



Université de Technologie de Compiègne

SR02

Rapport de TD

5 - Librairie de sémaphores

Printemps 2015

Anaïs NEVEUX - Romain PELLERIN
Groupe 2
<i>2 mai 2015</i>

Table des matières

I	Séance 1	3
1	Création d'une bibliothèque P(), V()	4
1.1	Question 1	4
1.1.1	int init_semaphore(void)	4
1.1.2	int detruire_semaphore(void)	4
1.1.3	int val_sem(int sem, int val)	5
1.1.4	int P(int sem)	5
1.1.5	int V(int sem)	6
1.2	Question 2	6
1.3	Question 3	6
1.4	Question 4	7
1.5	Question 5	7
II	Séance 2	9
2	Utilisation de la bibliothèque sur une section critique	10
2.1	Zone critique sans sémaphore	10
2.1.1	main()	10
2.1.2	afaire(int isFather)	11
2.1.3	Sortie du programme sans sémaphore	11
2.2	Ajout d'un sémaphore	12
2.2.1	Code corrigé	12
2.2.2	Sortie du programme avec sémaphore	12
3	Utilisation de la bibliothèque pour un producteur-consommateur	14
3.1	Rôles du consommateur et du producteur	14
3.2	Index de lecture et d'écriture	14
3.3	Code	14
3.4	Sortie du programme	16

Première partie

Séance 1

1 Création d'une bibliothèque P(), V()

1.1 Question 1

1.1.1 `int init_semaphore(void)`

Le but de cette fonction est de créer un ensemble de `N_SEM` sémaphores, qui pourront être utilisés dans le cadre de processus père-fils (suite à un `fork()`) uniquement, à cause de l'utilisation d'`IPC_PRIVATE` comme clé lors de la création. Dans le cas où notre `semid`, représentant notre groupement, ne vaut ni -2 (aucune création), ou -1 (erreur lors d'une précédente création), on retourne -1. Dans le cas d'une erreur lors de la création, un message d'erreur est envoyé et on retourne -2. Dans les autres cas, on instancie une variable de type `union semun` qui servira à l'initialisation des sémaphores grâce au flag `SETALL`. On retourne 0.

`sem_pv.c`

```
int init_semaphore(void) {
    if (semid != -2 && semid != -1) {
        fprintf(stderr, "Déjà créé.\n");
        return -1; // Erreur récupérable avec errno
    }
    else if((semid = semget(IPC_PRIVATE, N_SEM, 0666)) == -1) { // crée un ou
        des sémaphores, alternative : semget(CLE, N_SEM, IPC_CREAT|IPC_EXCL|0666)
        fprintf(stderr, "Erreur lors de la création du groupe de sémaphores.\n");
        return -2; // Erreur récupérable avec errno
    }
    union semun bla; //Création de la structure union
    short val[1] = {0}; //Affectation de la valeur 0
    bla.array = val;
    semctl(semid,0,SETALL,bla); //initialisation de la valeur 0 pour chaque
        semaphore du groupement
    return 0;
}
```

1.1.2 `int detruire_semaphore(void)`

Cette fonction permet de détruire le groupement de sémaphore précédemment créé. Dans le cas où une erreur est survenue lors de la création, ou que celle-ci n'a jamais été faite, la fonction retourne -1. Autrement, elle retourne la valeur renvoyée par `semctl`. Le second argument de la fonction est ignoré, le flag `IPC_RMID` ne prenant pas en compte l'argument.

