Operating Systems, Practice Session 5

Synchronization

T. Tolga Sarı (sarita@itu.edu.tr)
Sultan Çoğay (cogay@itu.edu.tr)
Doğukan Arslan (arslan.dogukan@itu.edu.tr)

March 23, 2022

Today

Operating Systems, PS 5

Synchronization

Mutex Usage

Semaphore Usage

Signal Mechanism in Linux

Examples

Synchronized Operation of Process or Threads

- Sometimes synchronization is needed between different processes or between threads implemented in the same process because these threads want to access a shared resource provided by the operating system or maintained by the process itself in order to perform their tasks.

- For example, threads are trying to access a log file. If two threads try to write to the log file at the same time, the logs written to the file will be mixed up and become unreadable.

Mutex Creation

- Where default mutex attributes are appropriate, the macro `PTHREAD_MUTEX_INITIALIZER` can be used to statically generate mutexes.
- Dynamically creation of mutex the parameter `attr` with a call to `int pthread_mutex_init()` with the specified parameter `NULL` produces the same result, except that no error checking is performed.

Mutex Operations

- For each resource shared by different threads, a Mutex is created to regulate access to the resource:
  - `pthread_mutex_t thread`
- The thread dealing with the source tries to take ownership of the Mutex (Acquire).
  - `int pthread_mutex_lock(pthread_mutex_t *mutex);`
- While the thread holding the Mutex completes the Critical Section(CS) and leaves the Mutex, the other thread waiting for the Mutex to be released is awakened and takes ownership of the Mutex and gains access to the shared resource.
  - `int pthread_mutex_unlock(pthread_mutex_t *mutex);`
- To terminate the mutex created at runtime afterwards:
  - `int pthread_mutex_destroy(pthread_mutex_t *mutex);`

Mutex Example

```c
#include <pthread.h>
#include <stdlib.h>
#include <stdio.h>
int myglobal;
pthread_mutex_t lock;
void* thread_function(void *arg){
  int i,j;
  // changing the value of myglobal in thread_function
  for(i=0;i<20;i++){
    pthread_mutex_lock(&lock); //Entering the critical section
    j=myglobal;
    j=j+1;
    printf(".");
    // to force writing all user-space buffered data to stdout
    fflush(stdout);
    myglobal=j;
    pthread_mutex_unlock(&lock); //Exiting the critical section
    sleep(1);
  }
  pthread_exit(NULL);
}
```

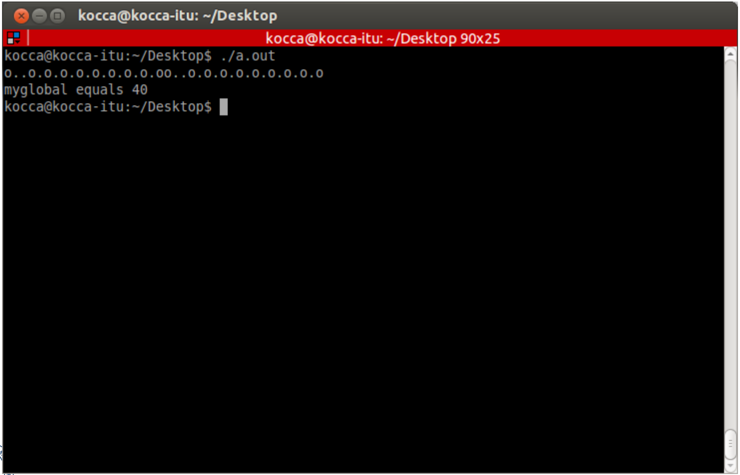
Mutex Example

```c
int main(void){
  pthread_t mythread;
  int i;
  myglobal=0;
  if (pthread_mutex_init(&lock, NULL) != 0)
  {
    printf("\n mutex init failed\n");
    return 1;
  }
  // creating a thread using thread_function as the start routine
  if(pthread_create(&mythread,NULL,thread_function,NULL)){
    printf("error creating thread");
    abort();
  }
```

Mutex Example

```c
  // changing the value of myglobal in main()
  for(i=0;i<20;i++){
    pthread_mutex_lock(&lock); //Entering the critical section
    myglobal = myglobal+1;
    printf("o");
    // to force writing all user-space buffered data to stdout
    fflush(stdout);
    pthread_mutex_unlock(&lock); //Exiting the critical section
    sleep(1);
  }
  pthread_join(mythread, NULL);
  pthread_mutex_destroy(&lock);
  printf("\nmyglobal equals %d\n",myglobal);
  // to block main to support the threads it created until they terminate
  pthread_exit(NULL);
}
```

Mutex Example



```
kocca@kocca-itu:~/Desktop$ ./a.out
o..o.o.o.o.o.o.o.o.oo..o.o.o.o.o.o.o.o.o
myglobal equals 40
kocca@kocca-itu:~/Desktop$ 
```

Semaphore

- POSIX semaphores provide the necessary synchronization infrastructure to access a common resource used by different threads or processes.
- Instead of locking and unlocking on semaphores, operations such as increasing and decreasing the semaphore value can be performed.

