**Name:Rohan Nivas Chavan**

**PRN:2019BTEIT00084**

**UOS LAB**

**6. IPC:Semaphores**

**6.3** Write two programs that will communicate both ways (i.e each process can read and write) when run concurrently via semaphores.

**Objectives:**

1. To learn about IPC through semaphore.
2. Use of system call and IPC mechanism to write effective application programs.

**Theory:**

A semaphore is a resource that contains an integer value, and allows processes to synchronize by testing and setting this value in a single atomic operation. This means that the process that tests the value of a semaphore and sets it to a different value (based on the test), is guaranteed no other process will interfere with the operation in the middle.

Two types of operations can be carried on a semaphore: wait and signal. A set operation first checks if the semaphore's value equals some number. If it does, it decreases its value and returns. If it does not, the operation blocks the calling process until the semaphore's value reaches the desired value. A signal operation increments the value of the semaphore, possibly awakening one or more processes that are waiting on the semaphore. How this mechanism can be put to practical use will be exlained later.

A semaphore set is a structure that stores a group of semaphores together, and possibly allows the process to commit a transaction on part or all of the semaphores in the set together. Here, a transaction means that we are guaranteed that either all operations are done successfully, or none is done at all. Note that a semaphore set is not a general parallel programming concept, it's just an extra mechanism supplied by SysV IPC.

**Data Dictionary:**

1 KEY Long int External identifier for the program

2 id int Number by which semaphore is known within a program

3 argument Union semun

To pass arguments to the semctl function

4 retval int Store return value of semop function

**Program:**

#include<stdio.h>

#include<sys/ipc.h>

#include<sys/shm.h>

#include<sys/types.h>

#include<string.h>

#include<errno.h>

#include<stdlib.h>

#include<unistd.h>

#include<string.h>

#define SHM\_KEY 0x12345

struct shmseg {

int cntr;

int write\_complete;

int read\_complete;

};

void shared\_memory\_cntr\_increment(int pid, struct shmseg \*shmp, int total\_count);

int main(int argc, char \*argv[]) {

int shmid;

struct shmseg \*shmp;

char \*bufptr;

int total\_count;

int sleep\_time;

pid\_t pid;

if (argc != 2)

total\_count = 10000;

else {

total\_count = atoi(argv[1]);

if (total\_count < 10000)

total\_count = 10000;

}

printf("Total Count is %d\n", total\_count);

shmid = shmget(SHM\_KEY, sizeof(struct shmseg), 0644|IPC\_CREAT);

if (shmid == -1) {

perror("Shared memory");

return 1;

}

// Attach to the segment to get a pointer to it.

shmp = shmat(shmid, NULL, 0);

if (shmp == (void \*) -1) {

perror("Shared memory attach");

return 1;

}

shmp->cntr = 0;

pid = fork();

/\* Parent Process - Writing Once \*/

if (pid > 0) {

shared\_memory\_cntr\_increment(pid, shmp, total\_count);

} else if (pid == 0) {

shared\_memory\_cntr\_increment(pid, shmp, total\_count);

return 0;

} else {

perror("Fork Failure\n");

return 1;

}

while (shmp->read\_complete != 1)

sleep(1);

if (shmdt(shmp) == -1) {

perror("shmdt");

return 1;

}

if (shmctl(shmid, IPC\_RMID, 0) == -1) {

perror("shmctl");

return 1;

}

printf("Writing Process: Complete\n");

return 0;

}

/\* Increment the counter of shared memory by total\_count in steps of 1 \*/

void shared\_memory\_cntr\_increment(int pid, struct shmseg \*shmp, int total\_count) {

int cntr;

int numtimes;

int sleep\_time;

cntr = shmp->cntr;

shmp->write\_complete = 0;

if (pid == 0)

printf("SHM\_WRITE: CHILD: Now writing\n");

else if (pid > 0)

printf("SHM\_WRITE: PARENT: Now writing\n");

//printf("SHM\_CNTR is %d\n", shmp->cntr);

/\* Increment the counter in shared memory by total\_count in steps of 1 \*/

for (numtimes = 0; numtimes < total\_count; numtimes++) {

cntr += 1;

shmp->cntr = cntr;

