LAB REPORT

Name: Anuran Chakraborty
Roll no.: 20 Class: UG-III Section: A1

Question:

Create child processes: X and Y.

a. Each child process performs 10 iterations. The child process just displays its name/id and the current iteration number, and sleeps for some random amount of time. Adjust the sleeping duration of the processes to have a different output (i.e. another interleaving of processes' traces).

- **void iterate(int time)**

  This function iterates 10 times printing the iteration number, the process id and then sleeping for a given amount of time which is passed as a parameter. The function used for sleeping is ***sleep(time)***. The process id is fetvhed by using the ***getpid()*** function.

```
void iterate(int time)
{
        int i;
        pid_t procid=getpid();
        for(i=1;i<=10;i++)
        {
                //Print process id and iteration number
                printf("ProcessId: %d Iteration: %d\n",procid,i);
                sleep(time);
        }
}
```

- **void processCreate()**

  This function is used to create two processes and then call the ***iterate(int)*** function from both using different delay. ***fork()*** system call is used to create the processes. The fork() system call returns -1 in case of fork failure, 0 to child process, positive integer to child process. Here in order to create two child process the fork() system call has been use twice. From within each of the child process blocks the iterate function is called using different parameters. The parent is then kept waiting till child process completion using ***wait(NULL)*** which resides in <sys/wait.h>.

```
void processCreate()
```

```
{
        pid_t pid=fork();

        if(pid==-1)
                printf("Fork failure\n");
        else
        if(pid==0)
        {
                //Child process X
                printf("Child X created\n");
                iterate(1);
        }
        else
        {
                pid_t pid2=fork();

                if(pid2==-1)
                printf("Fork failure\n");
                else
                if(pid2==0)
                {
                        //Child process Y
                        printf("Child Y created\n");
                        iterate(2);
                }
                else
                {
                        wait(NULL);
                }
                wait(NULL);
                return;
        }
}
```

- **int main()**

  This is the main function which calls ***processCreate().***

```
int main()
{
        processCreate();
        return 0;
}
```

b. Modify the program so that X is not allowed to start iteration i before process Y has terminated its own iteration i-1. Use semaphore to implement this synchronization.

We have initialized two semaphores mutexX, mutexY in the beginning to be used by the respective processes. The type ***sem_t*** resides in **<*semaphore.h*>.**

```
sem_t *mutexX;
sem_t *mutexY;
```

- **void iterate(int time, int flag)**

  This function will iterate from 1 to 10 printing the iteration numbers and process id of the process calling it, and also sleeping for a specific amount of time. Here it is also ensured that the above condition of X printing i after Y prints i-1 has been satisfied using semaphore. The variable flag determines which process is calling the function. If X calls the function ***flag=0*** else ***flag=1.***
  In case the function is called by process X then it waits on mutexX, prints it and signals mutexY. Therefore the next iteration X must wait as it has not signaled mutexX, and Y can easily enter the critical section. Again Y waits on mutexY and signals mutexX, so next time X can enter critical section but not Y.
  The functions ***sem_wait(sem_t*)*** is the wait function, ***sem_post(sem_t*)*** is the signal function.

```
void iterate(int time,int flag)
{
        int i;
        pid_t procid=getpid();



        for(i=1;i<=10;i++)
        {

                if(flag==0) //If process is X
                {
                        sem_wait(mutexX);
                }
                else //If process is Y
                {
                        sem_wait(mutexY);
                }

                //Print process id and iteration number
                printf("Process Name: %c ProcessId: %d Iteration:
%d\n",(char)(flag+'X'),procid,i);
```

```
                sleep(time);

                if(flag==0) //If process is X
                {
                        sem_post(mutexY);
                }
                else //If process is Y
                {
                        sem_post(mutexX);
                }

        }
}
```

- **void processCreate()**

  This function is used to create two processes and then call the ***iterate(int)*** function from both using different delay. ***fork()*** system call is used to create the processes. The fork() system call returns -1 in case of fork failure, 0 to child process, positive integer to child process. Here in order to create two child process the **fork()** system call has been use twice. From within each of the child process blocks the iterate function is called using different parameters. The parent is then kept waiting till child process completion using ***wait(NULL)*** which resides in **<sys/wait.h>**. The first child i.e. X calls iterate by passing 0 as the flag parameter while y calls it by passing 1 as the flag.

```
void processCreate()
{
        pid_t pid=fork();

        if(pid==-1)
                printf("Fork failure\n");
        else
        if(pid==0)
        {
                //Child process X
                printf("Child X created\n");
                iterate(1,0);
        }
        else
        {
                pid_t pid2=fork();

                if(pid2==-1)
                printf("Fork failure\n");
                else
                if(pid2==0)
```

```
                {
                        //Child process Y
                        printf("Child Y created\n");
                        iterate(2,1);
                }
                else
                {
                        wait(NULL);
                }
                wait(NULL);
                return;
        }
}
```

- **int main()**

  This is the main function. Here first the semaphores the unlinked if they already exist in memory. As the semaphores are to be used within processes the semaphores must either be kept in shared memory or named semaphores can be used. Here named semaphores has been used. ***sem_open (char* semname ,int mode, int permission, int val)*** is used to create named semaphores. The mode consists of constant values which resides in ***<sys/stat.h>.*** Here **O_CREAT** mode has been used which will create a new named semaphore if it does not already exist. **O_EXCL** together with **O_CREAT** throws an error if it already exists. The permissions are given as unix permission codes, the value is the initial value of the semaphore. **mutexX** has value 1 while **mutexY** has value 0. In case of semaphore creation failure ***sem_open()*** returns **SEM_FAILED**.
  Then the ***processCreate()*** function is called.

```
int main()
{
        //Initialize the semaphore
        sem_unlink(mx);
        sem_unlink(my);

        mutexX = sem_open(mx, O_CREAT | O_EXCL ,0777, 1);

        if (mutexX == SEM_FAILED) {
     perror("Failed to open semphore for empty");
     exit(-1);
        }

        mutexY = sem_open(my, O_CREAT | O_EXCL ,0777, 0);

        if (mutexY == SEM_FAILED) {
```

```
     perror("Failed to open semphore for empty");
     exit(-1);
        }

        processCreate();
        return 0;
}
```