Semaphore Operations

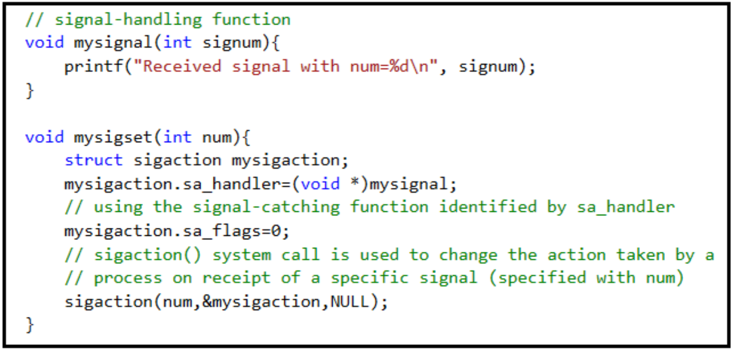
- By including the library `<semaphore.h>`, functions that can be used in semaphore operations can be accessed.
- To create semaphore:
  - `sem_init(sem_t *sem, int pshared, unsigned int value);`
    - `pshared == 0` –> The semaphore is used within the threads of the process. Therefore, it can be kept in a global variable or in a allocated space on the heap, without the need to use shared memory.
    - `pshared != 0` –> The semaphore can be used between different processes. In this case, the address in the first parameter should point to a location on shared memory.
  - If `sem_init()` is called more than once for the same semaphore, the system cannot be stable. Therefore, it should be guaranteed that each semaphore is initialized only once.
- To terminate the semaphore:
  - `int sem_destroy(sem_t *sem);`
    - Before the semaphore is terminated, the memory region where it is kept should not be freed.

Semaphore Operations

- To lock or wait for a semaphore:
    - `int sem_wait(sem_t *sem);`
        - The value of the semaphore is decremented by 1.
        - If the corresponding semaphore value is greater than 1, the reduction is performed instantly and the function returns.
        - If the value of the semaphore is already equal to 0, the sem_wait function waits for the value of the semaphore to increase by 1. When it increases, it immediately decreases by 1 and the function returns.
- To release or signal a semaphore:
    - `int sem_post(sem_t *sem);`
        - It is used to increase the value of the semaphore by 1.
        - If the value of the semaphore is already 0 and is blocked because another process is waiting for the same semaphore, the corresponding process is awakened.
        - In the above case, if multiple processes are blocked while waiting for the same semaphore, it is not guaranteed which process will be woken up.
- To get the current value of an existing semaphore:
    - `sem_getvalue(sem_t *sem, int *value)`

Handling Signals

- Necessary header files for handling signals:
  - signal.h
  - sys/types.h



```
// signal-handling function
void mysignal(int signum){
    printf("Received signal with num=%d\n", signum);
}

void mysigset(int num){
    struct sigaction mysigaction;
    mysigaction.sa_handler=(void *)mysignal;
    // using the signal-catching function identified by sa_handler
    mysigaction.sa_flags=0;
    // sigaction() system call is used to change the action taken by a
    // process on receipt of a specific signal (specified with num)
    sigaction(num,&mysigaction,NULL);
}
```

Handling Signals

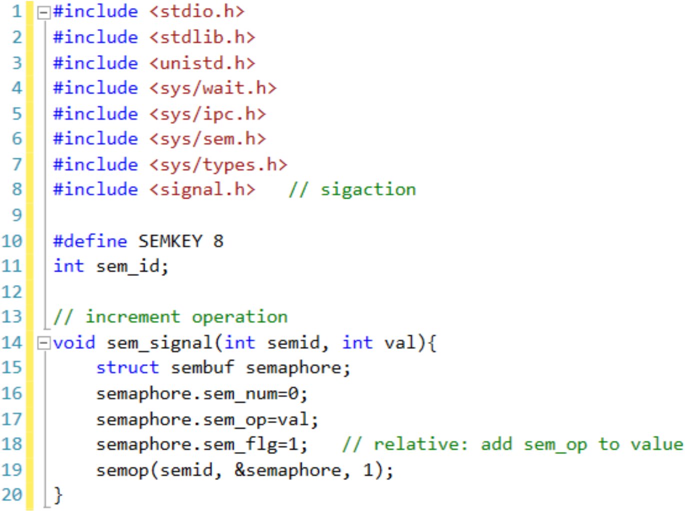
- Sending a signal (specified with `num=sig`) from a process to another process (with given `pid`):

```
int kill(pid_t pid, int sig);
```

- Waiting for a signal:

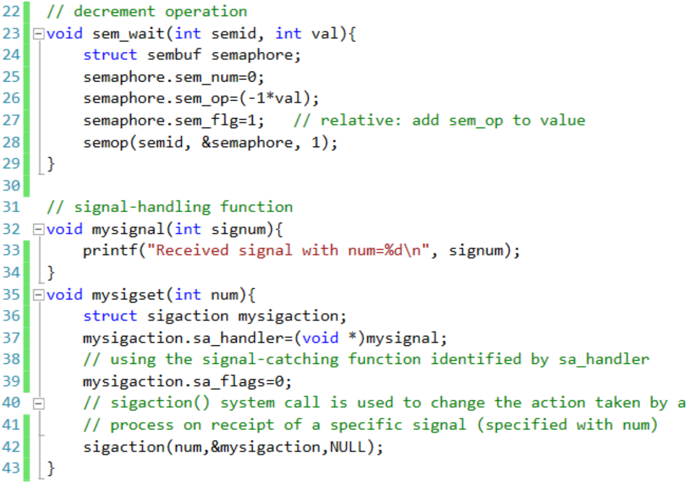
```
int pause(void);
```

Example 1



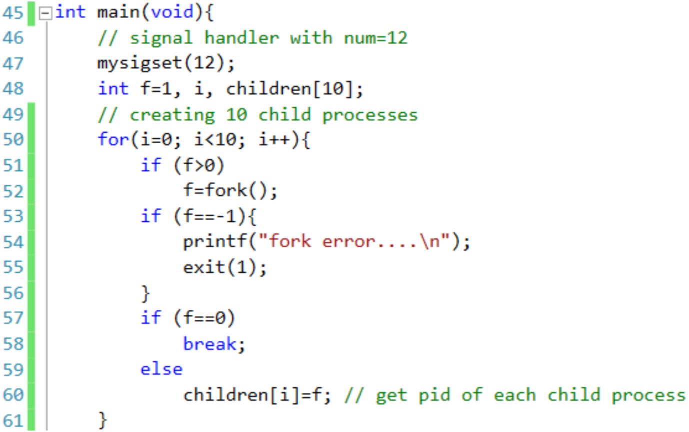
```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <sys/types.h>
#include <signal.h>   // sigaction

#define SEMKEY 8
int sem_id;

// increment operation
void sem_signal(int semid, int val){
    struct sembuf semaphore;
    semaphore.sem_num=0;
    semaphore.sem_op=val;
    semaphore.sem_flg=1;   // relative: add sem_op to value
    semop(semid, &semaphore, 1);
}
```

