Programmation des systèmes

L3 Informatique – Université de Lille
Processus – Travaux dirigés

L'équipe pédagogique

Introduction

Ce document est le support de travaux dirigés pour le cours de Programmation de Système, au 3ème année de Licence Informatique à l'Université de Lille. Le document a été crée par Philippe Marquet : il a évolué au cours des années avec des contributions par toute les équipes pédagogiques successives [1].

Table des matières

1. Philippe Marquet, Gilles Grimaud, Samuel Hym, Giuseppe Lipari, et beaucoup d'autres.

1 Clonage de processus

1.1 Compréhension du clonage simple

Observez le programme suivant.

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

int main() {
    pid_t pid;

    printf("A: ce message s'affiche combien de fois ?\n");
    fflush(stdout);

    pid = fork();
    if (pid == 0) {
        /* qui suis-je, le pere ou le fils ? */
        printf("je suis le ...\n");
    } else {
        /* qui suis-je, le pere ou le fils ? */
        printf("je suis le ...\n");
    }

    printf("B: ce message s'affiche combien de fois ?\n");
    fflush(stdout);

    exit(EXIT_SUCCESS);
}
```

Question 1 *Répondre au questions* Répondez aux questions posées dans le code. ■

Solution : Le message A apparaît une fois. B deux fois. ■

Question 2 Pouvez-vous déterminer l'ordre d'apparition des messages ? Si oui donnez-le, sinon pourquoi ? Pouvez-vous alors déterminer un ordre partiel d'apparition des messages ? ■

Solution : Un ordre partiel est possible : A doit apparaître avant les 2 B, et "je suis le ..." toujours après le A et chacun avant un des B. ■

Question 3 On ajoute dans le `else`, avant le `printf` l'instruction `wait(NULL)`. Cela change-t-il quelque chose ? ■

Solution : Maintenant, l'ordre est totale : A, "je suis le fils", B, "je suis le père", B. ■

Question 4 Pour garantir un fonctionnement plus sûr de ce code, que manque-t-il ? ■

Solution : Vérifier si `fork` a échoué. ■

Question 5 À quoi les instructions `fflush(stdout)` servent-elles ? ■

Solution : Si on ne met pas le `fflush` alors le message *A* apparaîtra plusieurs fois si on redirige vers un fichier... `\n` nest efficace que si la sortie standard est le terminal. ■

1.2 Généalogie de processus

On exécute le programme suivant.

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

int main() {
    int i;
    pid_t pid;

    for (i = 0; i < 3; i++) {
        pid = fork();
        if (pid == -1) {        /* erreur */
            perror("erreur fork");
            exit(EXIT_FAILURE);
        } else if (pid == 0) {  /* fils */
            fprintf(stderr, "fils : %d mon pid : %d mon père : %d\n", i, getpid(),
    getppid());
            exit(0);
        } else {                /* pere */
            fprintf(stderr, "pere : %d mon pid : %d mon père : %d\n", i, getpid(),
    getppid());
        }
    }
    exit(EXIT_SUCCESS);
}
```

Question 6 Déroulez son exécution et indiquez ce qui est affiché. ■

Question 7 Combien de processus sont lancés en tout (il pourra être utile de représenter sous forme arborescente les processus) ? ■

Question 8 Quels appels systèmes peut-on utiliser pour changer l'affichage et permettre un meilleur suivi des processus (savoir qui est fils de qui) ? ■

Solution : Il s'agit bien entendu de suivre le fils jusqu'au bout de son exécution et de ne pas croire (mais pourquoi donc) qu'il s'arrête après le premier affichage.

On note aussi que chaque processus « possède » un `i` propre ; la mémoire du père est dupliquée au moment du `fork()`.

On obtient donc :

— un processus père va afficher 0 et créer un fils qui va afficher 0 ;
— chacun de ces deux processus continue : ils ont chacun un `i` qui est incrémenté à 1 et ils vont chacun créer un fils. Chacun de ces quatre processus affiche 1 ;
— chacun de ces quatre processus ... huit processus affichent 2.
— chacun de ces huit processus sort de la boucle et termine.

L'ordonnancement des processus ne pouvant être prédit, de nombreuses (combien ?) permutations sont possibles.

On obtient un « arbre » d'exécution comme celui-ci , chacune des colonne figurant un des huit processus :

```
pere : 0
     +--fils : 0
        pere : 1
             +--fils : 1
                pere : 2
                     +--fils : 2
        pere : 2
             +-----------------fils : 2
pere : 1
     +---------------------------------fils : 1
                                      pere : 2
                                           +--fils : 2
pere : 2
     +-------------------------------------------------fils : 2
```

On peut comprendre ce qui se passe en préfixant les affichages par le PID du processus et celui de son père. Pour éviter les orphelins (PPID à 1) j'ajoute un `wait()`, que les pères attendent la terminaison de leur fils au plus tôt. Il n'est as utile de traiter e tous ces détails dès maintenant avec les étudiants, mais si certains d'entre-eux en venaient à taper ce programme en TP, on pourra le leur en toucher un mot... :

```c
int main() {
    int i;
    pid_t pid;

    for (i = 0; i < 3; i++) {
        pid = fork();
        if (pid == -1) {        /* erreur */
            perror("erreur fork");
            exit(EXIT_FAILURE);
        } else if (pid == 0) {  /* fils */
            fprintf(stderr, "[%d/%d] fils : %d\n", getpid(), getppid(), i);
        } else {                /* pere */
            fprintf(stderr, "[%d/%d] pere : %d\n", getpid(), getppid(), i);
            (void) wait((int *) 0);
        }
    }
    exit(EXIT_SUCCESS);
}
```

■

1.3 Quatre fils

Question 9 Donnez un programme qui affiche les entiers de 0 à 3 par 4 processus différents. L'exécution de ce programme se termine après l'affichage des 4 entiers. ■

Solution :

- On reprend en gros le schéma de création de processus de l'exercice précédent, mais les fils terminent après avoir affiché la valeur de `i`;
- afficher son numéro de pid : identifier qui affiche quoi;
- le père attend la terminaison de ses fils, que le programme ne termine pas avant que tous les fils aient terminés;
- on notera l'utilisation du case, préférée aux `if` successifs.

```c
#define NFILS 4

int main() {
    int i;

    for (i = 0; i < NFILS; i++) {
        switch (fork()) {
        case -1:                /* erreur */
            fprintf(stderr, "erreur fork\n");
            exit(EXIT_FAILURE);
        case 0:                 /* fils */
            fprintf(stderr, "[%4d] fils : %d\n", getpid(), i);
```

```
            exit(EXIT_SUCCESS);
        default:                /* pere */
            /* rien a faire , s'en va créer le fils suivant */
            ;
        }
    }

    for (i = 0; i < NFILS; i++)
        (void) wait((int *) 0);

    exit(EXIT_SUCCESS);
}
```

■

Question 10 Assurez que les processus fils affichent les entiers dans l'ordre croissant. ■

Solution :

— Il suffit de ne pas créer le second fils avant que le premier ait terminé, le troisième avant que le second ait terminé...

