Programming With Semaphores Cover Page

Names:  Eleazar Gomez, Anton Shirokov

E-mails:  [Eleazar.Gomez@student.csulb.edu](mailto:Eleazar.Gomez@student.csulb.edu), [anton.shirokov@student.csulb.edu](mailto:anton.shirokov@student.csulb.edu)

Date:  2 May 2019

Course Number:  CECS 326

Software Description:

This program deals with shared memory and semaphores. According to the instructions we are to allocate four 1536 byte shared memory segments, initialize them with random characters, and fork 5 child processes to perform swaps on these segments. The number of swaps each process performs is chosen by the user. A random 32-bit integer is generated to slow the processes down, and the sections of the segments to swap is also chosen at random. The swaps must be performed atomically (one process must complete its swap before another can start). To accomplish this, we use semaphores: decrementing before the swap to halt other processes and incrementing after the swap to resume other processes. The parent waits for the child processes to finish before destroying the memory it allocated.

Optional Critique:

The program specifications do not specify any output is necessary, so we chose output that would show to the best of our abilities that we satisfied the requirements. We output: the segments before any operations to show successful shared memory creation and initialization; the moment a process starts swapping and the moment a process stops swapping to show the atomic nature; the amount of operations a process completed to show no deadlock occurred; and finally the segments after all operations to show that swaps actually occurred.

As far as starvation goes, as seen when running our program all 5 processes complete the specified number of operations. This shows that there is no starvation, all processes were able to make progress until the end.

*// Eleazar Gomez, Anton Shirokov*

*// main.cpp*

*//*

*// This file contains the main logic of our program. The main function allocates*

*// the shared memory segments (4 in total), initializes them with characters,*

*// gets input on number of operations for each process to carry out, and forks*

*// 5 processes to perform swaps on the shared memory. The parent process waits*

*// for the processes to complete and deallocates the memory. The main function*

*// uses two other functions to carry out its purpose, one for getting user*

*// input, and one for each child process to execute. The output of the program*

*// consists of the groups of memory before any processes start, whenever a process*

*// starts or finishes a critical section, the number of operations a process*

*// completed, and the groups of memory after all processes finish. To make sure*

*// that only one process can be executing a critical section at a time, we included*

*// the semaphore code provided by the professor. We only use one semaphore variable*

*// (CRITICAL) that starts at 1. We call P before the critical section and V after;*

*// this setup results in the first process to reach this section being able to*

*// execute it and each subsequent process being unable to start the section until*

*// the other(s) are finished.*

#include *<string> // string*

#include *<iostream> // IO*

#include *<unistd.h> // fork*

#include *<sys/wait.h> // wait*

#include *<sys/shm.h> // shared mem*

#include *<limits> // numeric\_limits*

#include *"constants.h" // our constants*

#include *"semaphore.h" // code given*

**using** **namespace** std;

**enum** {CRITICAL}; *// sem var*

*// Forward declarations*

void performProcess(int, SEMAPHORE&, char\*, char\*, char\*, char\*);

int getUserInput();

*// "main" method. This method, as described above, allocates the shared memory*

*// segments, initializes them with characters, forks child processes to perform*

*// atomic swaps, waits, and deallocates the memory.*

*//*

*// Pre-conditions: None.*

*// Post-conditions: Processes ran, memory was allocated, altered, then destroyed.*

int main()

{

*// Set seed*

srand(time(NULL));

*// Allocate a shared memory segments.*

int segmentID1 = shmget(IPC\_PRIVATE, GROUP\_SIZE \* **sizeof**(char), PERMS);

int segmentID2 = shmget(IPC\_PRIVATE, GROUP\_SIZE \* **sizeof**(char), PERMS);

int segmentID3 = shmget(IPC\_PRIVATE, GROUP\_SIZE \* **sizeof**(char), PERMS);

int segmentID4 = shmget(IPC\_PRIVATE, GROUP\_SIZE \* **sizeof**(char), PERMS);