Example 1



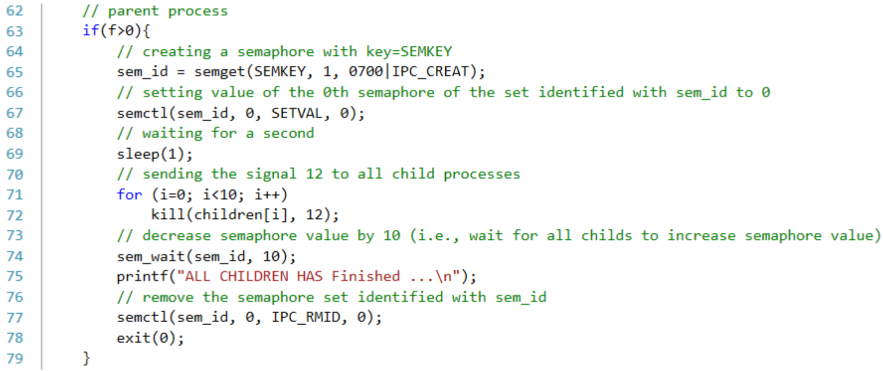
```c
// decrement operation
void sem_wait(int semid, int val){
    struct sembuf semaphore;
    semaphore.sem_num=0;
    semaphore.sem_op=(-1*val);
    semaphore.sem_flg=1;   // relative: add sem_op to value
    semop(semid, &semaphore, 1);
}

// signal-handling function
void mysignal(int signum){
    printf("Received signal with num=%d\n", signum);
}
void mysigset(int num){
    struct sigaction mysigaction;
    mysigaction.sa_handler=(void *)mysignal;
    // using the signal-catching function identified by sa_handler
    mysigaction.sa_flags=0;
    // sigaction() system call is used to change the action taken by a
    // process on receipt of a specific signal (specified with num)
    sigaction(num,&mysigaction,NULL);
}
```

Example 1



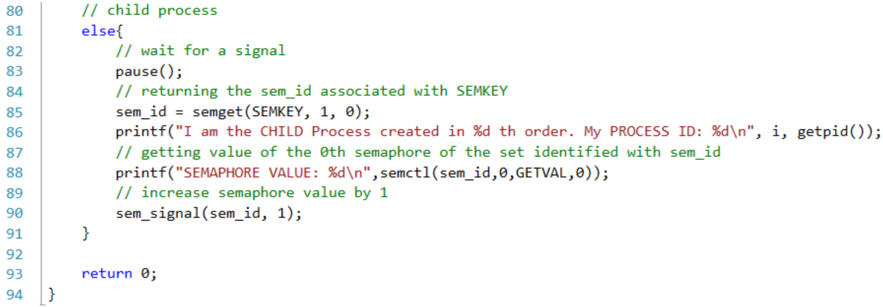
```c
int main(void){
    // signal handler with num=12
    mysigset(12);
    int f=1, i, children[10];
    // creating 10 child processes
    for(i=0; i<10; i++){
        if (f>0)
            f=fork();
        if (f==-1){
            printf("fork error....\n");
            exit(1);
        }
        if (f==0)
            break;
        else
            children[i]=f; // get pid of each child process
    }
```

Example 1



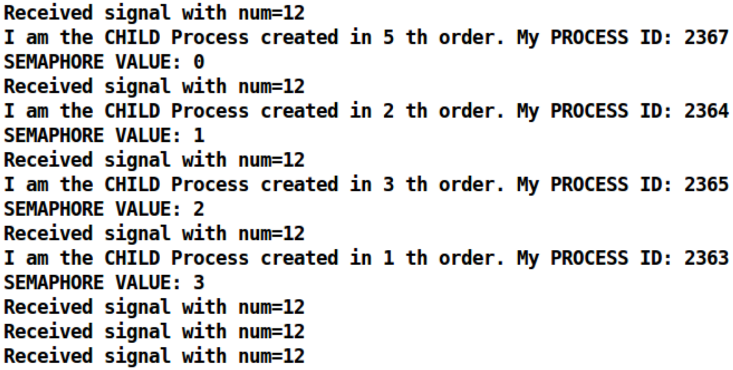
```c
    // parent process
    if(f>0){
        // creating a semaphore with key=SEMKEY
        sem_id = semget(SEMKEY, 1, 0700|IPC_CREAT);
        // setting value of the 0th semaphore of the set identified with sem_id to 0
        semctl(sem_id, 0, SETVAL, 0);
        // waiting for a second
        sleep(1);
        // sending the signal 12 to all child processes
        for (i=0; i<10; i++)
            kill(children[i], 12);
        // decrease semaphore value by 10 (i.e., wait for all childs to increase semaphore value)
        sem_wait(sem_id, 10);
        printf("ALL CHILDREN HAS Finished ...\n");
        // remove the semaphore set identified with sem_id
        semctl(sem_id, 0, IPC_RMID, 0);
        exit(0);
    }
```

