Semáforos

Vamos à corrida! 🏎️💨🏁

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/sem.h>

int main(){
    int shmid, *shmaddr, i;
    shmid=shmget(IPC_PRIVATE, sizeof(int), 00600); // shared memory zone for one integer
    shmaddr=(int*)shmat(shmid, NULL, 0); // attach a virtual address to the zone
    *shmaddr=0;  // access (write) the zone

    fork(); // concurrent access to the shared memory zone

    for(i=1;i<=1000000;i++){
        (*shmaddr)++;
    }

    printf("*shmaddr = %d\n", *shmaddr); // will any process print 2,000,000?

    shmctl(shmid, IPC_RMID, NULL); // removal of the shared memory zone
    return(0);
}
```

Solução? 🚀 Usar semáforos! 🚦

- **Os semáforos são mecanismos de sincronização** entre processos; os **semáforos System V** não são usados para transferência de dados
- Protegem a integridade dos recursos partilhados e são utilizados principalmente para controlar o acesso à memória partilhada
- As operações sobre semáforos são **atómicas**: executam-se de forma indivisível e incluem **incremento**, **decremento**, e **teste ao valor zero**
- Cabeçalhos necessários: `<sys/ipc.h>`, `<sys/sem.h>`
- Fluxo típico de utilização de semáforos: `semget()` → `semctl()` → `semop()` → `semctl()`

semget – semaphore set get 

```
int semget(key_t key, int nsems, int semflg)
```

- Cria/obtém um conjunto (array) de semáforos ou recupera o identificador de um conjunto já existente; retorna o ID ou -1 em caso de erro
- **key** é usada para identificar o conjunto de semáforos
- Key/chave **IPC_PRIVATE** → cria um novo conjunto de semáforos privado
- Chave hexadecimal (e.g., **0x1234abcd**) → cria um novo conjunto de semáforos não privado se ainda não existir (é necessário definir **IPC_CREAT** em **semflg**); sem **IPC_EXCL**, devolve um conjunto existente se a chave já existir
- **nsems** é o número de semáforos a criar
- **semflg** (e.g., **00600**) → define as permissões de acesso; combina-se com **|**
  - **IPC_CREAT** → cria o conjunto de semáforos; **IPC_EXCL** → falha se a chave já existir

semctl – semaphore control

```
int semctl(int semid, int semnum, int cmd, [union semun arg])
```

- **semid** é o identificador do conjunto de semáforos e **semnum** é o número do semáforo dentro desse conjunto
- **cmd** especifica a operação a aplicar ao semáforo identificado por **semid** e **semnum**
- **arg** é opcional e fornece parâmetros adicionais específicos do comando
- ```
  union semun{ // manual definition required before calling semctl
        int val; /* Value for cmd=SETVAL */
        // Other fields for other commands
  } arg;
  ```
- `arg.val=1; semctl(semid, 0, SETVAL, arg); // initialize`
- `semctl(semid, 0, IPC_RMID, NULL); // remove`
  ```

semop – semaphore operation

```
int semop(int semid, struct sembuf *sops, unsigned nsops)
```

- **semid** é o identificador do conjunto de semáforos; o array **sops** especifica as operações a realizar sobre o conjunto identificado por **semid**; **nsops** é o número de operações contidas no array **sops**
- ```
  struct sembuf{
      unsigned short sem_num;
      short sem_op;
      short sem_flg;
  } *sops;
  ```
- Operações: incremento (e.g., **sem_op=1**), decremento (e.g., **sem_op=-1**), e teste ao valor zero (**sem_op=0**); **sem_op** pode ser qualquer número inteiro para incremento/decremento
- **sem_flg=0** (ou **IPC_NOWAIT**, ou **SEM_UNDO**)
  ```

Pequeno lembrete: struct ⚔️ union

```c
#include <stdio.h>

int main() {
    struct MyStruct{
        int i;
        char c;
    } s;
    s.i = 456;
    s.c = 'a'; // ASCII code 97
    printf("s.i = %d, s.c = %c (ASCII code %d)\n", s.i, s.c, s.c);

    union MyUnion{
        int i;
        char c;
    } u;
    u.c = 'a'; // ASCII code 97, stored in the first byte of memory shared by u.i and u.c
    u.i = 2062536546; // Value: 0x7AFFFF62
    // Little-endian representation (LSB first): |62|FF|FF|7A|
    // Big-endian representation (MSB first):    |7A|FF|FF|62|
    printf("u.i = %d = 0x%X, u.c = %c (ASCII code %d)\n", u.i, u.i, u.c, u.c);

    return 0;
}
```

`struct`: os membros são armazenados em locais de memória separados e sequenciais. Escrever num membro não afeta os outros.

`union`: memória sobreposta, com o tamanho do maior atributo. Escrever num membro sobrescreve os dados dos outros.