*// Attach to segments*

char\* str1 = (char\*) shmat(segmentID1, 0, SHM\_RND);

char\* str2 = (char\*) shmat(segmentID2, 0, SHM\_RND);

char\* str3 = (char\*) shmat(segmentID3, 0, SHM\_RND);

char\* str4 = (char\*) shmat(segmentID4, 0, SHM\_RND);

*// Initialize chunk data*

**for** (int i = CHUNK1\_START; i <= CHUNK3\_END; i++)

{

\*(str1 + i) = LOWER\_CASE\_LETTERS[rand() % L\_SIZE];

}

**for** (int i = CHUNK1\_START; i <= CHUNK3\_END; i++)

{

\*(str2 + i) = UPPER\_CASE\_LETTERS[rand() % L\_SIZE];

}

**for** (int i = CHUNK1\_START; i <= CHUNK3\_END; i++)

{

\*(str3 + i) = UPPER\_CASE\_LETTERS[rand() % L\_SIZE];

}

**for** (int i = CHUNK1\_START; i <= CHUNK3\_END; i++)

{

\*(str4 + i) = UPPER\_CASE\_LETTERS[rand() % L\_SIZE];

}

*// Get input on number of operations*

int numberOfOperations = getUserInput();

*// Print initial data*

printf("**\n\n**Group1:**\n\n**%s**\n**", str1);

printf("**\n\n**Group2:**\n\n**%s**\n**", str2);

printf("**\n\n**Group3:**\n\n**%s**\n**", str3);

printf("**\n\n**Group4:**\n\n**%s**\n\n\n**", str4);

*// Initialize semaphore*

SEMAPHORE sem(1);

sem.V(CRITICAL);

*// Fork*

**for** (int i = 1; i < NUM\_PROCESSES + 1; i++)

{

long childPID = 0; *// Stores PID of child (0 if it is the child)*

childPID = fork();

*// Set unique random seed for each process*

*// Source: https://stackoverflow.com/questions/12779235/*

*// how-to-properly-choose-rng-seed-for-parallel-processes*

srand(time(NULL) \* i \* getpid());

**if** (childPID == 0) *// CHILD*

{

performProcess(numberOfOperations, sem, str1, str2, str3, str4);

*// Exit child process once finished*

exit(0);

}

}

*// Wait for all processes to finish*

wait(NULL);

wait(NULL);

wait(NULL);

wait(NULL);

wait(NULL);

*// Print results*

printf("**\n\n**Group1:**\n\n**%s**\n**", str1);

printf("**\n\n**Group2:**\n\n**%s**\n**", str2);

printf("**\n\n**Group3:**\n\n**%s**\n**", str3);

printf("**\n\n**Group4:**\n\n**%s**\n**", str4);

*// Detach from segments*

shmdt(str1);

shmdt(str2);

shmdt(str3);

shmdt(str4);

*// Destroy segment memory*

shmctl(segmentID1, IPC\_RMID, NULL);

shmctl(segmentID2, IPC\_RMID, NULL);

shmctl(segmentID3, IPC\_RMID, NULL);

shmctl(segmentID4, IPC\_RMID, NULL);

**return** 0;

}

*// "getUserInput" method. This method takes no parameters. Its only function is*

*// to retrieve the number of operations for each process to execute and return it.*

*// This value can not be 0 or less. The function checks for invalid input.*

*//*

*// Pre-conditions: None.*

*// Post-conditions: Integer representing the number of operations is returned.*

int getUserInput()

{

int numberOfOperations = 0;

**while** (true)

{

cout << "Enter # of process operations: ";

cin >> numberOfOperations;

**if** (cin.fail())

{

*// Reset variable, clear input, and output error message.*

numberOfOperations = 0;

cin.clear();

cin.ignore(numeric\_limits<streamsize>::max(), '\n');

cout << "Invalid input." << endl;

}

**else** **if** (numberOfOperations < 1)

{

*// Reset variable, output error message (invalid input).*

numberOfOperations = 0;

cout << "Invalid input." << endl;