Example 1



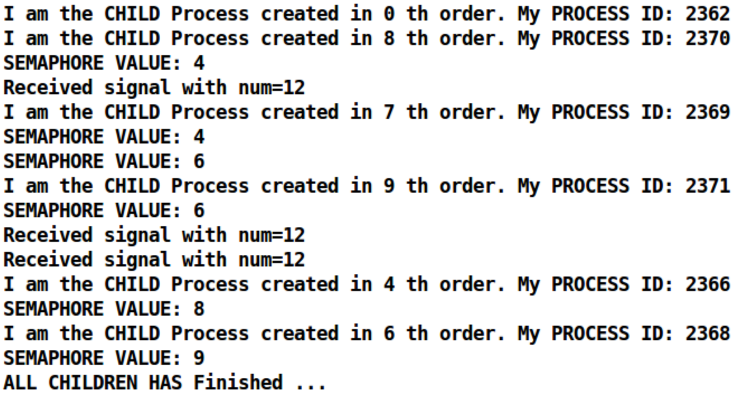
```c
    // child process
    else{
        // wait for a signal
        pause();
        // returning the sem_id associated with SEMKEY
        sem_id = semget(SEMKEY, 1, 0);
        printf("I am the CHILD Process created in %d th order. My PROCESS ID: %d\n", i, getpid());
        // getting value of the 0th semaphore of the set identified with sem_id
        printf("SEMAPHORE VALUE: %d\n",semctl(sem_id,0,GETVAL,0));
        // increase semaphore value by 1
        sem_signal(sem_id, 1);
    }

    return 0;
}
```

Output of Example 1



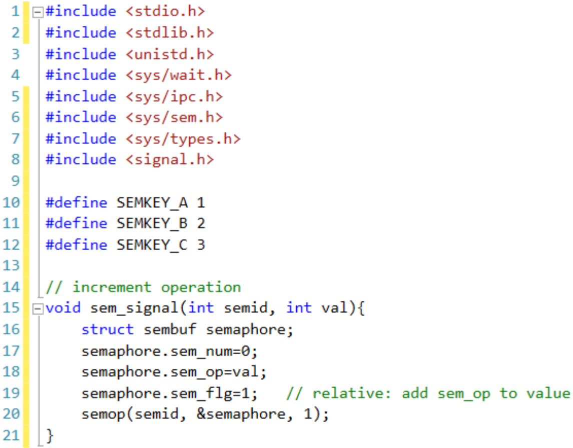
```
Received signal with num=12
I am the CHILD Process created in 5 th order. My PROCESS ID: 2367
SEMAPHORE VALUE: 0
Received signal with num=12
I am the CHILD Process created in 2 th order. My PROCESS ID: 2364
SEMAPHORE VALUE: 1
Received signal with num=12
I am the CHILD Process created in 3 th order. My PROCESS ID: 2365
SEMAPHORE VALUE: 2
Received signal with num=12
I am the CHILD Process created in 1 th order. My PROCESS ID: 2363
SEMAPHORE VALUE: 3
Received signal with num=12
Received signal with num=12
Received signal with num=12
```

Output of Example 1 (Continues)



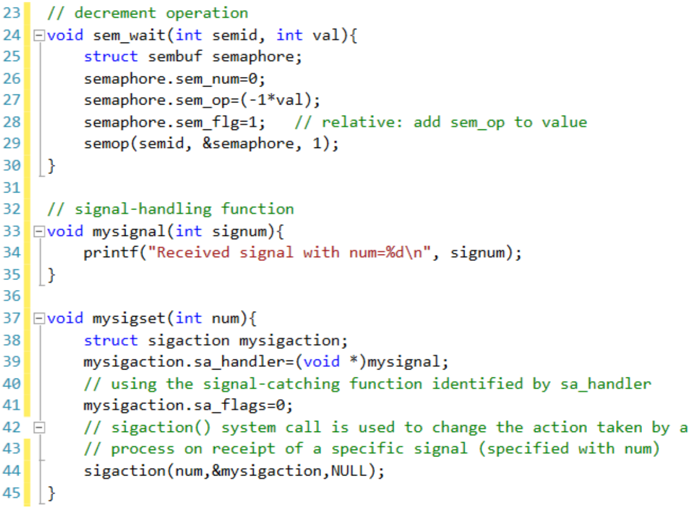
```
I am the CHILD Process created in 0 th order. My PROCESS ID: 2362
I am the CHILD Process created in 8 th order. My PROCESS ID: 2370
SEMAPHORE VALUE: 4
Received signal with num=12
I am the CHILD Process created in 7 th order. My PROCESS ID: 2369
SEMAPHORE VALUE: 4
SEMAPHORE VALUE: 6
I am the CHILD Process created in 9 th order. My PROCESS ID: 2371
SEMAPHORE VALUE: 6
Received signal with num=12
Received signal with num=12
I am the CHILD Process created in 4 th order. My PROCESS ID: 2366
SEMAPHORE VALUE: 8
I am the CHILD Process created in 6 th order. My PROCESS ID: 2368
SEMAPHORE VALUE: 9
ALL CHILDREN HAS Finished ...
```

Example 2 - Deadlock



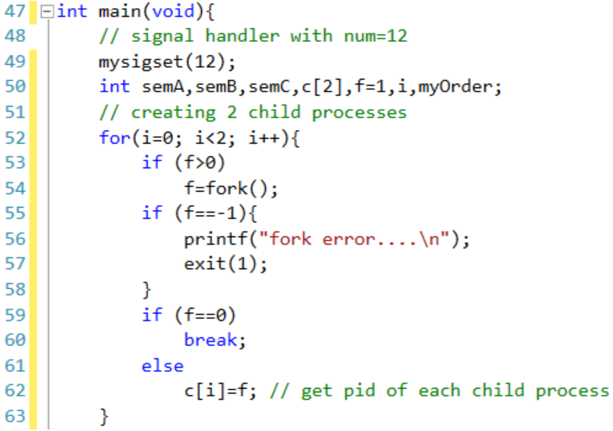
```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <sys/types.h>
#include <signal.h>

#define SEMKEY_A 1
#define SEMKEY_B 2
#define SEMKEY_C 3

// increment operation
void sem_signal(int semid, int val){
    struct sembuf semaphore;
    semaphore.sem_num=0;
    semaphore.sem_op=val;
    semaphore.sem_flg=1;   // relative: add sem_op to value
    semop(semid, &semaphore, 1);
}
```