```
s.i = 456, s.c = a (ASCII code 97)
u.i = 2062536546 = 0x7AEFCF62, u.c = b (ASCII code 98)
```

Solução? 🚀 Usar semáforos! 🚦

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/sem.h>

int main(){
    int i;

    int shmid, *shmaddr;
    shmid=shmget(IPC_PRIVATE, sizeof(int), 00600);  // shared memory zone for one integer
    shmaddr=(int*)shmat(shmid, NULL, 0); // attach a virtual address to the zone
    *shmaddr=0; // access (write) the zone

    int semid;
    union semun { int val; } arg;
    struct sembuf sop[1];
    semid=semget(IPC_PRIVATE, 1, 00600);  // create a semaphore array with one semaphore
    arg.val=1; semctl(semid, 0, SETVAL, arg); // initialize the semaphore counter to 1
    sop[0].sem_flg=0;

    fork(); // concurrent access to the shared memory zone

    for(i=1;i<=1000000;i++){
        // entry section (down)
        sop[0].sem_num=0; sop[0].sem_op=-1; semop(semid, sop, 1);
        // critical section
        (*shmaddr)++;
        // exit section (up)
        sop[0].sem_num=0; sop[0].sem_op=1; semop(semid, sop, 1);
    }
    printf("*shmaddr = %d\n", *shmaddr); // will any process print 2,000,000? Yes!

    shmctl(shmid, IPC_RMID, NULL); // removal of the shared memory zone
    semctl(semid, 0, IPC_RMID, NULL); // comment and practice ipcs and ipcrm

    return(0);
}
```

Comandos Linux para semáforos 💻

- Listar conjuntos de semáforos (**i**nter-**p**rocess **c**ommunication **s**tatus –**s**emaphore):

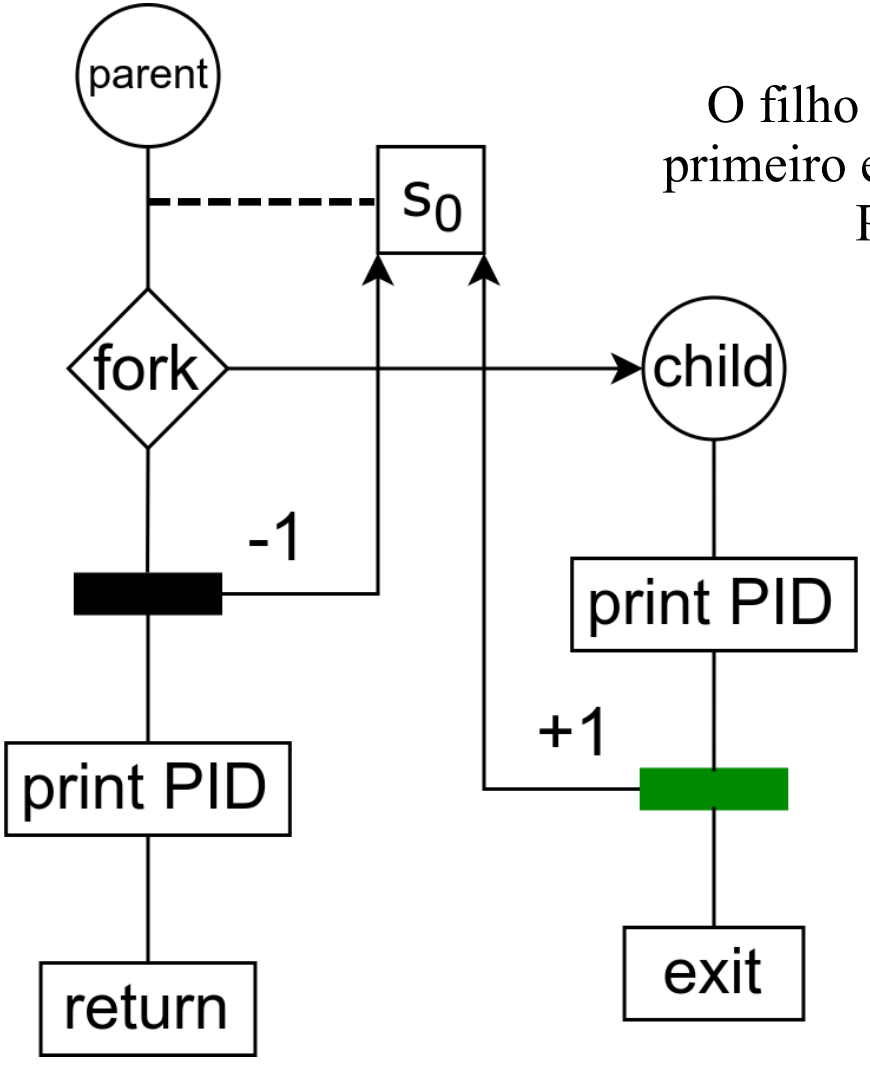
```
ipcs -s
```

- Remover conjuntos de semáforos:

```
ipcrm -s id  # by semaphore array ID
ipcrm -S key # by key
```

3.1.a

(variante do exercicio 2.1) Codifique um programa que envolva dois processos - pai e filho - e recorra a semáforos para sincronização de forma a garantir o seguinte comportamento: a) o processo pai mostra o seu PID depois do filho mostrar o seu PID; b) o comportamento reverso de a).



