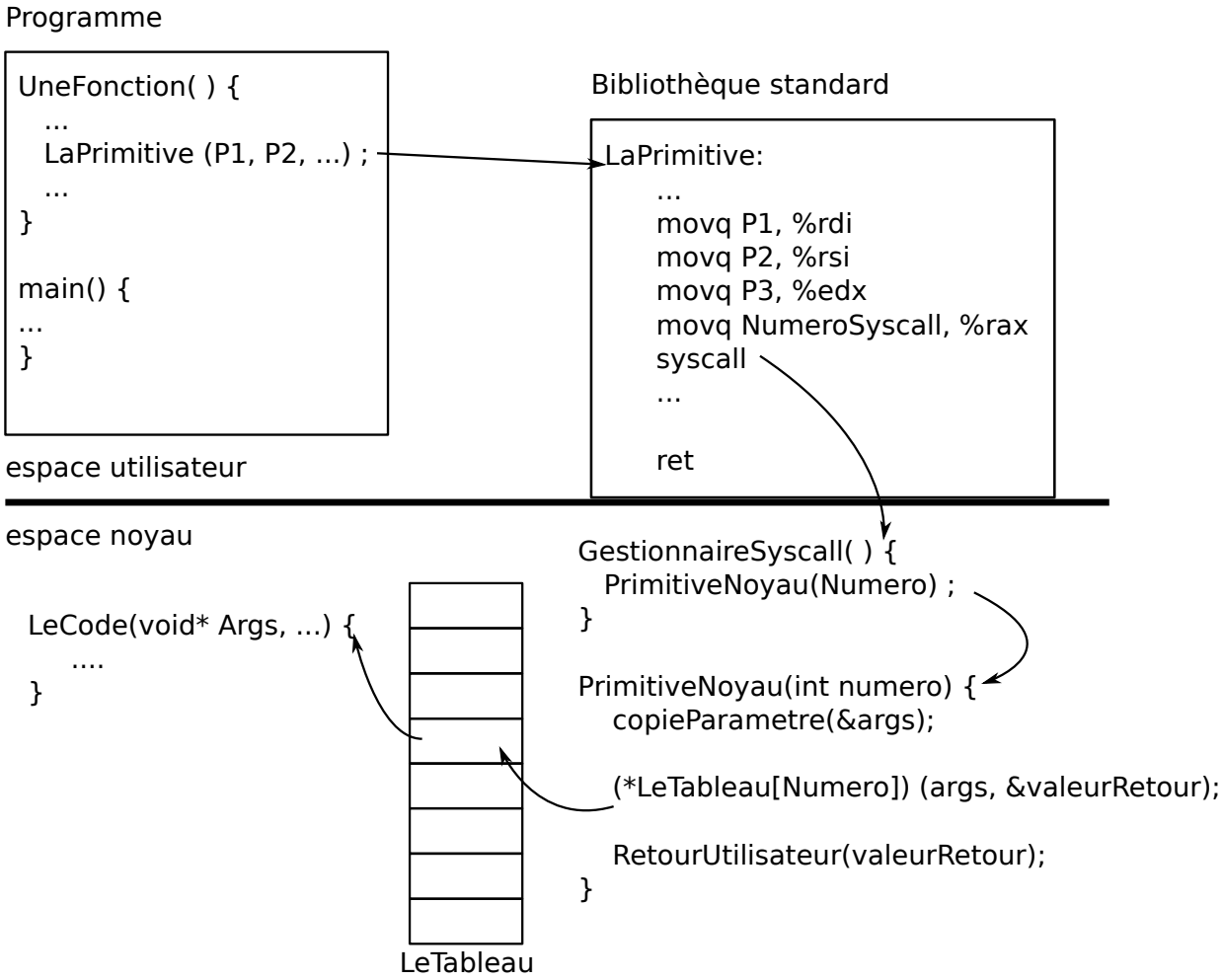
```
#define ERROR_AT_LINE(status,errnum,filename,linenum,...) { \
   fprintf(stderr,"%s:%d:", filename,linenum);  \
   fprintf(stderr,__VA_ARGS__);  \
   fprintf(stderr,":%s\n", strerror(errnum));\
   abort();                                  \
}
```

2 Point de vue système

2.1 Généralités sur l'exécution des tâches système

- L'exécution des tâches système s'effectue en général sur le compte des différents processus hébergés par le système
- Le système ne se déroule pour son propre compte que dans très peu de cas
- On distingue deux types d'actions
  - Le traitement des appels système
  - La prise en compte des interruptions

2.2 Traitement des appels système



2.3 Prise en compte des interruptions

- Les interruptions sont acquittées par le processus en cours
- Il peut être amené à passer du mode utilisateur au mode noyau
- Il fait passer les processus qui attendaient l'événement lié à cette interruption dans le mode prêt pour exécution

3 Ordonnancement des processus sous Linux

3.1 Priorité statique

- `int sched_setscheduler(pid_t pid, int policy, const struct sched_param *p);`

  permet d'affecter à un processus une *priorité statique* (entre 0 et 99) stipulant la file d'attente de l'ordonnanceur (*scheduler*) où il doit être placé.