Example 2 - Deadlock



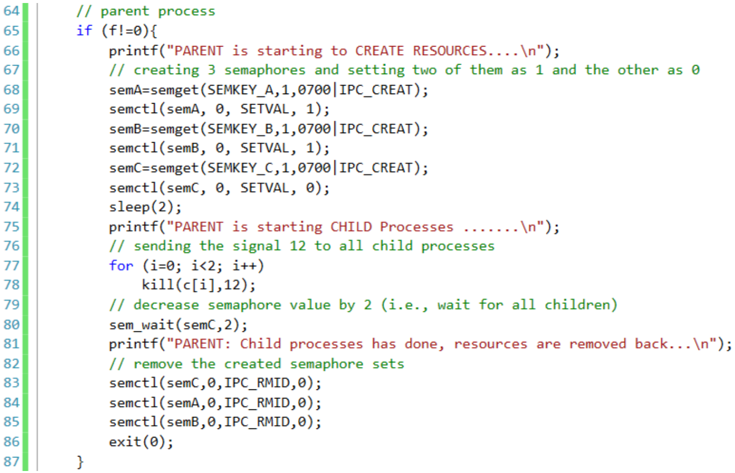
```
// decrement operation
void sem_wait(int semid, int val){
    struct sembuf semaphore;
    semaphore.sem_num=0;
    semaphore.sem_op=(-1*val);
    semaphore.sem_flg=1;    // relative: add sem_op to value
    semop(semid, &semaphore, 1);
}

// signal-handling function
void mysignal(int signum){
    printf("Received signal with num=%d\n", signum);
}

void mysigset(int num){
    struct sigaction mysigaction;
    mysigaction.sa_handler=(void *)mysignal;
    // using the signal-catching function identified by sa_handler
    mysigaction.sa_flags=0;
    // sigaction() system call is used to change the action taken by a
    // process on receipt of a specific signal (specified with num)
    sigaction(num,&mysigaction,NULL);
}
```

Example 2 - Deadlock



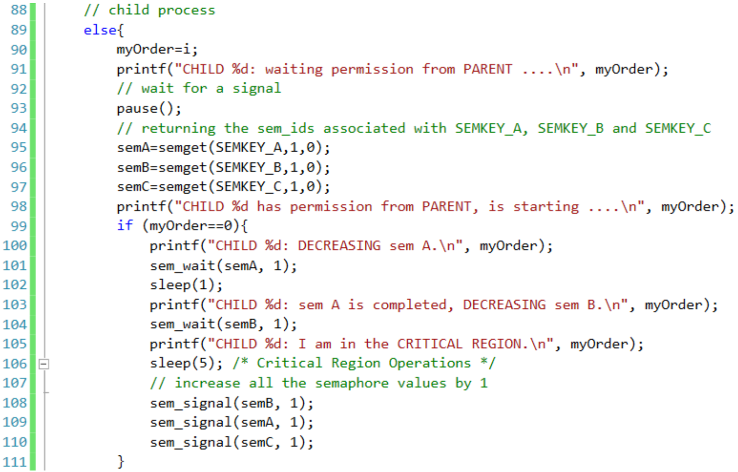
```c
int main(void){
    // signal handler with num=12
    mysigset(12);
    int semA,semB,semC,c[2],f=1,i,myOrder;
    // creating 2 child processes
    for(i=0; i<2; i++){
        if (f>0)
            f=fork();
        if (f==-1){
            printf("fork error....\n");
            exit(1);
        }
        if (f==0)
            break;
        else
            c[i]=f; // get pid of each child process
    }
```

Example 2 - Deadlock



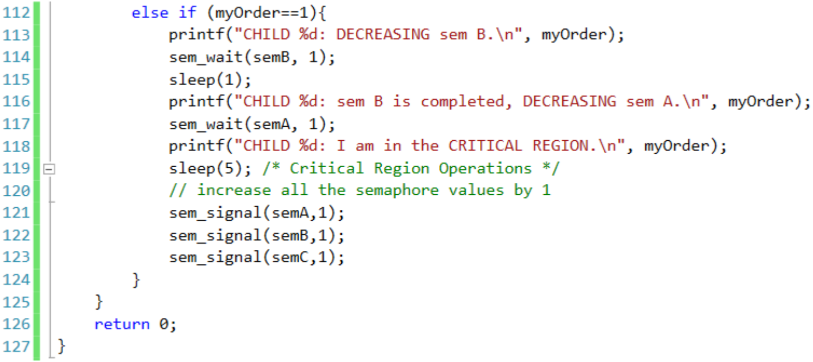
```
// parent process
if (f!=0){
    printf("PARENT is starting to CREATE RESOURCES....\n");
    // creating 3 semaphores and setting two of them as 1 and the other as 0
    semA=semget(SEMKEY_A,1,0700|IPC_CREAT);
    semctl(semA, 0, SETVAL, 1);
    semB=semget(SEMKEY_B,1,0700|IPC_CREAT);
    semctl(semB, 0, SETVAL, 1);
    semC=semget(SEMKEY_C,1,0700|IPC_CREAT);
    semctl(semC, 0, SETVAL, 0);
    sleep(2);
    printf("PARENT is starting CHILD Processes .......\n");
    // sending the signal 12 to all child processes
    for (i=0; i<2; i++)
        kill(c[i],12);
    // decrease semaphore value by 2 (i.e., wait for all children)
    sem_wait(semC,2);
    printf("PARENT: Child processes has done, resources are removed back...\n");
    // remove the created semaphore sets
    semctl(semC,0,IPC_RMID,0);
    semctl(semA,0,IPC_RMID,0);
    semctl(semB,0,IPC_RMID,0);
    exit(0);
}
```