— On remplace donc la boucle finale de `wait()` du processus père par un `wait()` après chaque création d'un fils.

```
    default: /* pere */
    /* attend que le fils termine pour créer le fils suivant */
    (void) wait((int *)0);
```

■

1.4 Multi-fourche

Question 11 Écrivez une fonction

```
    void trif (void(*f1)(void), void(*f2)(void), void(*f3)(void));
```

Le processus exécutant `trif(f1, f2, f3)` engendre des processus exécutant respectivement les fonctions `f1()`, `f2()` et `f3()`, et attend la fin des processus engendrés pour terminer la fonction.

Cette fonction est par exemple utilisée dans le contexte suivant :

```
    static void f(int seconds, const char *fname) {
      sleep(seconds);
      fprintf(stderr, "Fonction %s() exécutée par le processus "
      "de pid %d\n", fname, getpid());
```

```
}

static void fa() { f(4, "fa"); }
static void fb() { f(2, "fb"); }
static void fc() { f(3, "fc"); }

int main() {
  trif(fa, fb, fc);
  fprintf(stderr, "terminaison de main()\n");

  exit(EXIT_SUCCESS);
}
```

■

Solution : On imbrique des `fork()`, sans oublier les `exit()` pour les fils et des `wait()` (de préférence finaux) pour le père :

```
static void
trif(void(*f1)(void), void(*f2)(void), void(*f3)(void))
{
    switch (fork()) {
        case -1 : /* erreur */
            perror("fork");
            exit(EXIT_FAILURE);
        case 0 : /* fils */
            f1();
            exit(EXIT_SUCCESS);
        default : /* pere */
            switch (fork()) {
                case -1 : /* erreur */
                    perror("fork 2");
                    exit(EXIT_FAILURE);
                case 0 : /* fils */
                    f2();
                    exit(EXIT_SUCCESS);
                default : /* pere */
                    switch(fork()) {
                        case -1 : /* erreur */
                            perror("fork 3");
                            exit(EXIT_FAILURE);
                        case 0 : /* fils */
                            f3();
                            exit(EXIT_SUCCESS);
                        default : /* pere */
                            ;
                    }
            }
    }
```

```
    /* attente des fils */
    wait((int *)0); wait((int *)0); wait((int *)0);
    fprintf(stderr, "terminaison de trif()\n");
}
```

■

Question 12 La fonction `multif()` généralise la fonction `trif()` précédente.

```
typedef int (*func_t) (char *);
int multif (func_t f[], char *args[], int n);
```

Le type `func_t` est défini comme « pointeur sur une fonction à un paramètre chaîne de caractères et retournant un entier ».

Les arguments de `multif()` sont un tableau de telles fonctions, un tableau des arguments à passer à ces fonctions, et la taille `n` de ces tableaux. Chacune des fonctions est exécutée par un processus différent. La fonction `f[i]()` sera appelée avec `args[i]` comme argument. Ce processus se termine en retournant comme statut la valeur de la fonction.

La fonction `multif()` se termine elle-même en retournant la conjonction des valeurs retournées par les processus fils : elle ne retourne `EXIT_SUCCESS` que si chacun des processus fils a retourné `EXIT_SUCCESS`.

Pour tester en pratique cette `multif()`, on définira une fonction `testfunc` telle que :

— si son argument est la chaîne `"true"`, elle retourne `EXIT_SUCCESS`,

— si son argument est la chaîne `"false"`, elle retourne `EXIT_FAILURE`.

On définira aussi dautres comportements sur dautres valeurs de largument (fonction qui prend du temps pour se terminer, par exemple en utilisant `sleep`, pour vérifier que les fonctions ne sont pas exécutées les unes après les autres mais de façon concurrente, etc.).

Enfin la *commande* `multif` prendra en argument les valeurs des arguments pour `testfunc` et se terminera avec la valeur de retour de `multif()` (`./multif true false` devra donc déclencher `multif()` avec `n = 2`, le tableau `f` contiendra 2 fois `testfunc`, le tableau `args` contiendra les chaînes `"true"` et `"false"` et se terminera en retournant `EXIT_FAILURE`). ■

2 Tubes

2.1 Calcul parallèle

Pour renvoyer des données vers le processus père, on peut utiliser des tubes anonymes.

Question 1 Écrire un programme multi-processus dans lequel un processus père crée `N` processus fils. Li-ème processus fils calcule tous les nombres premiers entre iM et $(i+1)M$, (ou M est le nombre d'éléments traités par un processus) et les renvoie sur un tube, puis se termine. Le processus père reçoit toutes les données sur le tube et les affiches à l'écran. ■

Solution : Voici la solution, très simple. Attention à bien fermer les descripteurs non utilisés, si non le père reste bloqué sur la read.