`sem_pv.c`

```
int detruire_semaphore(void) {
    if (semid == -2 || semid == -1) {
        fprintf(stderr, "Le groupe de sémaphores ne peut être détruit, il n'a
            jamais été créé.\n");
    }
}
```

```
    return -1;
}

return semctl(semid,0,IPC_RMID); // supprime complètement le groupe de sémaphores, le deuxième argument est ignoré
}
```

1.1.3 int val_sem(int sem, int val)

Cette fonction permet d'attribuer une valeur particulière `val` à un sémaphore `sem` indiqué en paramètre. Si le numéro du sémaphore n'est pas compris entre 0 et le nombre de sémaphores, alors la fonction retourne -2 après avoir envoyé un message d'erreur. Si aucun groupement de sémaphores n'a été créé, alors la fonction retourne -1. La fonction instancie le champs `val` d'un `union semun` afin d'utiliser l'opération `setval`. La fonction retourne la valeur renvoyée par `semctl`.

```
sem_pv.c

int val_sem(int sem, int val) {
    if (semid == -2 || semid == -1) {
        fprintf(stderr, "Le groupe de sémaphores n'existe pas.\n");
        return -1;
    }
    else if (sem < 0 || sem >= N_SEM) {
        fprintf(stderr, "Mauvais numéro de sémaphore.\n");
        return -2;
    }

    union semun bla;
    bla.val = val;
    return semctl(semid,sem,SETVAL,bla);
}
```

1.1.4 int P(int sem)

Les primitives P et V sont deux opérations de bases sur les sémaphores. Elles doivent être atomiques.

L'opération P permet de demander un accès à une section critique (le sémaphore est décrémenté s'il est supérieur à 0). Si l'accès n'est pas possible (car déjà égal à 0), on va attendre que le(s) processus utilisant la section critique libère la place (en incrémentant le sémaphore).

La fonction P prend comme argument le numero du sémaphore sur lequel réaliser l'opération. Elle utilise un tableau de structure `sembuf` (ici d'une seule case, car nous ne touchons qu'à un seul sémaphore). Cette structure se compose de trois éléments : `sem_num` qui correspond au numero du sémaphore sur lequel on applique l'opération, `sem_op` qui indique quelle opération effectuer et `sem_flag`, qui ne nous intéresse pas ici. Suivant la valeur que prend le champ `sem_op`, soit on décrémente la valeur du sémaphore, soit on l'incrémente, soit on effectue l'opération Z.

Afin d'appliquer l'opération, on utilise la fonction `semop`, qui prend en paramètre l'identifiant du groupement de sémaphores, notre tableau de structure, ainsi que le nombre de cases du tableau qu'il faut utiliser (ici 1). On retourne la valeur de `semop`. Si `init_semaphore` n'a pas été appelé avant on retourne -1, si le numero du sémaphore est incorrect on retourne -2.

sem_pv.c

```
int P(int sem) {
    if(sem<0 || sem>N_SEM) return -2;
    if(semid==-2 || semid==-1) reeturn -1;
    struct sembuf sops[1];
    sops[0].sem_num = sem; // sémaphore sur lequel on agit
    sops[0].sem_op = -1; // >0 incrémentation de la valeur, <0 decremente de la
        valeur, =0 opération (on attend)
    sops[0].sem_flg = 0; // IPC_NOWAIT, SEM_UNDO ou 0 (ne veut rien dire)
    return semop(semid, sops, 1); // 1 = nb de cases du tableau sops
}
```

1.1.5 int V(int sem)

Cette fonction libère un sémaphore (l'incrémente de 1) et donc débloque les autres processus en attente. La réalisation de la fonction V est en grande partie identique à celle de l'opération P. La différence se fait au niveau du champ `sem_op` qui prend une valeur positive afin d'incrémenter la valeur du sémaphore, tandis que pour l'opération P on utilisait une valeur négative.

sem_pv.c

```
int V(int sem) {
    if(sem<0 || sem>N_SEM) return -2;
    if(semid==-2 || semid==-1) reeturn -1;
    struct sembuf sops[1] = {{sem, 1, 0}};
    return semop(semid, sops, 1); // 1 = nb de cases du tableau sops
}
```

1.2 Question 2

L'utilisation de la commande `gcc -c sem_pv.c` permet d'obtenir l'objet.