O filho mostra o seu PID primeiro e o pai mostra o seu PID depois

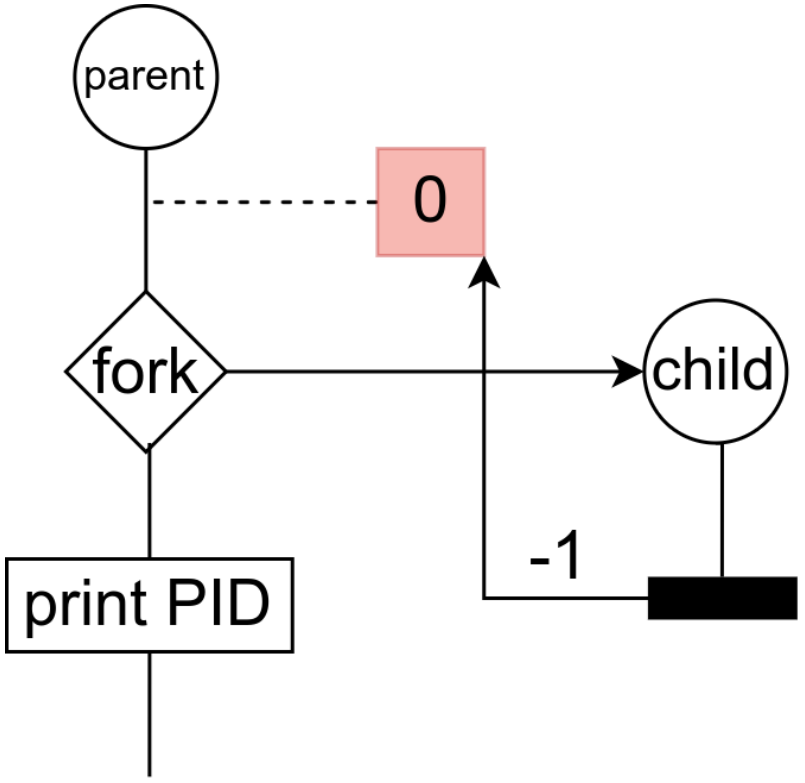
exercise3.1.a

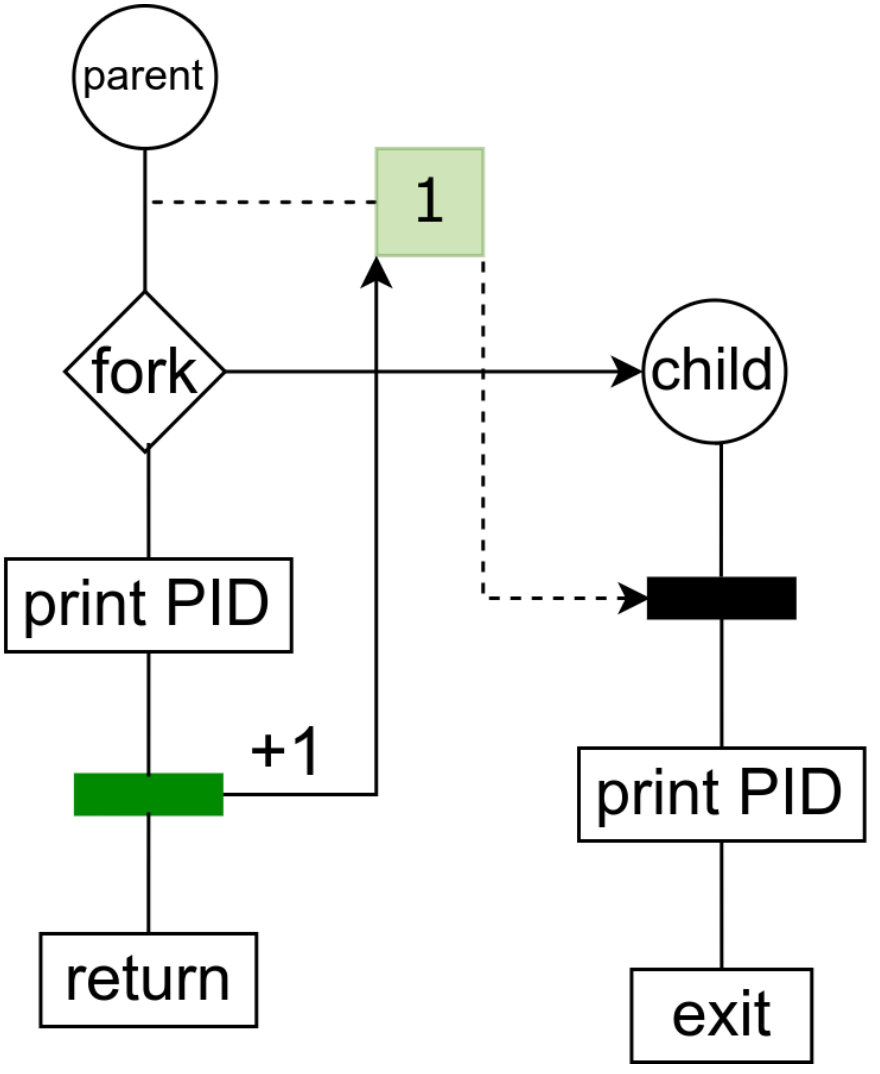
O filho mostra o seu PID primeiro e o pai mostra o seu PID depois

```c
// 3.1.a
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int main(){
    int semid;
    union semun { int val; } arg;
    struct sembuf sop[1];

    semid=semget(IPC_PRIVATE, 1, 00600); // semaphore 0: to block the Parent
    arg.val=0; semctl(semid, 0, SETVAL, arg);
    sop[0].sem_flg=0;

    if(fork()==0){
        printf("CHILD PID: %d\n", getpid());
        sop[0].sem_num=0; sop[0].sem_op=1;
        semop(semid, sop, 1);
        exit(0);
    }

    sop[0].sem_num=0; sop[0].sem_op=-1; semop(semid, sop, 1);
    printf("PARENT PID: %d\n", getpid());

    semctl(semid, 0, IPC_RMID, NULL);
    return 0;
}
```

3.1.b

(variante do exercicio 2.1) Codifique um programa que envolva dois processos - pai e filho - e recorra a semáforos para sincronização de forma a garantir o seguinte comportamento: a) o processo pai mostra o seu PID depois do filho mostrar o seu PID; b) o comportamento reverso de a).