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
#include <math.h>
#include <assert.h>
#include <sys/types.h>
#include <sys/wait.h>

#define N 5
#define HOWMANY 100

int is_prime(int i)
{
    for (int k=2; k <= (int)sqrt((double)i); k++)
        if ((i % k) == 0) return 0;

    return 1;
}

int main()
{
    int pd[2];
    assert(pipe(pd) == 0);
    for (int i=0; i<N; i++) {
        int ch = fork();
        assert(ch >= 0);
        if (ch == 0) {
            close(pd[0]);
            for (int k=i*HOWMANY;k<(i+1)*HOWMANY; k++)
                if (is_prime(k)) write(pd[1], &k, sizeof(int));
            exit(EXIT_SUCCESS);
        }
    }
```

```
35      close(pd[1]);
36
37      int number;
38      while(read(pd[0], &number, sizeof(int)))
39          printf("%d\n", number);
40
41      for (int i=0; i<N; i++) {
42          wait(NULL);
43      }
44      close(pd[0]);
45  }
```

■

Maintenant, on veut implanter une communication bidirectionale. Chaque processus fils calcule les nombre premiers dans un intervalle `start, stop` defini par le père. Chaque fils donc fait la chose suivante :

— Il lit les deux nombres `start` et `stop` sur un tube de communication père-fils ;
— Il calcule tous les premiers entre `start` et `stop` et renvoie les resultats sur un deuxième tube de communication fils-père ;
— Il attends la prochaine couple `start-stop`.

Les fils terminent quand le père ferme le descripteur d'écriture du tube de communication père-fils.

Le père prends en argument le nombre maximale `Max` d'entiers à calculer. Il crée les tubes père-fils et fils-père. Il decoupe l'intervalle $[0, Max - 1]$ en morceaux de taille M. Il envoie toutes les intervalles sur le tube père-fils, et il attends les résultats sur le tube fils-père.

Question 2 Implanter le programme. ■

Solution : L'interêt de cet exercice est de leur montrer l'occurrence d'un deadlock.

Voici une solution qui marche pour des arguments jusqu'à 900.000 :

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <unistd.h>
4   #include <signal.h>
5   #include <math.h>
6   #include <assert.h>
7   #include <sys/types.h>
8   #include <sys/wait.h>
9
10  #define N 5
11  #define M 100
12
13  struct pair {
14      unsigned int start;
15      unsigned int stop;
16  };
17
```

```c
int is_prime(int i)
{
    for (int k=2; k <= (int)sqrt((double)i); k++)
        if ((i % k) == 0) return 0;

    return 1;
}

int main(int argc, char *argv[])
{
    if (argc<2) {
        printf("Usage : %s num\n", argv[0]);
        exit(EXIT_FAILURE);
    }
    unsigned max_num = atoi(argv[1]);

    int pf[2];
    int fp[2];

    assert(pipe(pf) == 0);
    assert(pipe(fp) == 0);

    for (int i=0; i<N; i++) {
        int ch = fork();
        assert(ch >= 0);
        if (ch == 0) {
            close(pf[1]);
            close(fp[0]);
            struct pair st;
            while(read(pf[0], &st, sizeof(struct pair)) > 0) {
                // printf("Fils %d calcule l'interval [%u, %u]\n", getpid(), st.start, st.stop);
                for (int k=st.start;k<st.stop; k++)
                    if (is_prime(k)) write(fp[1], &k, sizeof(int));
            }
            printf("Fils %d terminé\n", getpid());
            exit(EXIT_SUCCESS);
        }
    }
    close(fp[1]);
    close(pf[0]);

    for (int i=2; i<max_num; i+=M) {
        struct pair st;
        st.start = i;
        st.stop = (i+M) < max_num ? (i+M) : max_num;
        write(pf[1], &st, sizeof(struct pair));
    }

    close(pf[1]);
```

```
67
68      int number;
69      while(read(fp[0], &number, sizeof(int)))
70          printf("%d\n", number);
71
72      for (int i=0; i<N; i++) {
73          wait(NULL);
74      }
75      close(fp[0]);
76  }
```

Le père crée deux tubes, `pf` (père-fils) et `fp` (fils-père). Après il écrit tous les intervalles sur le descripteur `pf[1]`, ferme ce descripteur, et il se met en attente des résultats sur le tube `fp[0]`.

Il faut remarquer que chaque intervalle doit être écrit par une seule `write()`. C'est pourquoi j'ai déclaré une structure de deux éléments. Si vous écrivez les deux nombres séparément et les lisez séparément, il y a les risque de lire des intervalles incorrectes. Une alternative est d'utiliser un tableau de deux entiers.

Il faut faire attention à la place où vous fermez le descripteur `pf[1]` ! Après avoir écrit tous les intervalles et avant de faire les lectures. Si vous le faites après, il y a un **deadlock**, les fils ne terminent jamais et le père non plus.

Dans le cas où l'argument est un nombre trop grand, il peut y avoir trop d'intervalles à écrire sur le tube `pf` ; le tube `fp` est vite rempli des solutions, et les fils sont bloqués sur la `write()` ; pendant que le tube `pf` est rempli d'intervalles et le père reste bloqué lui aussi sur la `write()`. Encore une fois on a un deadlock ! Avec 1.000.000 il y un deadlock sur ma machine.

Pour éviter ce problème, on peut créer un fils supplémentaire qui s'occupe d'écrire les intervalles sur le tube `pf` pendant que le père vide le tube `fp`. Vous pouvez, si vous pensez que c'est utile, montrer ce qui arrive dans un tel cas avec un schéma au tableau.

Il faut remarquer que le deadlock n'est pas un problème seulement de sémaphores : il peut arriver même sur des synchronisation sur des tubes ! Et avec des signaux ... ■

2.2 Attendre le lancement d'une commande

On veut écrire un outil `pps <str>` (periodic ps) qui tous les deux seconds cherche à savoir si une commande contenant le mot `str` est en exécution. Pour faire ça, tous les deux seconds :

1. on lance la commande `ps axu` dans un processus fils ;
2. on lit la sortie standard du processus : si on trouve la chaîne contenue dans le premier argument (`argv[1]`), alors on l'affiche sur la sortie standard et on termine ; sinon on affiche *not yet* et on continue.

Pour chercher dans la sortie de `ps` il est possible de lire par lignes avec la fonction de bibliothèque `getline()` :

```
ssize_t getline(char **lineptr, size_t *n, FILE *stream);
```

ou `lineptr` est un pointer vers un pointer vers un tampon qui contiendra la ligne lue du fichier ; `n` est un pointer vers un entier qui contiendra le nombre de caractères lus.

Si on donne `*lineptr=NULL` et `*n=0`, la `getline()` alloue dynamiquement (avec `malloc()`) un tampon de la bonne taille pour contenir la ligne et il renvoie le nombre de caractères lus dans `n`.

Si `*lineptr!=NULL`, la `getline()` fait un appel à `realloc()` pour adapter la taille du tampon et affecte la nouvelle valeur de `n`. Quand on aura terminé d'utiliser `getline()`, c'est à l'utilisateur de libérer la mémoire avec `free()`. Attention : `getline()` retourne -1 quand il n'y a plus rien à lire.