Example 2 - Deadlock



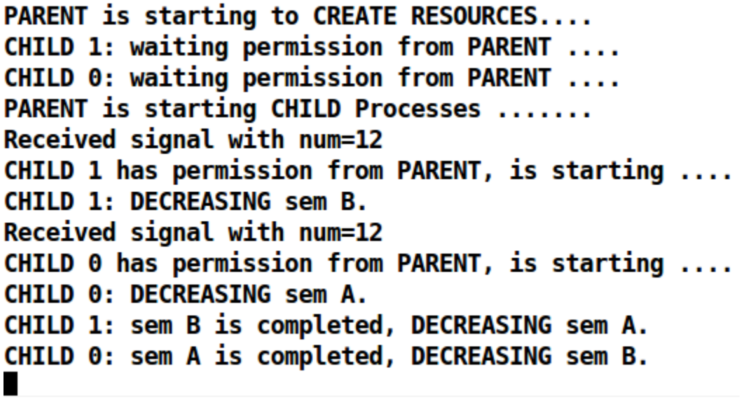
```
        // child process
        else{
            myOrder=i;
            printf("CHILD %d: waiting permission from PARENT ....\n", myOrder);
            // wait for a signal
            pause();
            // returning the sem_ids associated with SEMKEY_A, SEMKEY_B and SEMKEY_C
            semA=semget(SEMKEY_A,1,0);
            semB=semget(SEMKEY_B,1,0);
            semC=semget(SEMKEY_C,1,0);
            printf("CHILD %d has permission from PARENT, is starting ....\n", myOrder);
            if (myOrder==0){
                printf("CHILD %d: DECREASING sem A.\n", myOrder);
                sem_wait(semA, 1);
                sleep(1);
                printf("CHILD %d: sem A is completed, DECREASING sem B.\n", myOrder);
                sem_wait(semB, 1);
                printf("CHILD %d: I am in the CRITICAL REGION.\n", myOrder);
                sleep(5); /* Critical Region Operations */
                // increase all the semaphore values by 1
                sem_signal(semB, 1);
                sem_signal(semA, 1);
                sem_signal(semC, 1);
            }
```

Example 2 - Deadlock



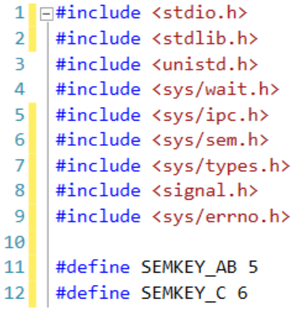
```
    else if (myOrder==1){
      printf("CHILD %d: DECREASING sem B.\n", myOrder);
      sem_wait(semB, 1);
      sleep(1);
      printf("CHILD %d: sem B is completed, DECREASING sem A.\n", myOrder);
      sem_wait(semA, 1);
      printf("CHILD %d: I am in the CRITICAL REGION.\n", myOrder);
      sleep(5); /* Critical Region Operations */
      // increase all the semaphore values by 1
      sem_signal(semA,1);
      sem_signal(semB,1);
      sem_signal(semC,1);
    }
  }
  return 0;
}
```

Output of Example 2



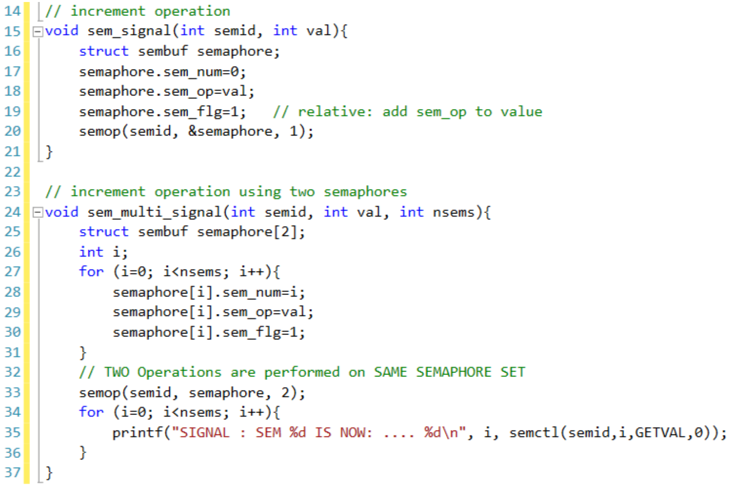
```
PARENT is starting to CREATE RESOURCES....
CHILD 1: waiting permission from PARENT ....
CHILD 0: waiting permission from PARENT ....
PARENT is starting CHILD Processes .......
Received signal with num=12
CHILD 1 has permission from PARENT, is starting ....
CHILD 1: DECREASING sem B.
Received signal with num=12
CHILD 0 has permission from PARENT, is starting ....
CHILD 0: DECREASING sem A.
CHILD 1: sem B is completed, DECREASING sem A.
CHILD 0: sem A is completed, DECREASING sem B.
```

Example 3 - Preventing Deadlock



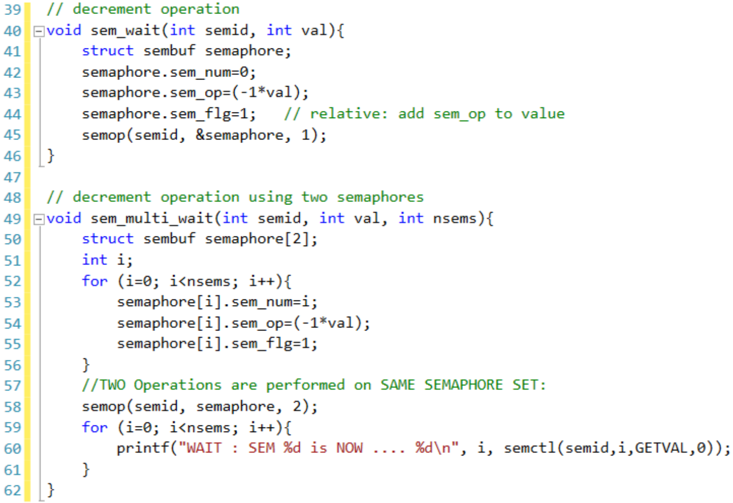
```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <sys/types.h>
#include <signal.h>
#include <sys/errno.h>

#define SEMKEY_AB 5
#define SEMKEY_C 6
```