ls

```
sem_pv.c sem_pv.h sem_pv.o
```

1.3 Question 3

L'utilisation de la commande `ar rvs libsempv.a sem_pv.o` permet donc d'ajouter les fonctions qui nous avons créés dans la bibliothèque `libsempv.a`. Si celle-ci existe déjà, alors les fonctions seront mises à jour. Le listage des fonctions grâce à la commande `nm -s libsempv.a` nous permet de vérifier si nos fonctions sont belle et bien présentes.

ar rvs libsempv.a sem_pv.o

```
ar: création de libsempv.a
a - sem_pv.o
```

nm -s libsempv.a

```
Indexe de l'archive:
semid in sem_pv.o
init_semaphore in sem_pv.o
```

```

destruire_semaphore in sem_pv.o
val_sem in sem_pv.o
P in sem_pv.o
V in sem_pv.o

sem_pv.o:
00000000000000eb T destruire_semaphore
                U fwrite
0000000000000000 T init_semaphore
000000000000001eb T P
                U semctl
                U semget
0000000000000000 D semid
                U semop
                U __stack_chk_fail
                U stderr
0000000000000277 T V
0000000000000148 T val_sem

```

1.4 Question 4

Voici le code du programme utilisant la bibliothèque créée auparavant. Il n'est pas nécessaire d'inclure les fichiers précédents, étant donné que la bibliothèque est prise en charge automatiquement par les programmes.

sem1.c

```

#include<stdio.h>

int main() {
    init_semaphore(); //initialisation du groupement de sémaphores
    val_sem(2,1); //initialisation de la valeur du sémaphore numéro 2 avec la
                  valeur 1.
    P(2); //Demande d'accès à la section critique en utilisant le sémaphore 2.
    sleep(30); // Attente du programme pendant 30 secondes.
    V(2); //On libère le sémaphore
    destruire_semaphore(); //destruction du groupement de sémaphores.
    return 0;
}

```

1.5 Question 5

Dans le cadre de la création de notre bibliothèque, nous avons défini `N_SEM` à 5. Nous créons donc un groupement de 5 sémaphores. Afin de visualiser la création et la destruction des sémaphores, nous exécutons le programme `sem1.c` en arrière-plan et utilisons la commande `ipcs -s` afin de visualiser tous les sémaphores. Une fois le programme terminé nous réexécutons la commande afin de voir si notre groupement a bel est bien été détruit.

ipcs -s [pendant exécution]

```

----- Tableaux de sémaphores -----
clef      semid    propriétaire perms    nsems

```

0x0052e2c1	0	postgres	600	17
0x0052e2c2	32769	postgres	600	17
0x0052e2c3	65538	postgres	600	17
0x0052e2c4	98307	postgres	600	17
0x0052e2c5	131076	postgres	600	17
0x0052e2c6	163845	postgres	600	17
0x0052e2c7	196614	postgres	600	17
0x0052e2c8	229383	postgres	600	17
0xcbc384f8	294920	anais	600	1
0x00000000	393225	anais	666	5

Comme nous pouvons le constater nous avons bien un groupement de 5 sémaphores créé, il s'agit de celui dont le semid est 393225.

ipcs -s [après exécution]

```
----- Tableaux de sémaphores -----
clef      semid      propriétaire perms      nsems
0x0052e2c1 0          postgres  600          17
0x0052e2c2 32769      postgres  600          17
0x0052e2c3 65538      postgres  600          17
0x0052e2c4 98307      postgres  600          17
0x0052e2c5 131076     postgres  600          17
0x0052e2c6 163845     postgres  600          17
0x0052e2c7 196614     postgres  600          17
0x0052e2c8 229383     postgres  600          17
0xcbc384f8 294920     anais     600          1
```

Nous voyons donc qu'à la fin de notre programme, le groupement de sémaphores est détruit. Il n'y a pas de problèmes de « cores » dans le programme. Si nous n'avions pas fait appel à `détruire_semaphore()` dans notre programme, le groupement aurait persisté, auquel cas nous aurions dû le supprimer manuellement.