En fin, pour chercher l'occurrence de la sous-chaîne `s2` dans une chaîne `s1`, on peut utiliser la fonction de bibliothèque suivante :

```
int strncmp(const char *s1, const char *s2, size_t n);
```

qui renvoie `0` si la chaîne `s2` est égale au premiers `n` caractères de la chaîne `s1`.

Question 3 Implanter la commande `pps`. S'assurer que les fils se terminent quand le père se termine. ■

Solution : Voici une première solution.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
#include <math.h>
#include <assert.h>
#include <string.h>
#include <sys/types.h>
#include <sys/wait.h>

int main(int argc, char *argv[])
{
    if (argc < 2) {
        printf("Usage : pps <str>\n");
        exit(EXIT_SUCCESS);
    }
    while (1) {
        int pd[2];
        assert(pipe(pd) == 0);
        int ch = fork();
        assert(ch >= 0);
        if (ch == 0) {
            dup2(pd[1], 1);
            close(pd[0]);
            execlp("ps", "ps", "axu", NULL);
            perror("Exec failed");
            exit(EXIT_FAILURE);
        }
        close(pd[1]);
        FILE *f = fdopen(pd[0], "r");

        char *line = NULL;
        size_t len = 0;
        while (getline(&line, &len, f) != -1) {
```

```
35              for (size_t i = 0; i<len-strlen(argv[1]); i++) {
36                  if (strncmp("pps", line+i, 3) == 0) break;
37
38                  if (strncmp(argv[1], line+i, strlen(argv[1])) == 0) {
39                      printf("Found: %s\n", line);
40                      exit(EXIT_SUCCESS);
41                  }
42              }
43          }
44          // deallocate the buffer
45          free(line); line = NULL; len = 0;
46          fclose(f);
47          printf("Not yet\n");
48          wait(NULL);
49          sleep(2);
50      }
51  }
```

Il faut forker et exécuter la commande tout les 2 secondes. Ça peut être utile pour expliquer le fonctionnement de l'allocation de mémoire faite par la `getline()`.

Aussi, il faut faire attention que `ps` retourne la commande `pps` avec l'argument, donc il faut éliminer la ligne correspondante.

À la fine de la boucle `while`, on ferme le descripteur et on libère la mémoire.

Une autre solution est d'utiliser la librairie `regex.h` (voir le `man regex`). ■

3 Signaux

3.1 Timeout

On cherche à s'assurer qu'une commander se termine dans un délai spécifié. On voudrait donc implanter une commande

```
timeout nsec cmd
```

qui exécute la commande `cmd` et si cette dernière ne se termine pas avant `nsec` seconds, il lui envoie un signal `SIG_KILL` et sort avec `EXIT_FAILURE`, sinon elle retourne la sortie de la commande `cmd`.

Question 1 Implanter la commande `timeout`. ■

Solution : Voici un possible solution, très simple. Vous pouvez discuter l'utilisation de la flag `SA_RESTART` :

- Si `n.sa_flags = SA_RESTART`, la `waitpid` retournera le pid du fils avec son status, pour signaler sa terminaison anormale
- Si `n.sa_flags = SA_RESTART`, la `waitpid` retournera un erreur (`errno = EINTR`), et ne récupère pas la valeur de sortie du fils ; il faut alors faire un deuxième `waitpid` pour récupérer le status et éviter que le processus devient zombie.

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
#include <math.h>
#include <assert.h>
#include <string.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <signal.h>

pid_t ch;  // PID du fils

void shandler(int signo)
{
    kill(ch, SIGKILL);
}

int main(int argc, char *argv[])
{
    if (argc < 3) {
        fprintf(stderr, "Usage: %s nsec cmd\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    int nsec = atoi(argv[1]);
    if (nsec < 0) {
```

```
28          fprintf(stderr, "nsec=%d (it must be larger than or equal to 0\n", nsec);
29          exit(EXIT_FAILURE);
30      }
31
32      ch = fork();
33      assert(ch >= 0);
34      if (ch == 0) {
35          execvp(argv[2], &argv[2]);
36          perror("Error executing the command");
37          exit(EXIT_FAILURE);
38      }
39      struct sigaction n, old;
40      int status = 0;
41      n.sa_handler = shandler;
42      // n.sa_flags = 0;
43      n.sa_flags = SA_RESTART;
44      sigemptyset(&n.sa_mask);
45      sigaction(SIGALRM, &n, &old);
46
47      alarm(nsec);
48
49      int ret = waitpid(ch, &status, 0);
50      if (ret < 0) {
51          perror("Interrupted");
52          exit(EXIT_FAILURE);
53      }
54      else if (WIFEXITED(status))
55          return WEXITSTATUS(status);
56      else {
57          printf("Timeout expired, child process killed\n");
58          return EXIT_FAILURE;
59      }
60  }
```

■

3.2 Simuler un sémaphore

D'un point de vu théorique, la communication à mémoire partagée (utilisée par exemple dans le threads) est équivalent à une structure à échange de messages (utilisé par exemple entre processus via les tubes). Dans cet exercice on démontre l'équivalence entre sémaphores et tubes pour réaliser un mécanisme d'exclusion mutuelle.

Un programme est composé d'un processus père et `N` processus fils. Les fils exécutent le code suivant :

```
1   {
2       for (int i=0; i<10; i++) {
3           mysem_wait();
4           printf("Operation 1 : %d - %d\n", getpid(), i);
5           usleep(rand()%100);
```

```
6           printf("Operation 2 : %d - %d\n", getpid(), i);
7           mysem_post();
8           usleep(rand()%1000);
9       }
10      printf("terminé!\n");
11      exit(1);
12  }
```

Les deux `printf` à la ligne 5 et à la ligne 7 simulent des operations qui doivent être exécuté ensemble dans une *section critique*. Autrement dit, les deux `printf` d'un même processus ne doivent pas s'entremêler avec les mêmes opérations d'un autre processus fils. On peut penser que ces opérations simulent deux écritures sur un fichier qui doivent se faire consécutivement.

Pour réaliser l'exclusion mutuelle, on utilisera le père comme *serveur* et le mécanisme de tubes pour communiquer entre père et fils. Les fonctions `mysem_wait()` et `mysem_post()` réalisent l'opération d'exclusion mutuelle comme s'il s'agissait d'un sémaphore.