Preventing Deadlock



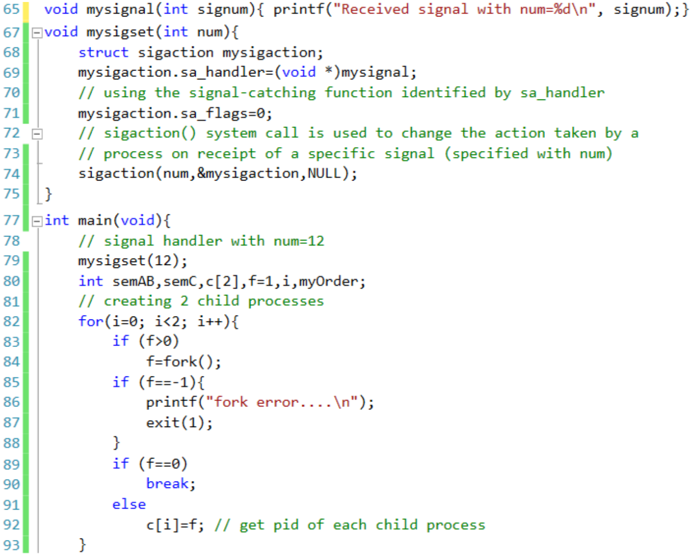
```
// increment operation
void sem_signal(int semid, int val){
    struct sembuf semaphore;
    semaphore.sem_num=0;
    semaphore.sem_op=val;
    semaphore.sem_flg=1;   // relative: add sem_op to value
    semop(semid, &semaphore, 1);
}

// increment operation using two semaphores
void sem_multi_signal(int semid, int val, int nsems){
    struct sembuf semaphore[2];
    int i;
    for (i=0; i<nsems; i++){
        semaphore[i].sem_num=i;
        semaphore[i].sem_op=val;
        semaphore[i].sem_flg=1;
    }
    // TWO Operations are performed on SAME SEMAPHORE SET
    semop(semid, semaphore, 2);
    for (i=0; i<nsems; i++){
        printf("SIGNAL : SEM %d IS NOW: .... %d\n", i, semctl(semid,i,GETVAL,0));
    }
}
```

Preventing Deadlock



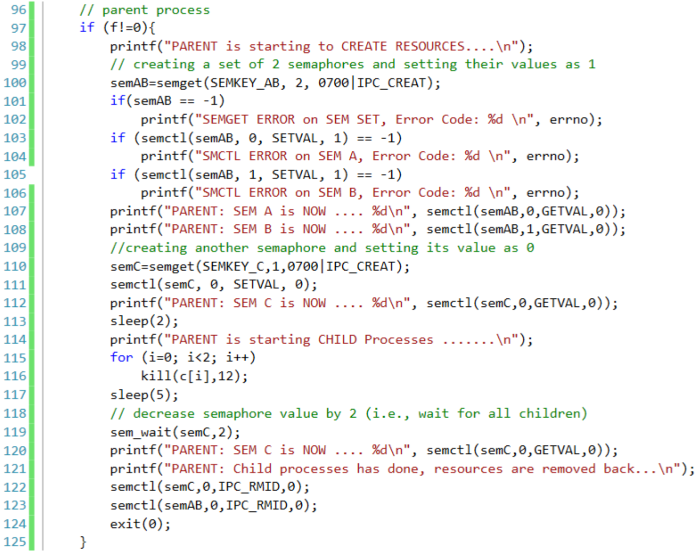
```
// decrement operation
void sem_wait(int semid, int val){
    struct sembuf semaphore;
    semaphore.sem_num=0;
    semaphore.sem_op=(-1*val);
    semaphore.sem_flg=1;   // relative: add sem_op to value
    semop(semid, &semaphore, 1);
}

// decrement operation using two semaphores
void sem_multi_wait(int semid, int val, int nsems){
    struct sembuf semaphore[2];
    int i;
    for (i=0; i<nsems; i++){
        semaphore[i].sem_num=i;
        semaphore[i].sem_op=(-1*val);
        semaphore[i].sem_flg=1;
    }
    //TWO Operations are performed on SAME SEMAPHORE SET:
    semop(semid, semaphore, 2);
    for (i=0; i<nsems; i++){
        printf("WAIT : SEM %d is NOW .... %d\n", i, semctl(semid,i,GETVAL,0));
    }
}
```

Preventing Deadlock



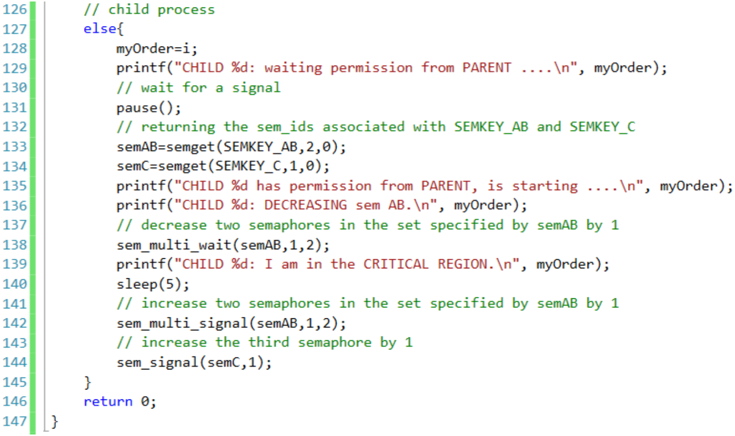
```c
void mysignal(int signum){ printf("Received signal with num=%d\n", signum);}
void mysigset(int num){
    struct sigaction mysigaction;
    mysigaction.sa_handler=(void *)mysignal;
    // using the signal-catching function identified by sa_handler
    mysigaction.sa_flags=0;
    // sigaction() system call is used to change the action taken by a
    // process on receipt of a specific signal (specified with num)
    sigaction(num,&mysigaction,NULL);
}
int main(void){
    // signal handler with num=12
    mysigset(12);
    int semAB,semC,c[2],f=1,i,myOrder;
    // creating 2 child processes
    for(i=0; i<2; i++){
        if (f>0)
            f=fork();
        if (f==-1){
            printf("fork error....\n");
            exit(1);
        }
        if (f==0)
            break;
        else
            c[i]=f; // get pid of each child process
    }
```