Deuxième partie

Séance 2

2 Utilisation de la bibliothèque sur une section critique

Ici, l'enjeu est de constater que lorsque deux processus distincts utilisent une même zone de mémoire partagée (lecture puis écriture), on ne peut assurer l'intégrité des données sans sémaphore. En effet, nous allons constater qu'entre le moment où un processus lit une donnée partagée et le moment où il l'écrit, celle-ci a pu changer entre temps (modifiée par un autre processus).

2.1 Zone critique sans sémaphore

2.1.1 main()

Voici le code de la fonction `main()` qui fait le `fork()` et lance la boucle de 100 itérations (fonction `affaire(int isFather)`) pour les deux processus.

excl-mutu-none.c

```
...
int main() {
    shmid = shmget(IPC_PRIVATE, sizeof(int), IPC_CREAT | IPC_EXCL | 0666); // Création
        d'un espace de mémoire partagé de la taille de 1 int
    adr = (int*) shmat(shmid, NULL, 0); // Shell memory attachment
    *adr = 0;
    srand(time(NULL));
    switch (son = fork()) {
        case 0:
            printf("%d Fils créé\n", getpid());
            affaire(0);
            shmdt(adr); // détache le segment
            break;
        case -1:
            printf("Erreur\n");
            exit(0);
        default:
            printf("%d Père créé.\n", getpid());
            affaire(1);
            wait();
            printf("Valeur finale affichée par le père après fin du fils:\t%d\n",
                *adr);
            shmdt(adr); // détache le segment
            shmctl(shmid, IPC_RMID, 0); // Shell memory control : permet de
                supprimer la mémoire partagée
    }
    return 0;
}
```

2.1.2 `afaire(int isFather)`

Et voici la fonction `afaire(int isFather)` qui fait les 100 itérations. Pour chaque itération, voici ce qui est fait :

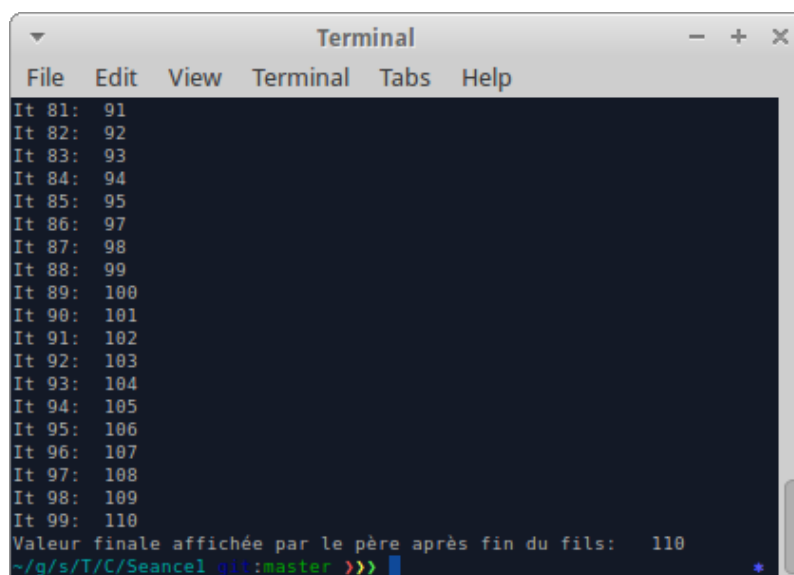
1. Affecter E à une variable A, entière, locale au process.
2. Attendre entre 20 et 100 ms
3. Incrémenter A.
4. Affecter la variable locale A à la variable "partagée" E.
5. Attendre entre 20 et 100 ms
6. Affichage dans le process père de la valeur de E.