La stratégie est la suivante :

— Un tube `fp` (fils-père) est utilisé pour envoyer les messages des fils vers le père.

— Le père envoie des signaux `SIG_USR1` vers le fils pour les débloquer.

Donc la `mysem_wait()` doit faire les choses suivantes :

— Envoyer au père sur le tube un message contenant l'identifiant de l'operation (`wait` dans ce cas) et son pid.

— Attendre les *feu vert* du père en attendant un signal `SIG_USR1`;

La `mysem_post()` doit simplement envoyer un message au père contenant l'identifiant de l'opération et son pid.

Le père attend des messages des fils, et garde l'état de la section critique, en envoyant des signaux si nécessaire.

Question 2 Écrire la structure de donnée `sem_msg` que le fils envoie au père sur le tube. Pourquoi il est nécessaire une structure de données au lieu de deux variables separées ? ■

Solution : Voici un exemple de structure :

```
1  struct sem_msg {
2      enum operation op;
3      pid_t pid;
4  };
```

Il faut utiliser une structure pour écrire les deux champs en une seule `write` atomique. ■

Question 3 Écrire le code des fonctions `mysem_wait()` et `mysem_post()`. ■

Solution : On commence par la `mysem_post` qui est plus simple.

```
    msg.op = POST;

    msg.pid = getpid();
    write(resource.fp[1], &msg, sizeof(msg));
}

void install_handler()
{
```

Pour la `mysem_wait()` :

```
{
    struct sem_msg msg;
    msg.op = WAIT;
    msg.pid = getpid();
    write(resource.fp[1], &msg, sizeof(msg));
    sigsuspend(&mask_noblock_usr1);
}

void mysem_post()
{
```

En effet, si on utilise la `pause()` on pourrait avoir un deadlock. Il est utile d'expliquer aux étudiants pourquoi il y a un deadlock avec la `pause` en expliquant l'enchaînement exact d'exécution des processus qui porte sur un deadlock.

Pour resoudre le problème du deadlock, il faut plutôt masquer le signal `SIGUSR1` et le démasquer avec `sigsuspend`. Pour faire ça, on utilise deux variables globales `mask_block_usr1` et `mask_noblock_usr1` et la `sigsuspend`. ■

Question 4 Écrire la structure de données que le père utilise pour garder l'état de la section critique. ■

Solution : Une exemple de structure :

```
// FIFO QUEUE
struct resource {
    int busy;
    pid_t blocked[10];
    int first_blocked;
    int last_blocked;
    int n_blocked;
    int fp[2];
} resource = { .busy = 0, .first_blocked = 0, .last_blocked = 0, .n_blocked = 0};
```

■

Question 5 Écrire le reste du programme. ■

Solution : Enfin le main :

```
    pid_t ch[N];

    assert(pipe(resource.fp) == 0);

    for (int i=0; i<N; i++) {
        ch[i] = fork();
        assert(ch[i] >= 0);
        if (ch[i] == 0) {
            install_handler();
            close(resource.fp[0]);
            sigemptyset(&mask_block_usr1);
            sigaddset(&mask_block_usr1, SIGUSR1);
            sigprocmask(SIG_BLOCK, &mask_block_usr1, &mask_noblock_usr1);
            fils();
        }
    }

    close(resource.fp[1]);

    while(1) {
        struct sem_msg msg;
        int n = read(resource.fp[0], &msg, sizeof(msg));
        if (n<=0) {
            printf("Nothing else to read, exiting\n");
            exit(EXIT_SUCCESS);
        }
        if (msg.op == WAIT) {
            if (resource.busy == 0) {
                resource.busy = msg.pid;
                kill(msg.pid, SIGUSR1);
            }
            else {
                resource.blocked[resource.last_blocked] = msg.pid;
                resource.last_blocked = (resource.last_blocked + 1) % N;
                resource.n_blocked ++;
            }
        }
        else if (msg.op == POST) {
            if (resource.busy == 0)
                printf("Error ! POST received on a free resource !\n");
            else if (resource.n_blocked) {
                pid_t unblock_pid = resource.blocked[resource.first_blocked];
                resource.first_blocked = (resource.first_blocked + 1) % N;
                resource.n_blocked --;
```

```
45                  kill(unblock_pid, SIGUSR1);
46              }
47              else {
48                  resource.busy = 0;
49              }
50          }
51      }
52  }
```

■

4 Threads - part 2

4.1 Producteur/Consommateur avec mutex

Dans cet exercice, on fournit une deuxième implémentation de la structure classique des Producteurs/Consommateurs en utilisant les *mutex* et les *variables conditions* au lieu des sémaphores.

Notes

Pour mémoire, dans POSIX on peut déclarer une variable `pthread_mutex_t` pour représenter un mutex, et une variable de type `pthread_cond_t` pour représenter une variable conditions. Les fonctions pour opérer sur ces variables sont :

```
int pthread_mutex_lock(pthread_mutex_t *m);     // prend le verrou sur le
mutex m
int pthread_mutex_unlock(pthread_mutex_t *m);   // relâche le verrou
```

pour les mutex et

```
// bloque sur la condition et relâche le mutex
int pthread_cond_wait(pthread_cond_t *c, pthread_mutex_t *m);
// réveil un thread bloqué sur la condition
int pthread_cond_signal(pthread_cond_t *c);
// réveil tous les threads bloqués sur la condition
int pthread_cond_broadcast(pthread_cond_t *c);
```

pour les variables condition.

Voici l'enoncé.

Un thread (dit *producteur*) fait la première partie du calcul et mémorise les résultats intermédiaires à fur et mesure qui sont générés dans une structure de données tampon (appelée *mailbox*); un deuxième thread (appelé *consommateur*) récupère ces donnés intermédiaires avant de faire la deuxième partie du calcul. Il s'agit d'une *pipeline* d'exécution : pendant que le thread *producteur* produit l'n-ème donnée, le consommateur peut travailler sur les données produites précédemment.

Le problème qu'on essaye de résoudre est la synchronisation des données entre le producteur et le consommateur. Nous ne connaissons pas la vitesse à laquelle les données sont produites par le producteur, ni la vitesse à laquelle elles sont consommées par le deuxième thread. Il peut arriver que le consommateur essaye de récupérer les données avant qu'elles soient produites. De plus, comme la taille du tampon est limitée, il peut arriver que le producteur produit ses données trop rapidement en remplissant complètement le tampon avant que le consommateur puisse les récupérer. Il faut alors bloquer le producteur quand il n'y a plus de place dans le tampon.