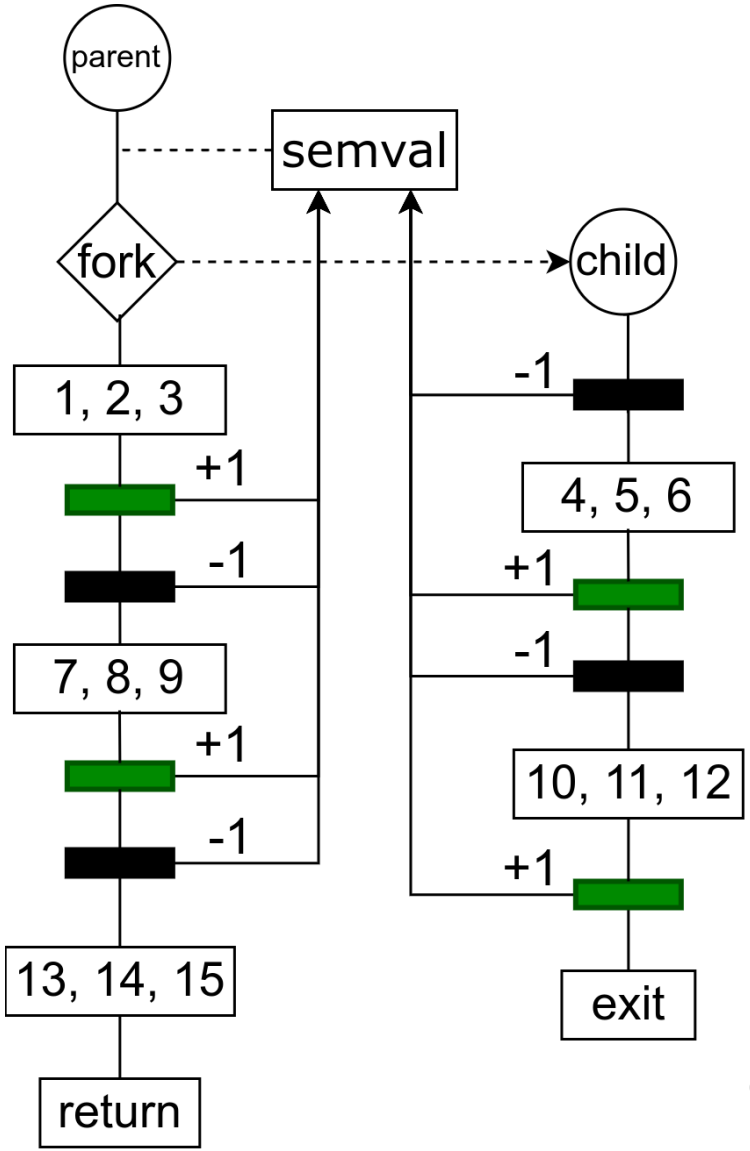




O pai mostra o seu PID primeiro e o filho mostra o seu PID depois

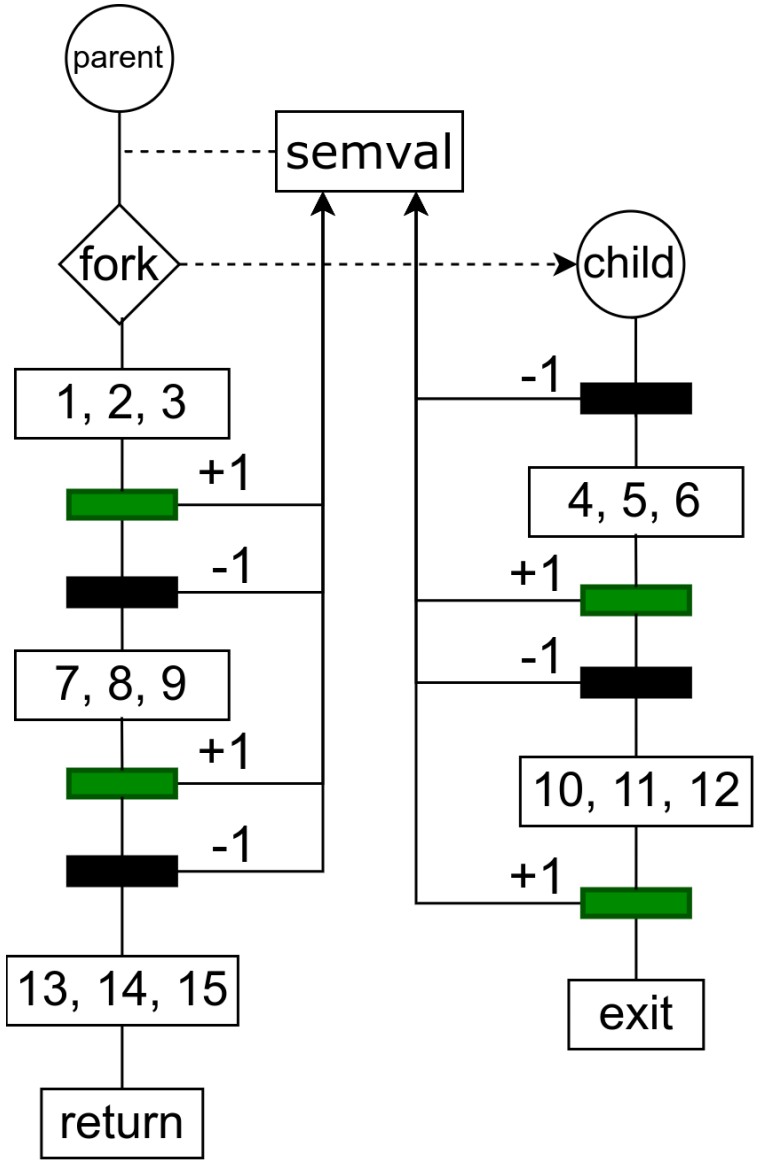
3.2.a

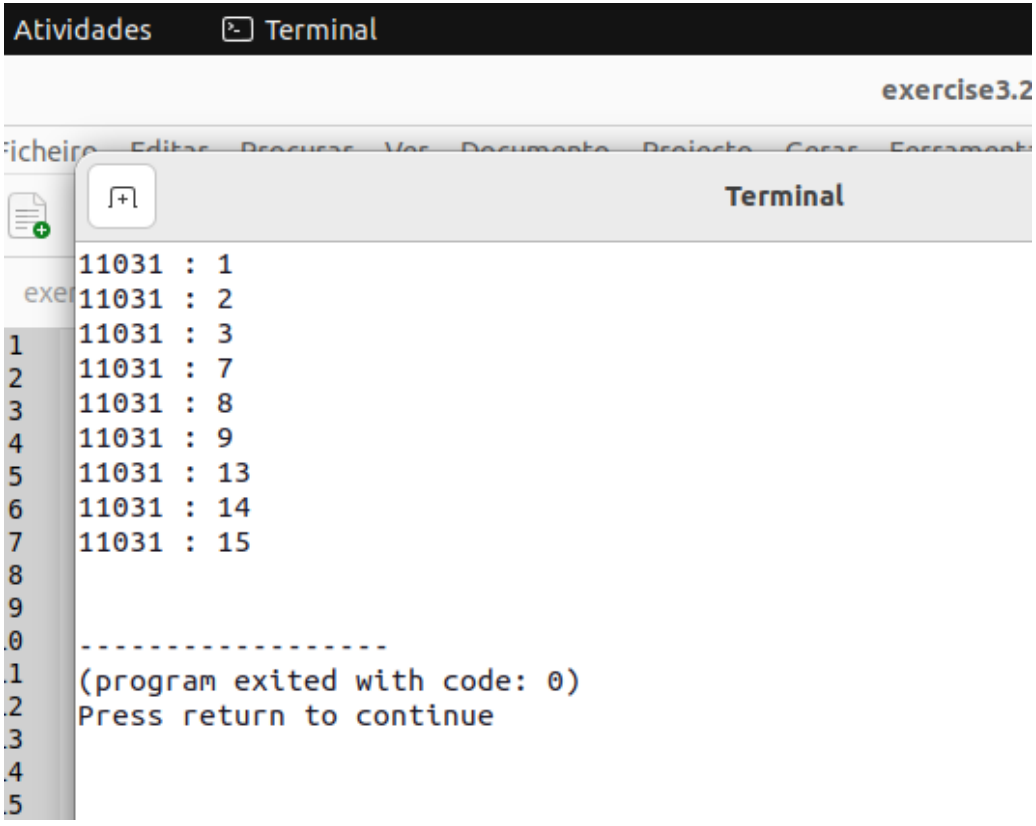
(variante do exercício 2.2) Usando **semáforos** para sincronização, construa um programa no qual dois processos (pai e filho) contam alternadamente até 15 (mostrando a evolução do contador): o processo pai conta de 1 a 3, dando de seguida a vez ao filho; este conta de 4 a 6, passando a vez novamente ao pai; o pai contar de 7 a 9, dando de seguida a vez ao filho; este conta de 10 a 12, passando a vez novamente ao pai, após o que pode terminar; finalmente, o pai conta de 13 a 15, após o que termina. a) Demonstre que um semáforo não é suficiente; b) Resolva o problema com dois semáforos.