}

**else**

{

*// Valid input.*

**break**;

}

}

**return** numberOfOperations;

}

*// "performProcess" method. This is the method that each process will be*

*// executing until it completes its required number of operations. The method*

*// takes 6 parameters: the number of operations to complete (int), the sem var*

*// (SEMAPHORE&), and the four shared memory segments (char\*). The operations*

*// take place in an repeating while loop that is only broken when the process*

*// completes its operations. The first act is to generate a random*

*// 32 bit integer, and check if its less than the number provided in the lab*

*// instructions in order to slow the process down. If this occurs the main*

*// functionality of the function takes place: the random choosing of two of*

*// the segments and one chunk in each section, and the atomic swapping of*

*// these chunks. The sem var is incremented and decremented in order to prevent*

*// two processes from performing this section at once.*

*//*

*// Pre-conditions: The four shared memory segments have been allocated,*

*// the number of operations has been retrieved, and the*

*// sem var has been initialized.*

*// Post-conditions: The process has completed and is returning to the main function.*

void performProcess(int numberOfOperations, SEMAPHORE& sem,

char\* str1, char\* str2, char\* str3, char\* str4)

{

int speed\_check = 0; *// will hold random variable to slow down*

int operationsComplete = 0; *// counts number of operations we have completed*

**while** (true)

{

speed\_check = rand(); *// Generate random 32 bit integer.*

*// If less than 5000;*

**if** (speed\_check < SPEED)

{

*// Get two random groups (1 - 4) that are not the same.*

int group1;

int group2;

group1 = (rand() % NUM\_GROUPS) + 1;

group2 = group1;

**while** (group2 == group1)

{

group2 = (rand() % NUM\_GROUPS) + 1;

}

*// Get random chunks for the groups.*

int group1chunk = (rand() % NUM\_CHUNKS) + 1;

int group2chunk = (rand() % NUM\_CHUNKS) + 1;

*// Based on the random groups, choose the correct shared segment.*

char\* mem1;

char\* mem2;

**switch** (group1)

{

**case** 1:

mem1 = str1;

**break**;

**case** 2:

mem1 = str2;

**break**;

**case** 3:

mem1 = str3;

**break**;

**case** 4:

mem1 = str4;

**break**;

}

**switch** (group2)

{

**case** 1:

mem2 = str1;

**break**;

**case** 2:

mem2 = str2;

**break**;

**case** 3:

mem2 = str3;

**break**;

**case** 4:

mem2 = str4;

**break**;

}

*// Get starting indexes*

int g1\_start = 0;

int g2\_start = 0;

*// Offset based on chunk*

g1\_start += (group1chunk - 1) \* CHUNK\_SIZE;

g2\_start += (group2chunk - 1) \* CHUNK\_SIZE;

int offset = 0; *// used to index second chunk*

bool first = true; *// flag checking for first iteration*

*// Swap*

sem.P(CRITICAL); *// Decrement sem*

**for** (int i = g1\_start; i < g1\_start + CHUNK\_SIZE; i++)

{

**if** (first)

{

*// Starting critical section*

first = false;

cout << "Process " << getpid() << " swapping" << endl;

}

*// Swap algorithm*

char temp = \*(mem1 + i);

\*(mem1 + i) = \*(mem2 + g2\_start + offset);

\*(mem2 + g2\_start + offset) = temp;

offset++;

}

*// Critical section over*

cout << "Process " << getpid() << " done swapping" << endl;

sem.V(CRITICAL); *// Increment sem (P calls == V calls)*

operationsComplete++;

*// If we match the user's input, break the while loop*

**if** (operationsComplete == numberOfOperations)

{

*// This output shows there was no deadlock.*

cout << endl;

cout << getpid() << " completed " << operationsComplete;

cout << " operations" << endl;

cout << endl;