Pour régler ces problèmes, nous allons implémenter la structure de données *mailbox* et les fonctions pour mémoriser et récupérer les données en respectant le consignes suivantes :

— Le thread consommateur doit se bloquer s'il essaye de récupérer des données d'une mailbox vide (producteur trop lent) ;
— Le thread producteur doit se bloquer s'il essaye d'écrire des données dans une mailbox pleine (consommateur trop lent) ;
— Aucune donnée doit être perdue ; toutes les données produites par le producteur sont éventuellement récupérées par le consommateur.

On considère que la mailbox contient `MBOX_SIZE` places disponibles. Aussi, on considère qu'il y a plusieurs producteurs et plusieurs consommateurs qui utiliser la même mailbox.

Question 1 *Structure de donnée* Proposer une structure de donnée `struct mailbox` pour représenter le tampon.

Ensuite, écrire les code de la fonction :

```
void mbox_init(struct mailbox *m);
```

qui initialise la structure pointée par `m`. ■

Solution : Voici une solution possible :

```
#define MBOX_SIZE  4

struct mailbox {
    int h;
    int t;
    int n;
    int array[MBOX_SIZE];
    pthread_mutex_t m;
    pthread_cond_t c_vide;
    pthread_cond_t c_plein;
};

void mbox_init(struct mailbox *mbox)
{
    mbox->n = 0; mbox->h = mbox->t = 0;
    pthread_mutex_init(&mbox->m, NULL);
    pthread_cond_init(&mbox->c_vide, NULL);
    pthread_cond_init(&mbox->c_plein, NULL);
}
```

■

Question 2 *Put et get* Écrire le code des fonctions pour mettre et retirer une donnée de la mailbox :

```
void mbox_put(struct mailbox *m, int d);
int  mbox_get(struct mailbox *m);
```

■

Solution : Dans la solution suivante, il faut faire attention à l'utilisation du `while` pour véfifier la condition de blocage.

```
void mbox_put(struct mailbox *mbox, int d)
{
    pthread_mutex_lock(&mbox->m);
    while(mbox->n == MBOX_SIZE)
        pthread_cond_wait(&mbox->c_plein, &mbox->m);

    mbox->array[mbox->h] = d;
    mbox->h = (mbox->h + 1) % MBOX_SIZE;
    mbox->n++;

    pthread_cond_signal(&mbox->c_vide);

    pthread_mutex_unlock(&mbox->m);
}


int  mbox_get(struct mailbox *mbox)
{
    int d;
    pthread_mutex_lock(&mbox->m);
    while (mbox->n == 0) pthread_cond_wait(&mbox->c_vide, &mbox->m);

    d = mbox->array[mbox->t];
    mbox->t = (mbox->t + 1) % MBOX_SIZE;
    mbox->n--;

    pthread_cond_signal(&mbox->c_plein);

    pthread_mutex_unlock(&mbox->m);
    return d;
}
```

Si on fait `if` à la place de `while` le programme ne marche pas correctement. Pour montrer pourquoi, il faut faire le schéma suivant au tableau ;

— Un thread consommateur C1 fait la `mbox_get()` pendant que la mailbox est vide, et bloque sur la condition `c_vide` ;

— Un thread producteur commence à exécuter la `mbox_put()` et prends le verrou sur le mutex `m`;
— Pendant ce temps, un deuxième consommateur C2 essaye de faire la `mbox_get()`, mais comme le mutex est verrouillé par le producteur, il bloque sur le `pthread_mutex_lock()`;
— Le producteur réveille le première consommateur avec la `pthread_cond_signal()`;
— Le consommateur C1 est désormais réveillé, mais il doit d'abord reprendre le verrou sur le mutex;
— Quand le producteur lève le verrou sur le mutex, il peut arriver que C2 passe en première, prends le verrou, trouve qu'il y a un élément dans la mailbox (`n == 1`), le récupère (`n==0`), et lève le verrou, donnant feu vert à C1.
— Si on utilise `if` au lieu de `while`, le consommateur C1 ne vérifie plus la condition de blocage; il récupère donc un élément qui n'existe pas dans la mailbox, et met la valeur de `n` à `-1`, ce qui n'est pas correct.

Dont la nécessité de vérifier à nouveau la condition de blocage quand le thread se réveille. ■

4.2 Lecteurs/Écrivains

Il s'agit d'un problème classique des système concurrents.

Un ensemble de threads $\mathcal{L}$ veut accéder en lecture à une ressource partagé, pendant que un autre ensemble de threads $\mathcal{E}$ veut accéder en écriture. Les threads de l'ensemble $\mathcal{L}$ peuvent accéder à la ressource au même temps, parce que une lecture ne modifie pas l'état de la ressource, et donc il n'y a pas de problème de consistence. En revanche, un thread de l'ensemble $\mathcal{E}$ modifie la ressource et donc doit y accéder de manière exclusive.

Comme exemple, supposons d'avoir une base de donnée. Les opération de lecture de la base peuvent être fait en parallèle parce qu'il ne modifient pas la base de données. Un opération d'écriture (mis à jour) doit être fait de manière exclusive : aucun autre thread ne peut lire (ni écrire) la base pendant l'écriture.

Le tableau suivant résume les cas possibles. En verticale on montre le opération d'un premier thread, et en horizontale le opérations d'un deuxième thread. La lettre "C" signifie que un accès concurrent est possible; la lettre "E" signifie qu'un accès exclusif est nécessaire.

	T1 lecture	T1 écriture
T2 lecture	C	E
T2 écriture	E	E

Donc, un accès concurrent est possible seulement entre lecteurs. De qu'un écrivain veut accéder à la ressource, ce dernière passe en mode *exclusif*.

Un thread lecteur utilise la structure suivante :

```
struct read_write res;

void *lecteur(void *arg)
{
  while(condition) {
    rw_read_begin(&res);
    /* Operation de lecture */
    rw_read_end(&res);
    /* Autres operations */
  }
}
```

Le thread signale le fait qu'il est en train de commencer la lecture avec l'opération `rw_read_begin();` ensuite, il lit la ressource. Quand il a terminé, il signale la terminaison avec l'appelle `rw_read_end()`.