/\* Sleeping for a second for every thousand \*/

sleep\_time = cntr % 1000;

if (sleep\_time == 0)

sleep(1);

}

shmp->write\_complete = 1;

if (pid == 0)

printf("SHM\_WRITE: CHILD: Writing Done\n");

else if (pid > 0)

printf("SHM\_WRITE: PARENT: Writing Done\n");

return;

}

## Compilation and Execution Steps

Reading process: shared memory: counter is 11000

Reading process: Reading done, Detaching shared memory

Reading Process: Complete

#include<stdio.h>

#include<sys/types.h>

#include<sys/ipc.h>

#include<sys/shm.h>

#include<sys/sem.h>

#include<string.h>

#include<errno.h>

#include<stdlib.h>

#include<unistd.h>

#include<string.h>

#define SHM\_KEY 0x12345

#define SEM\_KEY 0x54321

#define MAX\_TRIES 20

struct shmseg {

int cntr;

int write\_complete;

int read\_complete;

};

void shared\_memory\_cntr\_increment(int, struct shmseg\*, int);

void remove\_semaphore();

int main(int argc, char \*argv[]) {

int shmid;

struct shmseg \*shmp;

char \*bufptr;

int total\_count;

int sleep\_time;

pid\_t pid;

if (argc != 2)

total\_count = 10000;

else {

total\_count = atoi(argv[1]);

if (total\_count < 10000)

total\_count = 10000;

}

printf("Total Count is %d\n", total\_count);

shmid = shmget(SHM\_KEY, sizeof(struct shmseg), 0644|IPC\_CREAT);

if (shmid == -1) {

perror("Shared memory");

return 1;

}

// Attach to the segment to get a pointer to it.

shmp = shmat(shmid, NULL, 0);

if (shmp == (void \*) -1) {

perror("Shared memory attach: ");

return 1;

}

shmp->cntr = 0;

pid = fork();

/\* Parent Process - Writing Once \*/

if (pid > 0) {

shared\_memory\_cntr\_increment(pid, shmp, total\_count);

} else if (pid == 0) {

shared\_memory\_cntr\_increment(pid, shmp, total\_count);

return 0;

} else {

perror("Fork Failure\n");

return 1;

}

while (shmp->read\_complete != 1)

sleep(1);

if (shmdt(shmp) == -1) {

perror("shmdt");

return 1;

}

if (shmctl(shmid, IPC\_RMID, 0) == -1) {

perror("shmctl");

return 1;

}

printf("Writing Process: Complete\n");

remove\_semaphore();

return 0;

}

/\* Increment the counter of shared memory by total\_count in steps of 1 \*/

void shared\_memory\_cntr\_increment(int pid, struct shmseg \*shmp, int total\_count) {

int cntr;

int numtimes;

int sleep\_time;

int semid;

struct sembuf sem\_buf;

struct semid\_ds buf;

int tries;

int retval;

semid = semget(SEM\_KEY, 1, IPC\_CREAT | IPC\_EXCL | 0666);

//printf("errno is %d and semid is %d\n", errno, semid);

/\* Got the semaphore \*/

if (semid >= 0) {

printf("First Process\n");

sem\_buf.sem\_op = 1;

sem\_buf.sem\_flg = 0;

sem\_buf.sem\_num = 0;

retval = semop(semid, &sem\_buf, 1);

if (retval == -1) {

perror("Semaphore Operation: ");

return;

}

} else if (errno == EEXIST) { // Already other process got it

int ready = 0;

printf("Second Process\n");

semid = semget(SEM\_KEY, 1, 0);

if (semid < 0) {

perror("Semaphore GET: ");

return;

}

/\* Waiting for the resource \*/

sem\_buf.sem\_num = 0;

sem\_buf.sem\_op = 0;

sem\_buf.sem\_flg = SEM\_UNDO;

retval = semop(semid, &sem\_buf, 1);

if (retval == -1) {

perror("Semaphore Locked: ");

return;

}

}

sem\_buf.sem\_num = 0;

sem\_buf.sem\_op = -1; /\* Allocating the resources \*/

sem\_buf.sem\_flg = SEM\_UNDO;

retval = semop(semid, &sem\_buf, 1);

if (retval == -1) {

perror("Semaphore Locked: ");

return;