- Pour décider du processus à exécuter, l'ordonnanceur prend le premier processus prêt dans la file de plus haute priorité

- NB : Valeurs fortes de *priorité statique* = Fortes priorités

3.2 Politique d'ordonnancement

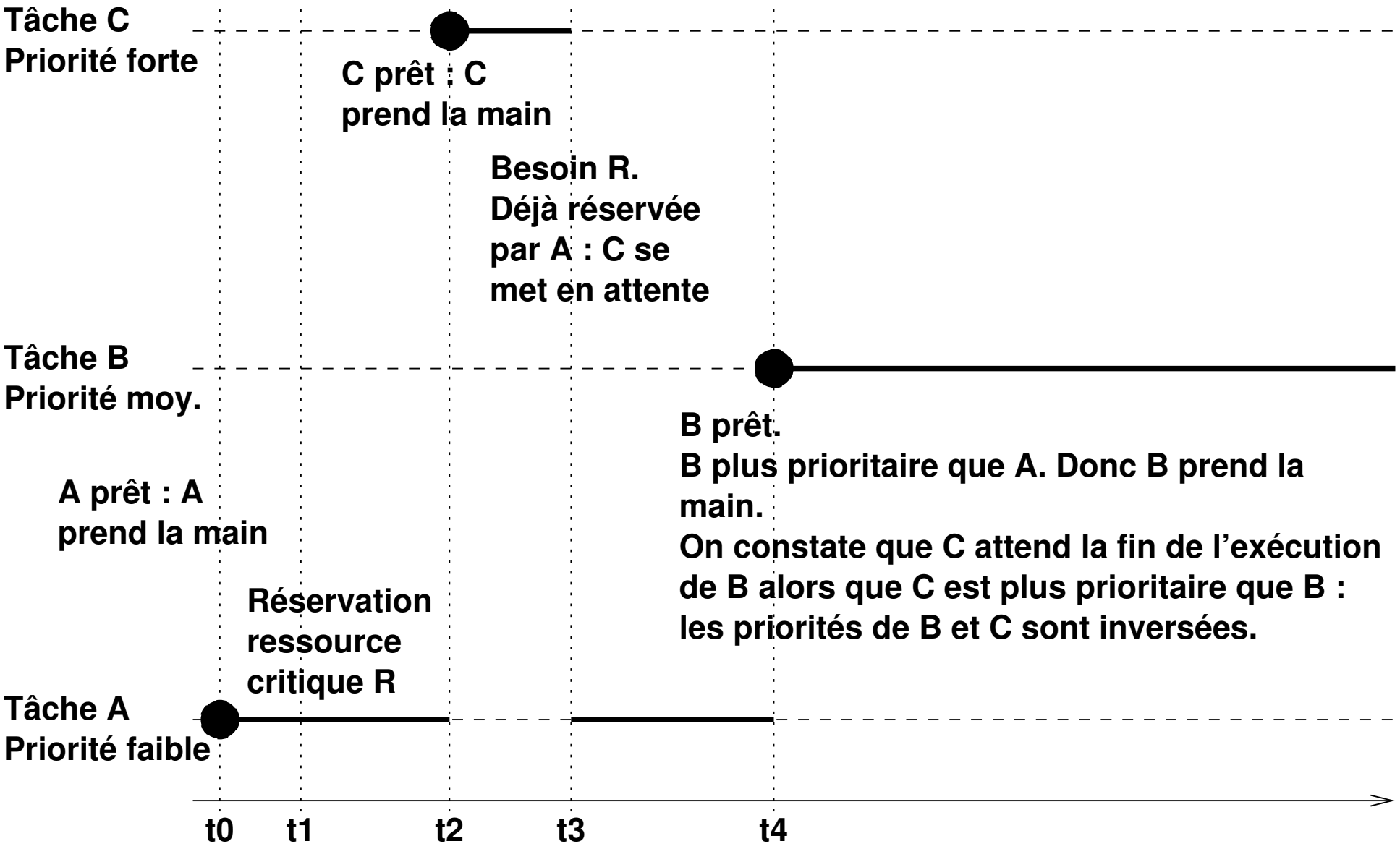
- `int sched_setscheduler(pid_t pid, int policy, const struct sched_param *p);`