Un thread écrivain adopte une structure similaire.

```
void *redacteur(void *arg)
{
  while(condition) {
    rw_write_begin(&res);
    /* Operation de lecture */
    rw_write_end(&res);
    /* Autres operations */
  }
}
```

Pour résumer :

— La fonction `rw_read_begin()` bloque le thread lecteur s'il y a déjà un écrivain en train d'écrire.

— La fonction `rw_write_begin()` bloque le thread écrivain s'il y a déjà un autre écrivain en train d'écrire ou au moins un lecteur en train de lire.

Question 3 Écrire la structure de données `struct read_write` avec les champs nécessaires à implémenter un tel comportement et la fonction `rw_init()` pour l'initialiser. ■

Solution : Dans ce première exercice on cherche juste à résoudre le problème le plus simplement possible, sans trop réfléchir aux problèmes de famine. Il suffit de mettre un compteur pour les lecteurs et un pour les écrivains, et une variable condition pour bloquer les lecteurs, et une variable condition pour bloquer les écrivains.

```
struct read_write {
    int nlect;
    int necri;
    pthread_mutex_t m;
    pthread_cond_t lect;
    pthread_cond_t ecri;
};

void rw_init(struct read_write *res)
{
    res->nlect = res->necri = 0;
    pthread_mutex_init(&res->m, NULL);
    pthread_cond_init(&res->lect, NULL);
    pthread_cond_init(&res->ecri, NULL);
}
```

■

Question 4 Implementer les fonctions `rw_read_begin()`,`rw_read_end()`,`rw_write_begin()` et `rw_write_end()`. ■

Solution : Voici un solution :

```
void rw_read_begin(struct read_write *res)
{
    pthread_mutex_lock(&res->m);
    while (res->necri > 0) pthread_cond_wait(&res->lect, &res->m);
    res->nlect++;
    pthread_mutex_unlock(&res->m);
}

void rw_read_end(struct read_write *res)
{
    pthread_mutex_lock(&res->m);
    res->nlect--;
    if (res->nlect == 0) pthread_cond_signal(&res->ecri);
    pthread_mutex_unlock(&res->m);
}

void rw_write_begin(struct read_write *res)
{
    pthread_mutex_lock(&res->m);
    while (res->nlect > 0 || res->necri > 0) pthread_cond_wait(&res->ecri, &res->m);
    res->necri++;
    pthread_mutex_unlock(&res->m);
}

void rw_write_end(struct read_write *res)
{
    pthread_mutex_lock(&res->m);
    res->necri--;
    if (res->necri == 0) {
        pthread_cond_broadcast(&res->lect);
        pthread_cond_signal(&res->ecri);
    }
    pthread_mutex_unlock(&res->m);
}
```

■

Le problème de la *famine* (livelock) consiste dans le fait qu'un thread n'arrive jamais à accéder à une ressource même si tous les autres threads sont actifs. Dans le problème des lecteurs/écrivains le problème de la famine se manifeste pour les écrivains : dans une implémentation simple, il peut arriver que les lecteurs sont toujours en train de lire la ressource : par exemple, avant qu'un lecteur est termine de lire une ressource, un deuxième lecteur commence à lire ; et avant que le deuxième termine, un troisième commence sa lecture ; et avant que le troisième termine, le premier recommence une nouvelle lecture. Le diagramme temporel est montré dans la figure **??**.

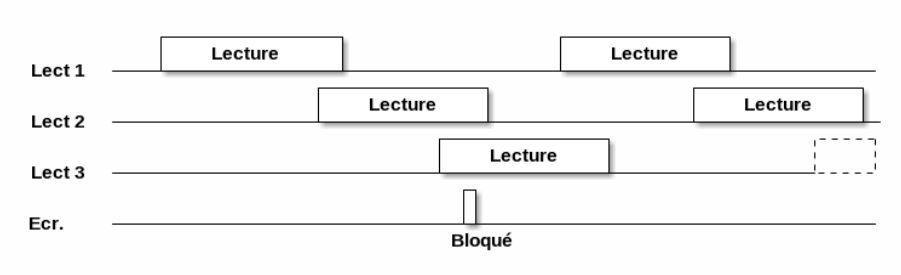

FIGURE 1 – Diagramme temporel montrant le problème de la famine pour les écrivains.

Il faut préciser que le problème de la famine est différent du problème de l'interlocage : seulement l'écrivain est bloqué pendant que les lecteurs monopolise la ressource. Aussi, l'écrivain peut occasionnellement être debloqué : il suffit d'un ralentissement ou d'un alignement particulier de l'exécution des lecteurs pour débloquer l'écrivain.

Pour éviter le problème de la famine, nous pouvons empêcher aux lecteurs de lire la ressource s'il y a un écrivain bloqué.

Question 5 Proposez une nouvelle implémentation de la structure des données `struct read_write` et des fonctions `rw_init()`,`rw_read_begin()`,`rw_read_end()`,`rw_write_begin()` et `rw_write_end()` pour éviter le problème de la famine des écrivains.

Attention : il ne faut pas introduire la famine des lecteurs ... ■

Solution : La solution avec les monitors est très compliqué. D'abord la structure :

```
struct read_write {
    int nlect;
    int necri;
    int nlect_bloque;
    int necri_bloque;

    pthread_mutex_t m;
    pthread_cond_t lect;
    pthread_cond_t ecri;
};
```

On a ajouté 2 compteurs pour prendre note du nombre de lecteurs bloqués et des écrivains bloqués. La `rw_read_begin()` est la fonction la plus compliquée :

```
void rw_read_begin(struct read_write *res)
{
    pthread_mutex_lock(&res->m);

    // S'il y a un écrivain, bloque
    while (res->necri > 0) {
```

```
        res->nlect_bloque++;
        pthread_cond_wait(&res->lect, &res->m);
        res->nlect_bloque--;
    }
    // Il n'y a pas d'écrivains en train d'écrire
    assert(res->necri == 0);

    //Mais peut-être quelqu'un est bloqué, alons j'attends.
    while (res->nlect > 0 && res->necri_bloque > 0) {
        res->nlect_bloque++;
        pthread_cond_wait(&res->lect, &res->m);
        res->nlect_bloque--;
    }

    res->nlect++;
    pthread_mutex_unlock(&res->m);
}
```

L'idée est que d'abord on vérifie qu'il n'y a pas d'écrivains en train d'écrire. Ensuite, on vérifie qu'il n'y a pas d'écrivains bloqué pendant que un autre lecteur est en train de lire. Dans ce cas, on bloque et on incrément le compteur des threads bloqués. Ça empêche aux threads lecteurs de prendre le monopole de la ressource.