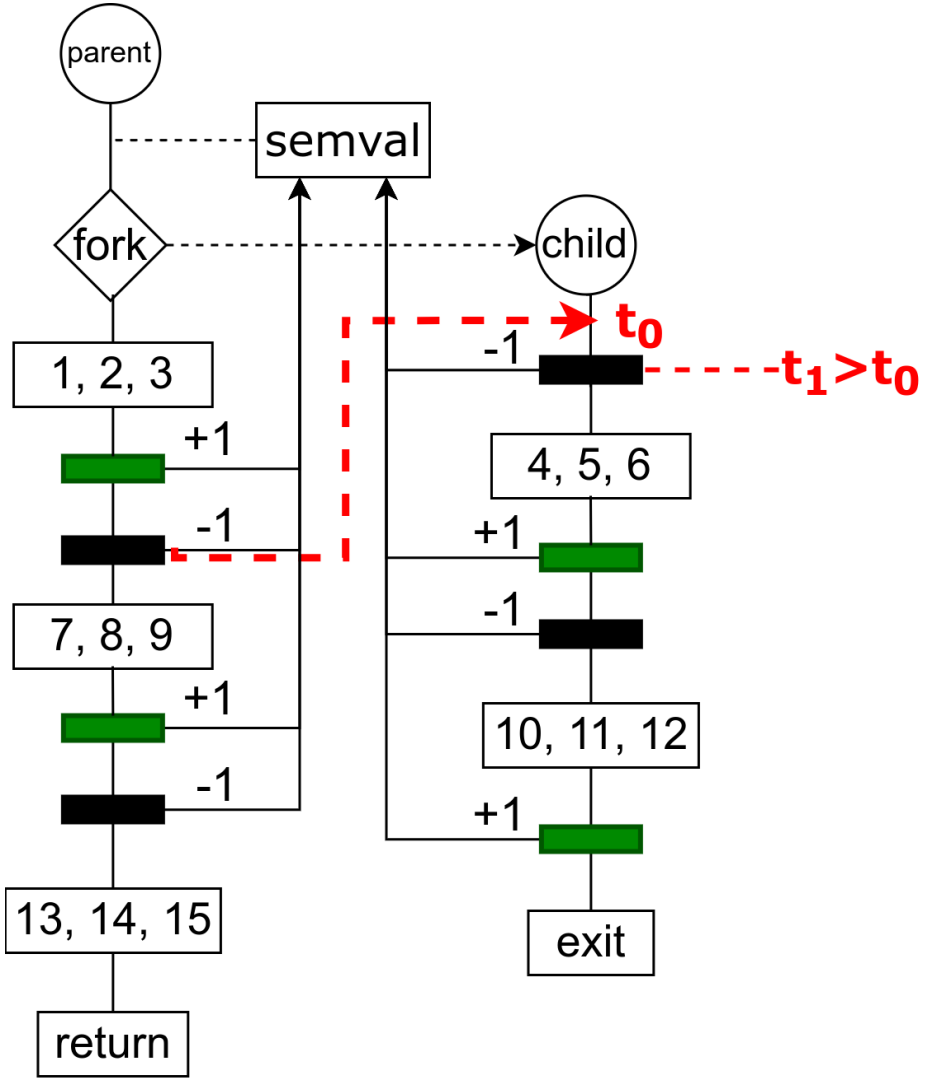

```c
if (fork()==0) { // Child
    sleep(1); // execute with(out) sleep
    sop[0].sem_num=0;sop[0].sem_op=-1; semop(semid,sop,1);
    count(4,6);
    sop[0].sem_num=0;sop[0].sem_op=+1; semop(semid,sop,1);

    sop[0].sem_num=0;sop[0].sem_op=-1; semop(semid,sop,1);
    count(10,12);
    sop[0].sem_num=0;sop[0].sem_op=+1; semop(semid,sop,1);

    exit(0);
}

count(1,3);
sop[0].sem_num=0; sop[0].sem_op=+1; semop(semid, sop, 1);

sop[0].sem_num=0; sop[0].sem_op=-1; semop(semid, sop, 1);
count(7,9);
sop[0].sem_num=0; sop[0].sem_op=+1; semop(semid, sop, 1);

sop[0].sem_num=0; sop[0].sem_op=-1; semop(semid, sop, 1);
count(13,15);
```