Preventing Deadlock



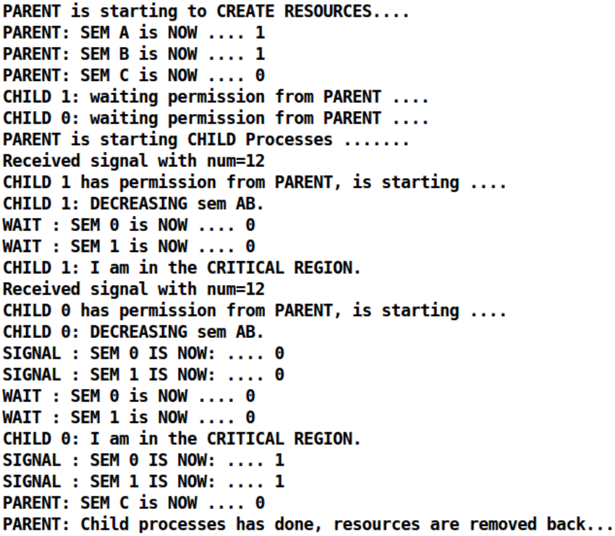
```
// parent process
if (f!=0){
    printf("PARENT is starting to CREATE RESOURCES....\n");
    // creating a set of 2 semaphores and setting their values as 1
    semAB=semget(SEMKEY_AB, 2, 0700|IPC_CREAT);
    if(semAB == -1)
        printf("SEMGET ERROR on SEM SET, Error Code: %d \n", errno);
    if (semctl(semAB, 0, SETVAL, 1) == -1)
        printf("SMCTL ERROR on SEM A, Error Code: %d \n", errno);
    if (semctl(semAB, 1, SETVAL, 1) == -1)
        printf("SMCTL ERROR on SEM B, Error Code: %d \n", errno);
    printf("PARENT: SEM A is NOW .... %d\n", semctl(semAB,0,GETVAL,0));
    printf("PARENT: SEM B is NOW .... %d\n", semctl(semAB,1,GETVAL,0));
    //creating another semaphore and setting its value as 0
    semC=semget(SEMKEY_C,1,0700|IPC_CREAT);
    semctl(semC, 0, SETVAL, 0);
    printf("PARENT: SEM C is NOW .... %d\n", semctl(semC,0,GETVAL,0));
    sleep(2);
    printf("PARENT is starting CHILD Processes .......\n");
    for (i=0; i<2; i++)
        kill(c[i],12);
    sleep(5);
    // decrease semaphore value by 2 (i.e., wait for all children)
    sem_wait(semC,2);
    printf("PARENT: SEM C is NOW .... %d\n", semctl(semC,0,GETVAL,0));
    printf("PARENT: Child processes has done, resources are removed back...\n");
    semctl(semC,0,IPC_RMID,0);
    semctl(semAB,0,IPC_RMID,0);
    exit(0);
}
```

Preventing Deadlock



```
    // child process
    else{
        myOrder=i;
        printf("CHILD %d: waiting permission from PARENT ....\n", myOrder);
        // wait for a signal
        pause();
        // returning the sem_ids associated with SEMKEY_AB and SEMKEY_C
        semAB=semget(SEMKEY_AB,2,0);
        semC=semget(SEMKEY_C,1,0);
        printf("CHILD %d has permission from PARENT, is starting ....\n", myOrder);
        printf("CHILD %d: DECREASING sem AB.\n", myOrder);
        // decrease two semaphores in the set specified by semAB by 1
        sem_multi_wait(semAB,1,2);
        printf("CHILD %d: I am in the CRITICAL REGION.\n", myOrder);
        sleep(5);
        // increase two semaphores in the set specified by semAB by 1
        sem_multi_signal(semAB,1,2);
        // increase the third semaphore by 1
        sem_signal(semC,1);
    }
    return 0;
}
```

Output of The Example 3



```
PARENT is starting to CREATE RESOURCES....
PARENT: SEM A is NOW .... 1
PARENT: SEM B is NOW .... 1
PARENT: SEM C is NOW .... 0
CHILD 1: waiting permission from PARENT ....
CHILD 0: waiting permission from PARENT ....
PARENT is starting CHILD Processes .......
Received signal with num=12
CHILD 1 has permission from PARENT, is starting ....
CHILD 1: DECREASING sem AB.
WAIT : SEM 0 is NOW .... 0
WAIT : SEM 1 is NOW .... 0
CHILD 1: I am in the CRITICAL REGION.
Received signal with num=12
CHILD 0 has permission from PARENT, is starting ....
CHILD 0: DECREASING sem AB.
SIGNAL : SEM 0 IS NOW: .... 0
SIGNAL : SEM 1 IS NOW: .... 0
WAIT : SEM 0 is NOW .... 0
WAIT : SEM 1 is NOW .... 0
CHILD 0: I am in the CRITICAL REGION.
SIGNAL : SEM 0 IS NOW: .... 1
SIGNAL : SEM 1 IS NOW: .... 1
PARENT: SEM C is NOW .... 0
PARENT: Child processes has done, resources are removed back...
```