L'argument `int isFather` de la fonction `void afaire(int isFather)` permet de savoir si le processus qui appelle cette fonction est le processus père ou non. Si tel est le cas, alors celui-ci affiche à chaque fin d'itération la valeur de E (entier partagé).

excl-mutu-none.c

```
void afaire(int isFather) {
    int j;
    for (j = 0; j<100;j++) {
        var_locale = *adr; // récupération de l'entier partagé
        usleep((rand()%81) + 20); // temps d'attente entre 20 et 100ms
        var_locale++;
        *adr = var_locale; // normalement, *adr vaut *adr + 1, mais ce n'est pas
                           // sûr...
        usleep((rand()%81) + 20); // temps d'attente entre 20 et 100ms
        if (isFather == 1) printf("It %d:\t%d\n",j,*adr);
    }
}
```

2.1.3 Sortie du programme sans sémaphore



```
Terminal
File Edit View Terminal Tabs Help
It 81: 91
It 82: 92
It 83: 93
It 84: 94
It 85: 95
It 86: 97
It 87: 98
It 88: 99
It 89: 100
It 90: 101
It 91: 102
It 92: 103
It 93: 104
It 94: 105
It 95: 106
It 96: 107
It 97: 108
It 98: 109
It 99: 110
Valeur finale affichée par le père après fin du fils: 110
~/g/s/T/C/Seancel git:master >>>
```

FIGURE 2.1 – Sortie du programme

On constate qu'à chaque itération du père, la valeur lue sur le segment partagée ne correspond pas au numéro de l'itération. Cela s'explique par le fait que l'exécution du père peut être interrompue par le CPU pour laisser le fils s'exécuter. Or le fils incrémente lui aussi cet entier partagé. Lorsque le fils et le père ont fini de s'exécuter, après les deux itérations de 100 tours donc, le père affiche la valeur de l'entier partagé. **On constate ici le problème : l'entier devrait valoir 200 (100*2 incrémentations) or ce n'est jamais le cas.**

2.2 Ajout d'un sémaphore

La solution à ce problème est d'ajouter un sémaphore d'exclusion mutuelle (qui vaut 1). Avant chaque accès à la zone critique (lecture) et jusqu'à l'écriture du nouvel entier partagé, on décrémente le sémaphore (cela bloque donc les autres processus qui font la même chose). Après l'écriture du nouvel entier partagé, on l'incrémente à nouveau de 1. Ainsi les autres processus peuvent à leur tour décrémente le sémaphore et modifier l'entier partagé, et ainsi de suite.

2.2.1 Code corrigé

```
excl-mutu.c

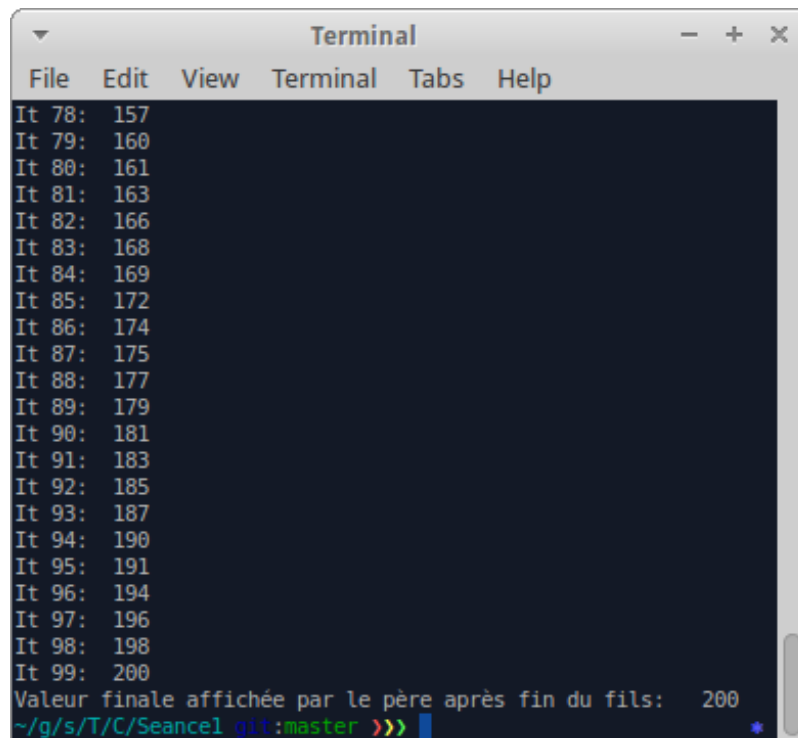
...
void afaire(int isFather) {
    int j;
    for (j = 0; j<100;j++) {
        P(0); // Début de zone critique
        // Incrémentation de l'entier partagé grâce à la variable locale
        V(0); // Fin de zone critique
        usleep((rand()%81) + 20);
        if (isFather == 1) printf("It %d:\t%d\n",j,*adr);
    }
}

int main() {
    ...
    init_semaphore();
    val_sem(0,1);
    switch (son = fork()) {
        case 0:
            ...
        case -1:
            ...
        default:
            ...
            detruire_semaphore();
    }
    return 0;
}
```