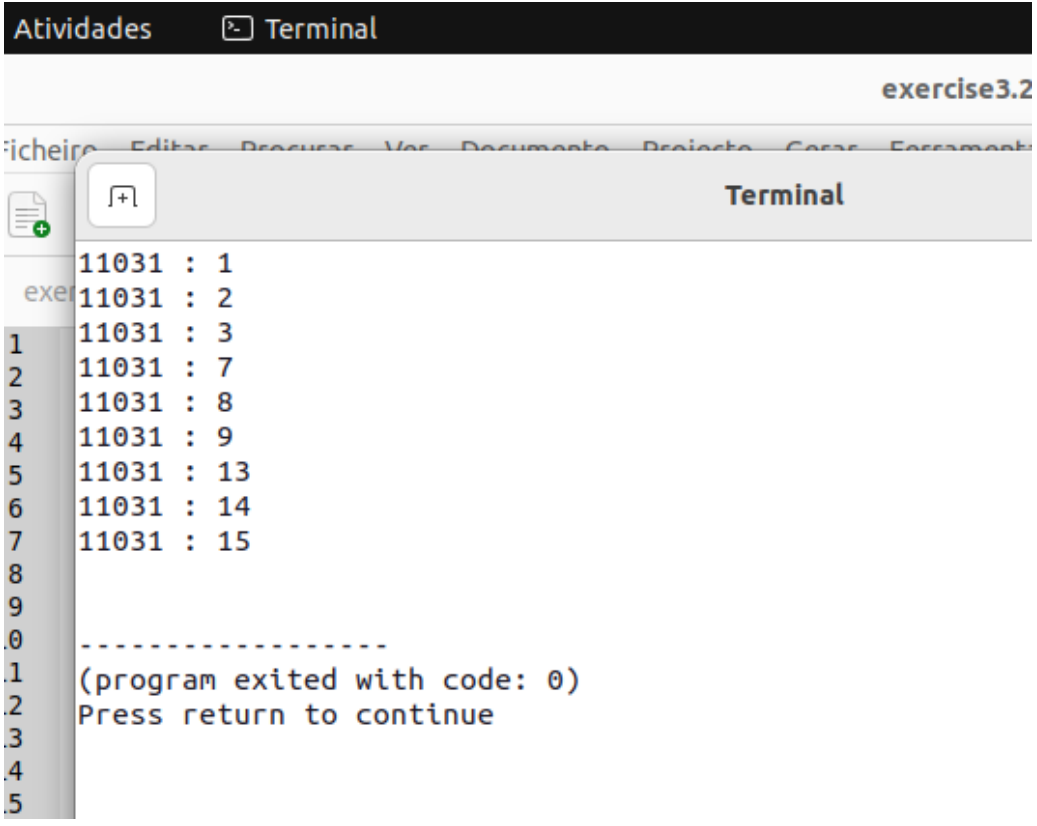
exercise3.2.a





exercise3.2.a

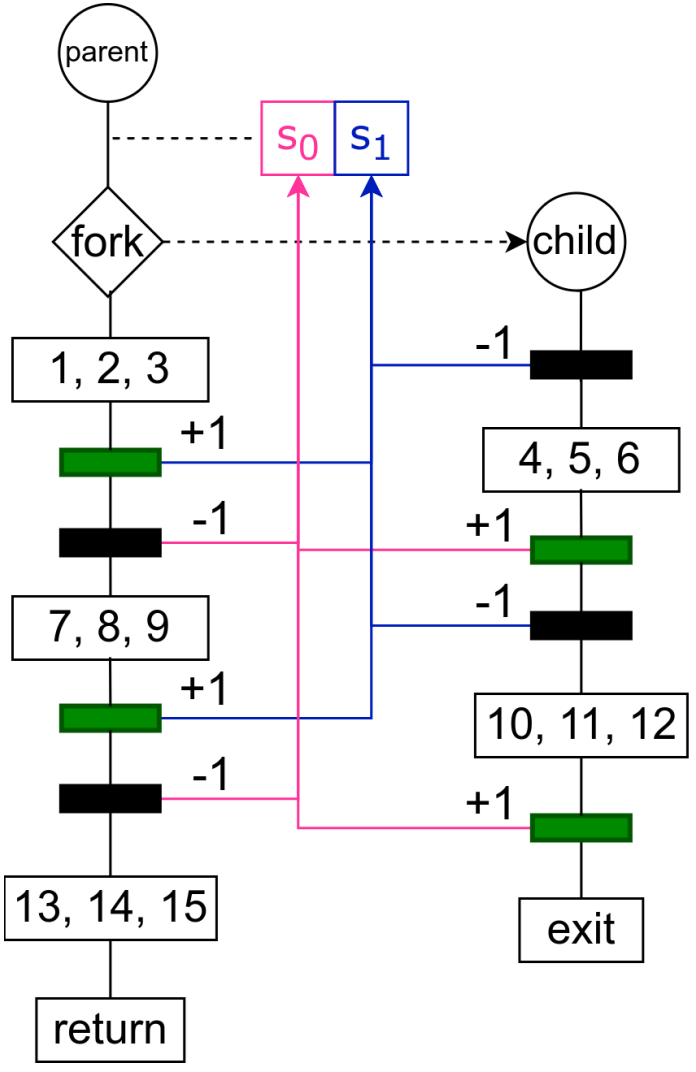




exercise3.2.a

3.2.b

(variante do exercício 2.2) Usando semáforos para sincronização, construa um programa no qual dois processos (pai e filho) contam alternadamente até 15 (mostrando a evolução do contador): o processo pai conta de 1 a 3, dando de seguida a vez ao filho; este conta de 4 a 6, passando a vez novamente ao pai; o pai contar de 7 a 9, dando de seguida a vez ao filho; este conta de 10 a 12, passando a vez novamente ao pai, após o que pode terminar; finalmente, o pai conta de 13 a 15, após o que termina. a) Demonstre que um semáforo não é suficiente; b) Resolva o problema com dois semáforos.



exercise3.2.b

3. (variante do exercício 2.3) Codifique um programa em que um processo filho envie inteiros (lidos do teclado) ao seu pai, este soma-lhes uma unidade e devolve-os ao filho, que apresenta o resultado no ecrã; após o envio do inteiro `0` ao pai, o filho não lhe deve enviar mais inteiros, devendo ambos terminar. Use memória partilhada para troca de dados e **semáforos** para sincronização.