Il n'est pas possible de faire un "ou" des deux conditions dans la même boucle, si non il y a un deadlock.

La `rw_read_end()` n'a pas changé :

```
void rw_read_end(struct read_write *res)
{
    pthread_mutex_lock(&res->m);
    res->nlect--;
    if (res->nlect == 0)
        // Comme avant, on reveille l'écrivain
        pthread_cond_signal(&res->ecri);

    pthread_mutex_unlock(&res->m);
}
```

La `rw_write_begin()` est presque comme avant :

```
void rw_write_begin(struct read_write *res)
{
    pthread_mutex_lock(&res->m);
    // Comme avant : s'il y a un lecteur ou un écrivain, je me bloque
    while ((res->nlect > 0) || (res->necri > 0)) {
        res->necri_bloque++;
        pthread_cond_wait(&res->ecri, &res->m);
        res->necri_bloque--;
    }
    res->necri++;
    pthread_mutex_unlock(&res->m);
}
```

Et finalement la `rw_write_end()` :

```
void rw_write_end(struct read_write *res)
{
    pthread_mutex_lock(&res->m);
    res->necri--;
    // priorité aux lecteurs bloqués
    if (res->nlect_bloque > 0)
        pthread_cond_broadcast(&res->lect);
    // après les écrivains
    else pthread_cond_signal(&res->ecri);

    pthread_mutex_unlock(&res->m);
}
```

Ici, on donne priorité aux lecteurs, ensuite aux écrivains. ■

4.3 Message à plusieurs destinataires

On considère un problème avec un seul producteur et `NCONS` consommateurs. Le producteur veux envoyer le même message à tous les consommateurs. Une solution plutôt simple et de faire `NCONS` mailbox, une pour chaque consommateur. Le producteur écrit le message dans toutes les mailbox, et chaque consommateur lit depuis la sienne.

Nous cherchons à réaliser une structure différente :

— une seule mailbox à `NPLACES` places ;
— un message est *consommé* (et sa place peut être libérée) seulement quand tous les `NCONS` consommateurs ont lu le message ;
— si un consommateur a déjà lu tous les messages dans la mailbox et il n'y a pas de nouveaux messages, le consommateur bloque ;
— si toutes les places sont occupés, le producteur bloque.

Par exemple, supposons une mailbox à `NPLACES=4` places, et `NCONS=2` consommateurs. Le producteur d'abord dépose 4 messages en appellant la `put()` 4 fois ; ensuite, le consommateur n. 1 appelle la `get()` 4 fois, et reçoit les 4 messages ; le consommateur n. 2 n'a pas encore rien retiré. Donc, aucun message n'est consommé et aucun place n'est libérée. Si le producteur appelle la `put()` une 5ème fois, il est bloqué ; de même, si le consommateur n.1 appelle la `get()` une 5ème fois, il est bloqué. Quand le consommateur n.2 fait la `get()`, il libère une place ; le producteur est débloqué et dépose un nouveau message ; par conséquent, le consommateur n.1 est débloqué et peut retirer le nouveau message.

Question 6 Écrire une structure de donnée `struct mbox_bc` pour représenter ce type de mailbox, et une fonction `mbox_bc_init()` pour l'initialiser. Pour simplicité, vous pouvez considérer que le nombre de consommateurs est connue à l'avance dans une constante `NCONS`, ainsi que le nombre de places disponibles `NPLACES`. ■

Solution : Il faut déclarer un tableau d'entier `tail[NCONS]` pour représenter l'index du première élément à consommer pour le `i`-ème consommateur ; un tableau de variables conditions `cons[NCONS]`

pour bloquer l'`i`-ème consommateur ; un tableau supplémentaire `navail[NCONS]` pour compter le nombre de consommateur qui ont déjà retiré l'`i`-ème message.

```
struct mbox_bc {
    int head;
    int tail[NCONS];
    int navail[NCONS];

    int array[NPLACES];

    pthread_mutex_t m;
    pthread_cond_t prod;
    pthread_cond_t cons[NCONS];
};
```

L'initialisation est facile :

```
void mbox_bc_init(struct mbox_bc *mbox)
{
    mbox->head = 0;
    pthread_mutex_init(&mbox->m, NULL);
    for (int i=0; i<NCONS; i++) {
        mbox->tail[i] = 0;
        mbox->navail[i] = 0;
        pthread_cond_init(&mbox->cons[i], NULL);
    }
}
```

■

Question 7 Écrire les fonctions :

```
    void mbox_bc_put(struct mbox_bc *mbox, int data);
    int  mbox_bc_get(struct mbox_bc *mbox, int n_consommateur);
```

ou `n_consommateur` est l'index du consommateur (un entier entre `0` et `M-1`). ■

Solution : Voici une solution :

```
void mbox_bc_put(struct mbox_bc *mbox, int data)
{
    pthread_mutex_lock(&mbox->m);
    int n = NPLACES;
    do {
        // Cherche le minimum entre navail
        for (int i=0; i<NCONS; i++)
            n = (mbox->navail[i] < n ? mbox->navail[i] : n);
        if (n == NPLACES) pthread_cond_wait(&mbox->prod, &mbox->m);
```

```
    } while (n == NPLACES);

    mbox->array[mbox->head] = data;
    mbox->head = (mbox->head + 1) % NPLACES;

    for (int i=0; i<NCONS; i++) {
        mbox->navail[i]++;
        pthread_cond_signal(&mbox->cons[i]);
    }

    pthread_mutex_unlock(&mbox->m);
}


int mbox_bc_get(struct mbox_bc *mbox, int index)
{
    int d;
    pthread_mutex_lock(&mbox->m);

    while (mbox->navail[index] == 0)
        pthread_cond_wait(&mbox->cons[index], &mbox->m);

    d = mbox->array[mbox->tail[index]];
    mbox->navail[index]--;
    pthread_cond_signal(&mbox->prod);
    mbox->tail[index] = (mbox->tail[index] + 1) % NPLACES;
    pthread_mutex_unlock(&mbox->m);
    return d;
}
```

■