**Programming Project: Interprocess Communication with Semaphores and Shared Memory**

Mark Tan

11 December 2017

CECS 326 MW

damtan1996@gmail.com

This program spawns four child processes in which each generates a random integer and checks if it is a factor of either a U value or a V value. At most, one process is responsible for comparing the random integer to a reference value, and processes must wait in case both references are used. If the random integer is less than 100 or is a factor of the reference value, it will conclude and allow the next child process to be active. When a process is waiting to use a value, it is added to the waiting queue and will only be removed once it is resumed. By default, the newly resumed process checks the U value first.

This current iteration of the program does not exhibit deadlock because there is always at least one process making progress. Although there is mutual exclusive use of resources due to how I initialize my semaphore variables (both are initialized at 1), there is no scenario where circular wait is apparent. In addition, this solution does not exhibit starvation because all processes are able to make progress at all times, even when some are in the waiting state.

#include <iostream>

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <limits.h>

#include <sys/types.h>

#include <sys/stat.h>

#include <sys/wait.h>

#include <sys/ipc.h>

#include <sys/shm.h>

#include <queue>

#include "semaphore.h"

using namespace std;

//factors

int U = 827395609;

int V = 962094883;

//semaphore names

enum {U\_proc, V\_proc};

/\*\*

\* Cleans up semaphore once child is finished

\*/

void parent\_cleanup (SEMAPHORE &sem, int shmid) {

wait(0);

shmctl(shmid, IPC\_RMID, NULL);

sem.remove();

}

/\*\*

\* Requests a user to finish the program and terminate parent and kids

\*/

void parent\_proc(string q, SEMAPHORE &sem, int shmid) {

cout << "Would you like to quit (type !wq to quit)? ";

cin >> q;

if(q == string("!wq")) {

parent\_cleanup(sem, shmid);

exit(0);

}

}

/\*\*

\* Generates a random int and checks if it is a factor of X or less than 100

\* where each process works on a certain X variable or waits

\*/

void child\_proc(queue<long> q, SEMAPHORE &sem) {

int rand\_num, X;

do{

rand\_num = rand();

cout << "CHILD PROCESS: " << getpid() << endl;

if(U\_proc > -1) {

//make child work on U

sem.P(U\_proc);

X = U;

sem.V(U\_proc);

if(q.front() == getpid())

q.pop();

} else if (V\_proc > -1) {

//make child work on V

sem.P(V\_proc);

X = V;

sem.V(V\_proc);

if(q.front() == getpid())

q.pop();

} else {

//make child wait, add to FIFO queue

q.push(getpid());

}

cout << rand\_num;

if(rand\_num % X != 0) {

cout << " is not a factor of " << X << endl;

} else {

cout << " is a factor of " << X << endl;

break;

}

}while(rand\_num >= 100);

q.push(getpid());

if(q.front() == getpid())

q.pop();

exit(0);

}

int main() {

long childPID;

string quit;

queue<long> proc\_order;

int shmid, status;

SEMAPHORE sem(2); //semaphores set to 2 because there are two semaphores

sem.V(U\_proc); //U\_proc = 1, functions as mutex

sem.V(V\_proc); //V\_proc = 1, functions as mutex

shmid = shmget(IPC\_PRIVATE, sizeof(char), PERMS);

//PARENT PROCESS

for(int i = 0; i < 4; i++) {

//generate child

cout << getpid() << " IS GENERATING A NEW CHILD \n" << endl;

childPID = fork();

//assign different tasks to child and parent

if(childPID == 0)

child\_proc(proc\_order, sem);

}

//this loop allows children to run concurrently

for(int i = 0; i < 4; i++) {

childPID = wait(&status);

parent\_proc(quit, sem, shmid);

}

exit(0);

}