2.2.2 Sortie du programme avec sémaphore

On constate dans le screenshot ci-dessous que la valeur finale de l'entier partagé est 200, ce qui est la valeur attendue. À chaque fois que le fils ou le père ont lu et modifié l'entier partagé, ils se

sont exclus mutuellement (bloqués). Ainsi, aucun processus ne pouvait modifier l'entier partagé alors que l'autre n'avait pas fini de le modifier.



```
Terminal
File Edit View Terminal Tabs Help
It 78: 157
It 79: 160
It 80: 161
It 81: 163
It 82: 166
It 83: 168
It 84: 169
It 85: 172
It 86: 174
It 87: 175
It 88: 177
It 89: 179
It 90: 181
It 91: 183
It 92: 185
It 93: 187
It 94: 190
It 95: 191
It 96: 194
It 97: 196
It 98: 198
It 99: 200
Valeur finale affichée par le père après fin du fils: 200
~/g/s/T/C/Seancel git:master >>>
```

FIGURE 2.2 – Sortie du programme

3 Utilisation de la bibliothèque pour un producteur-consommateur

Le but de cet exercice est de produire et consommer (par deux processus respectivement différents) des entiers contenus un tableau (en mémoire partagée) circulaire de taille 5. Le producteur doit s'arrêter quand le tableau est plein (et attendre qu'il y ait une place libre avant de le remplir à nouveau) et inversement pour le consommateur, il ne doit « consommer » que s'il y a au moins une place occupée (donc si le tableau n'est pas vide). **Cela se met en œuvre grâce à deux sémaphores :**

- Un sémaphore correspondra aux emplacements libres dans le tableau, il sera donc initialisé à 5 et sera décrémenté à chaque fois qu'un entier est produit
- Un sémaphore correspondra aux emplacement occupés dans le tableau, il sera donc initialisé à 0 et sera incrémenté à chaque fois qu'un entier est produit

3.1 Rôles du consommateur et du producteur

Par conséquent, le consommateur incrémentera le sémaphore des places libres et décrémentera celui des places occupées à chaque fois qu'il lira un entier. Le producteur fera l'inverse à chaque fois qu'il écrira un nouvel entier.

Quand il n'y aura plus de place libre, le producteur se verra bloqué lorsqu'il essaiera de décrémenter le sémaphore. De même, lorsqu'il n'y aura plus de place occupée, le consommateur sera bloqué lorsqu'il essaiera de décrémenter le sémaphore. **Les contraintes du sujet de l'exercice sont donc respectées.**

3.2 Index de lecture et d'écriture

Un index `int i` propre au consommateur et au producteur sera incrémenté à chaque fois que l'un ou l'autre produira ou consommera. De manière à avoir un tableau circulaire, cet index sera toujours « lu » modulo 5.