}

cntr = shmp->cntr;

shmp->write\_complete = 0;

if (pid == 0)

printf("SHM\_WRITE: CHILD: Now writing\n");

else if (pid > 0)

printf("SHM\_WRITE: PARENT: Now writing\n");

//printf("SHM\_CNTR is %d\n", shmp->cntr);

/\* Increment the counter in shared memory by total\_count in steps of 1 \*/

for (numtimes = 0; numtimes < total\_count; numtimes++) {

cntr += 1;

shmp->cntr = cntr;

/\* Sleeping for a second for every thousand \*/

sleep\_time = cntr % 1000;

if (sleep\_time == 0)

sleep(1);

}

shmp->write\_complete = 1;

sem\_buf.sem\_op = 1; /\* Releasing the resource \*/

retval = semop(semid, &sem\_buf, 1);

if (retval == -1) {

perror("Semaphore Locked\n");

return;

}

if (pid == 0)

printf("SHM\_WRITE: CHILD: Writing Done\n");

else if (pid > 0)

printf("SHM\_WRITE: PARENT: Writing Done\n");

return;

}

void remove\_semaphore() {

int semid;

int retval;

semid = semget(SEM\_KEY, 1, 0);

if (semid < 0) {

perror("Remove Semaphore: Semaphore GET: ");

return;

}

retval = semctl(semid, 0, IPC\_RMID);

if (retval == -1) {

perror("Remove Semaphore: Semaphore CTL: ");

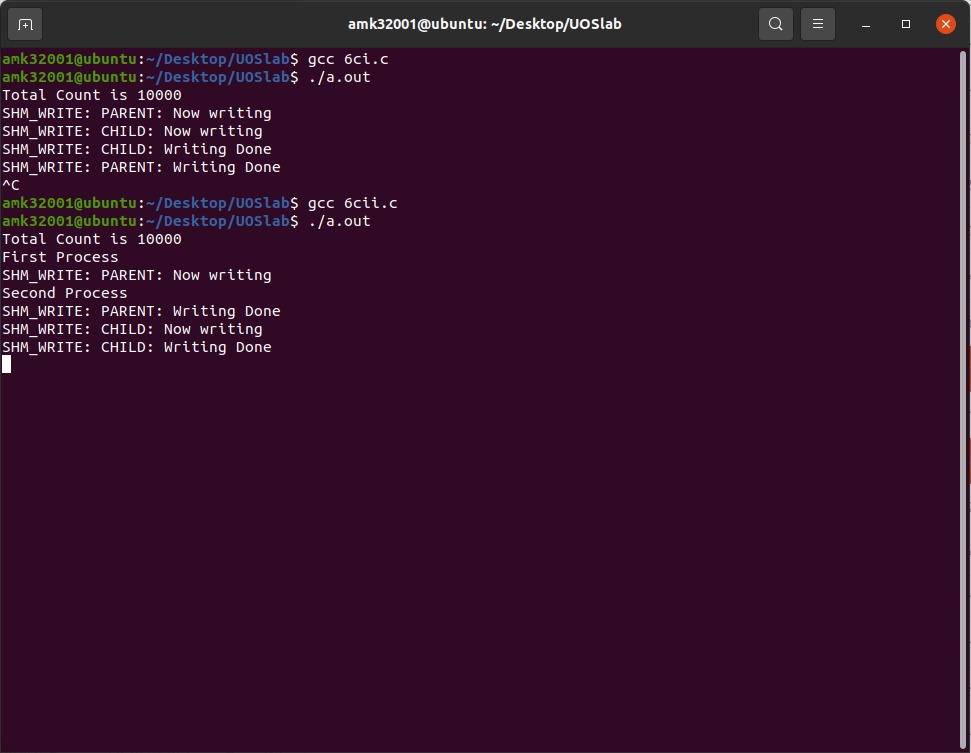
return;

}

return;

}

## Compilation and Execution Steps



## Execution Steps

Reading Process: Shared Memory: Counter is 20000

Reading Process: Reading Done, Detaching Shared Memory

Reading Process: Complete

**Conclusion:**

1. Study about IPC through semaphore.

2. Study of system call and IPC mechanism to write effective application programs.

**References:**

Dave’s Programming in C Tutorials