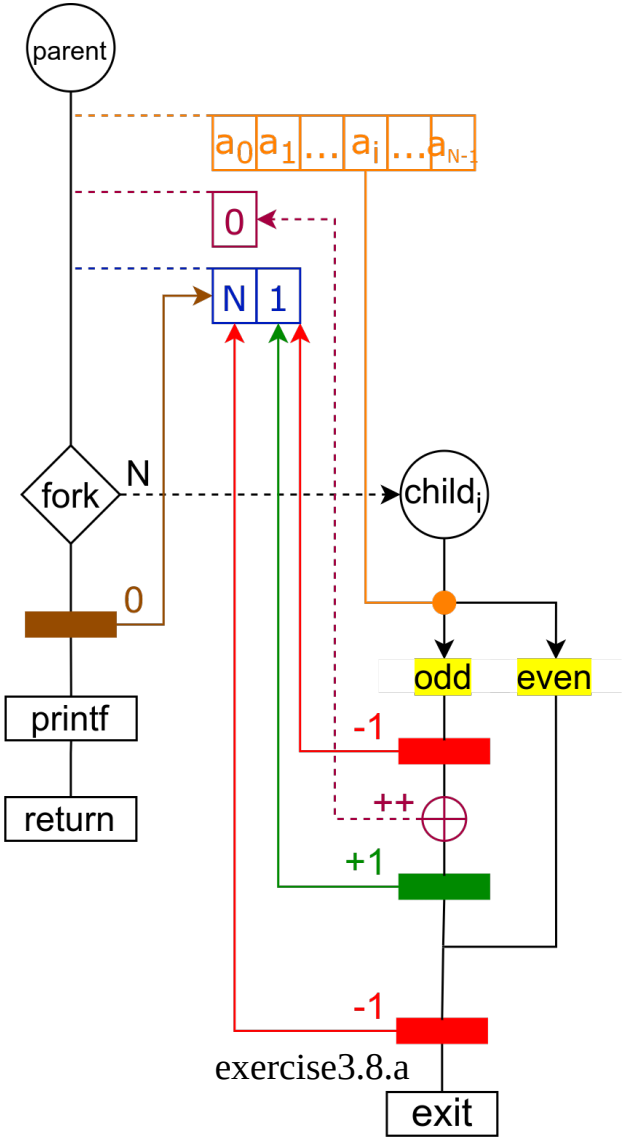
4. (variante do exercício 2.4) Considere um processo pai e 2 filhos: o 1º filho criado lê um inteiro do teclado e envia-o ao 2º filho; este soma uma unidade ao número recebido e envia-o ao pai; por seu turno, este soma uma unidade ao número recebido e envia-o ao 1º filho, que o mostra no ecrã; este comportamento repete-se até que o 1º filho leia o número `0` do teclado, situação em este circuito ainda deve ser feito, mas pela última vez. Resolva o problema usando memória partilhada para troca de dados e **semáforos** para sincronização.

5. (variante do exercício 2.5) Codifique um programa que se baseie em dois processos para o cálculo da soma dos primeiros 100 números inteiros positivos. O processo pai deve calcular a soma dos primeiros 50 números e o processo filho a soma dos últimos 50 números. O processo pai apresentará a soma global. Use memória partilhada para troca de dados e **semáforos** para sincronização.

6. (variante do exercício 2.6) Codifique um programa para calcular a soma dos primeiros 100 números positivos, à custa de somas parciais: um processo pai será responsável por criar 10 filhos; o primeiro filho calculará a soma parcial `1 + 2 + ... + 10`, o segundo filho calculará a soma `11 + 12 + ... + 20`, etc.; o processo pai calculará a soma final e apresentará o resultado. Use memória partilhada para troca de dados e **semáforos** para sincronização.

7. (variante do exercício 2.7) Construa um programa para calcular a soma dos quadrados dos primeiros 10 números positivos, com base num processo pai e em 10 filhos, devendo o cálculo ser efetuado em Round-Robin (com um processo ativo de cada vez): depois de criar os 10 filhos, o pai passa a vez ao 1º filho, o qual calcula e acumula o quadrado de 1; por sua vez o 1º filho passa a vez ao 2º filho, o qual calcula e acumula o quadrado de 2, passa a vez ao 3º filho, e assim sucessivamente ... até que o 10º filho, depois de calcular e acumular o quadrado de 10, passa a vez ao pai, que apresenta a soma final. Use memória partilhada para troca de dados e **semáforos** para sincronização.

8. (variante do exercício 2.8) Escreva um programa onde um processo pai cria um array `A` de `N` inteiros em memória partilhada e preenche-o com números aleatórios. Caberá depois a `N` processos filhos verificar se cada célula do array é ímpar (um filho criado em n'ésimo lugar analisará a célula de índice n do array). Caberá ao pai apresentar o total de ímpares detetados. Resolva o problema para as seguintes situações: a) o array `A` é preenchido antes da criação dos filhos; b) o array `A` é preenchido depois da criação dos filhos. Use memória partilhada para troca de dados e **semáforos** para sincronização.


parent
a_0 a_1 ... a_i ... a_{N-1}
0
N 1
fork
N
$child_i$
0
odd
even
printf
-1
return
++
+1
-1
exercise3.8.a
exit

9. (variante do exercício 2.9) Escreva um programa que implemente o seguinte cenário: um processo pai cria 10 processos filhos; antes de terminar, cada filho mostra o seu PID; no entanto, primeiro devem ser os filhos de PID par a mostrar o seu PID, e só depois os filhos de PID ímpar; o pai só deve terminar depois de todos os filhos terem mostrado o seu PID. Resolva o problema usando **semáforos** para sincronizar os processos.