  permet d'affecter à un processus une *politique d'ordonnancement*

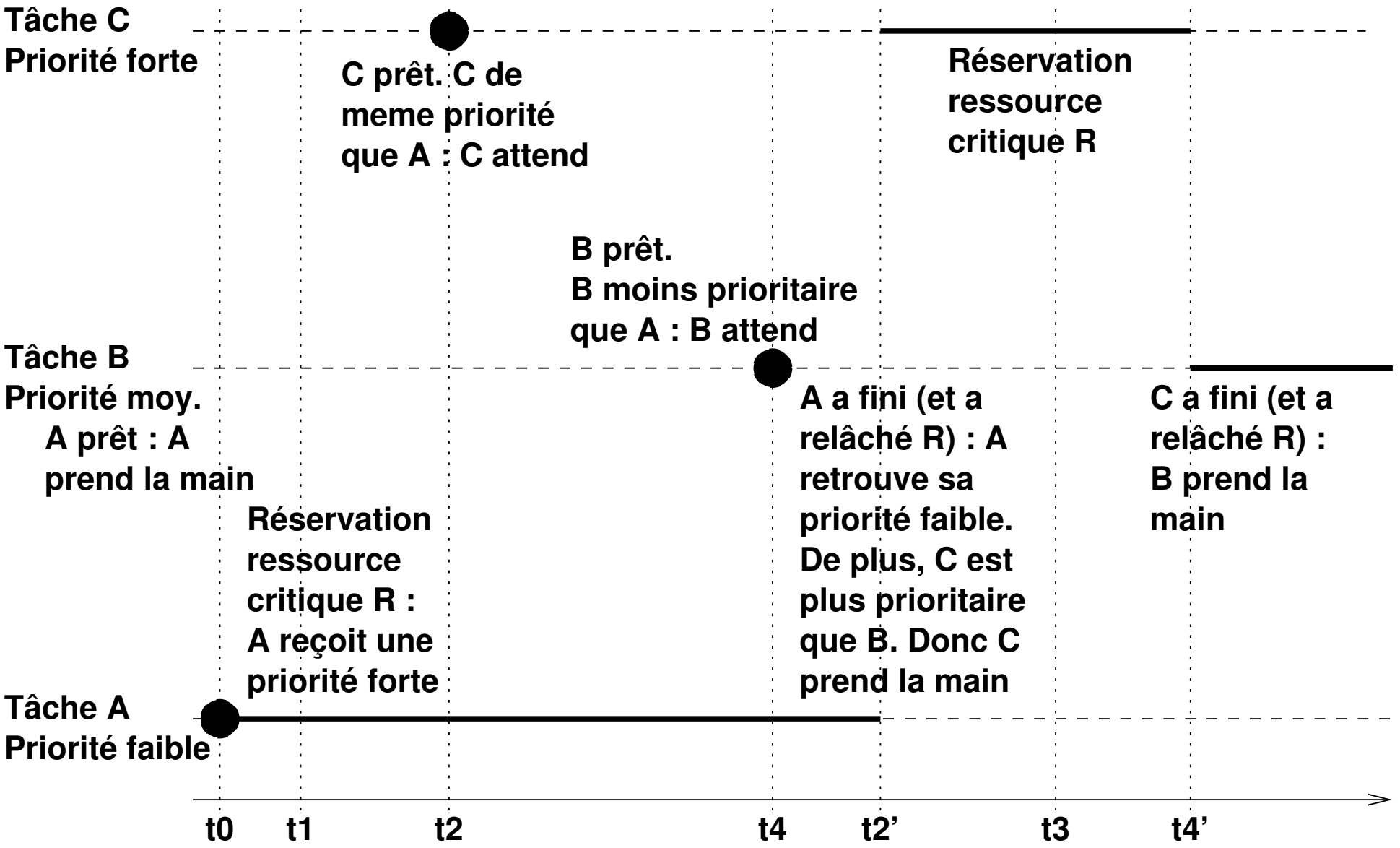
  - `SCHED_FIFO` : Le processus n'arrête son exécution que si
    - Il fait une E/S
    - Il fait un appel à `int sched_yield(void)` (il laisse alors passer les autres processus prêts de même priorité)
    - Il change sa *politique d'ordonnancement*
    - Il est préempté par des processus plus prioritaires (il reprend ensuite son exécution)
  - `SCHED_RR` : une condition d'arrêt supplémentaire est définie
    - La durée d'exécution atteint un quantum de temps (déterminable via la fonction `int sched_rr_get_interval(pid_t pid, struct timespec *tp)`)
  - `SCHED_OTHER` : ordonnancement en temps partagé (réservé aux processus de *priorité statique* 0)

3.3 Ordonnancement temps-réel Linux

3.3.1 Inversion de priorité : énoncé du problème



3.3.2 Inversion de priorité : exemple de solution



3.4 Ordonnancement standard

- La politique `SCHED_OTHER` est la politique utilisée par défaut par Linux
- Elle s'appuie sur l'ordonnanceur `CFS` (*Completely Fair Scheduler*)
- Le processus à exécuter est choisi dans la liste des processus de *priorité statique* nulle, en utilisant une *priorité dynamique* qui ne s'applique que dans cette liste.
  NB : Valeurs fortes = Faibles priorités
- La *priorité dynamique* est fonction, entre autres
  - d'une *priorité d'ordonnancement* (caractère « gentil » du processus fixé avec les appels système `int nice(int inc)` ou `int setpriority(int which, int who, int prio)`)
  - du fait que le processus ait relâché le processeur avant expiration de son délai
- Le processus le plus prioritaire (dynamiquement) est exécuté jusqu'à ce qu'il
  - Fasse une E/S
  - Devienne moins prioritaire (dynamiquement) qu'un autre processus

Bibliographie du chapitre