3.3 Code

prod-conso.c

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

// CONSTANTES (noms des sémaphores)
int const SEM_PLACES_LIBRES = 0;
int const SEM_PLACES_OCCUPEES = 1;
```

```
int shmid;
int *adr;

int main() {
    int i;
    shmid = shmget(IPC_PRIVATE, 5 * sizeof(int), IPC_CREAT | IPC_EXCL | 0666); // Création d'un espace de mémoire partagé de taille 5 entiers
    adr = (int*) shmat(shmid, NULL, 0); // Shell memory attachment

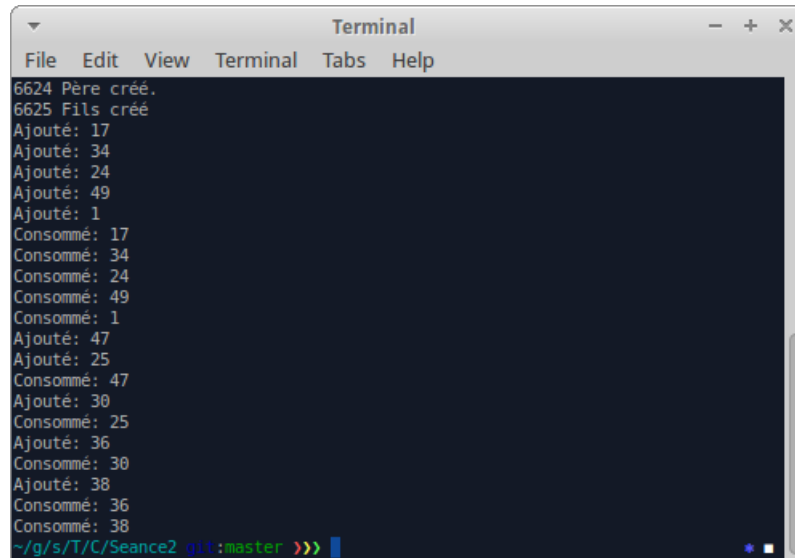
    // Initialisations des 2 sémaphores avec leurs valeurs
    init_semaphore();
    val_sem(SEM_PLACES_LIBRES, 5);
    val_sem(SEM_PLACES_OCCUPEES, 0);

    srand(time(NULL));

    switch (fork()) {
        case 0:
            printf("%d Fils créé\n", getpid()); // producteur
            int tmp;
            for (i = 0; i < 10; i++) { // On aurait pu faire un while(true)
                P(SEM_PLACES_LIBRES);
                tmp = (rand() % 50) + 1;
                adr[i % 5] = tmp;
                printf("Ajouté: %d\n", tmp);
                V(SEM_PLACES_OCCUPEES);
            }
            shmdt(adr); // détache le segment
            break;
        case -1:
            printf("Erreur\n");
            exit(0);
        default:
            printf("%d Père créé.\n", getpid()); // consommateur
            for (i = 0; i < 10; i++) { // On aurait pu faire un while(true)
                P(SEM_PLACES_OCCUPEES);
                printf("Consommé: %d\n", adr[i % 5]);
                V(SEM_PLACES_LIBRES);
            }
            shmdt(adr); // détache le segment
            shmctl(shmid, IPC_RMID, 0); // Shell memory control : permet de
                supprimer la mémoire partagée
            detruire_semaphore();
    }
    return 0;
}
```

3.4 Sortie du programme

Voici la sortie du programme (ci-dessous). On constate que le fils produit les 5 entiers jusqu'à ce que le tableau soit plein, avant que le CPU ne laisse le père s'exécuter. On constate que l'ordre de création (production) des entiers correspond bien à l'ordre de lecture (consommation), les index fonctionnent donc bien. Enfin, chaque processus s'arrête si le tableau est vide ou plein : la contrainte est respectée.



```
6624 Père créé.  
6625 Fils créé  
Ajouté: 17  
Ajouté: 34  
Ajouté: 24  
Ajouté: 49  
Ajouté: 1  
Consommé: 17  
Consommé: 34  
Consommé: 24  
Consommé: 49  
Consommé: 1  
Ajouté: 47  
Ajouté: 25  
Consommé: 47  
Ajouté: 30  
Consommé: 25  
Ajouté: 36  
Consommé: 30  
Ajouté: 38  
Consommé: 36  
Consommé: 38  
~/g/s/T/C/Seance2 git:master >>>
```

FIGURE 3.1 – Sortie du programme