**break**;

}

}

}

}

*// Eleazar Gomez, Anton Shirokov*

*// constants.h*

*//*

*// This file contains the constants used throughout our program so that*

*// we can eliminate magic numbers. It includes arrays of characters for*

*// initializing chunk data, group and chunk attributes, and the value*

*// to check speed\_check against to slow down the processes.*

#ifndef CONSTANTS\_H

#define CONSTANTS\_H

**const** int NUM\_PROCESSES = 5;

**const** char LOWER\_CASE\_LETTERS[26] = {'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h',

'i', 'j', 'k', 'l', 'm', 'n', 'o', 'p',

'q', 'r', 's', 't', 'u', 'v', 'w', 'x',

'y', 'z'};

**const** char UPPER\_CASE\_LETTERS[26] = {'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H',

'I', 'J', 'K', 'L', 'M', 'N', 'O', 'P',

'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X',

'Y', 'Z'};

*// Size letter arrays*

**const** int L\_SIZE = 26;

*// Start and end indicies for initial setup*

**const** int CHUNK1\_START = 0;

**const** int CHUNK3\_END = 1535;

*// Group/Chunk info*

**const** int NUM\_GROUPS = 4;

**const** int NUM\_CHUNKS = 3;

**const** int GROUP\_SIZE = 1536;

**const** int CHUNK\_SIZE = 512;

*// For speed\_check*

**const** int SPEED = 5000;

#endif

*// This software may be used in your CECS326 programs*

#include *<unistd.h>*

#include *<errno.h>*

#include *<sys/types.h>*

#include *<sys/stat.h>*

#include *<sys/ipc.h>*

#include *<sys/sem.h>*

#define PERMS S\_IWUSR|S\_IRUSR *// 0600*

**class** **mysembuf** {

**public**:

**struct** sembuf sb;

};

*// need to be checking for existing definition*

**union** semun { *// truncated definition*

int val; *// value used with SETVAL*

ushort \*array; *// array of values: GETALL and SETALL*

};

**class** **SEMAPHORE** {

**private**:

int size; *// how many semaphors we need*

pid\_t semid;

mysembuf \*pset, \*vset;

int init();

void set\_sembuf\_p(int, int, int);

void set\_sembuf\_v(int, int, int);

**public**:

*// create a number of semaphores denoted by size*

*// precondition: size is larger than zero*

*// postcondition: all semaphores are initialized to zero*

SEMAPHORE(int size);

~SEMAPHORE();

*// deallocates all semaphores created by constructor*

*// precondition: existence of SEMAPHORE object*

*// postcondition: all semaphores are removed*

int remove(); *// explicitly*

*// semaphore P operation on semaphore semname*

*// precondition: existence of SEMAPHORE object*

*// postcondition: semaphore decrements and process may be blocked*

*// return value: -1 denotes an error*

int P(int semname);

*// semaphore V operation on semaphore semname*

*// precondition: existence of SEMAPHORE object*

*// postcondition: semaphore increments and process may be resumed*

*// return value: -1 denotes an error*

int V(int semname);

};

*// This software may be used in your CECS326 programs*

*// simple implementation of SEMAPHORE class with some error*

*// and signal handling*

#include *"semaphore.h"*

SEMAPHORE::SEMAPHORE(int size) {

**this**->size = size;

semid = semget(IPC\_PRIVATE, size, PERMS);

init();

}

int SEMAPHORE::remove() {

semun dummy;

**return** semctl(semid, 0 */\*not used\*/*, IPC\_RMID, dummy);

}

int SEMAPHORE::P(int id){

int retval;

**struct** sembuf \*p = &((pset+id)->sb);

**while**(((retval=semop(semid,p,1))==-1)&&(errno==EINTR));

**return** retval;

}

int SEMAPHORE::V(int id){ *// return int values, error codes (-1)*

int retval;

**struct** sembuf \*v = &((vset+id)->sb);

**while**(((retval=semop(semid,v,1))==-1)&&(errno==EINTR)); *// EINTR = interrupt*

**return** retval;

}

void SEMAPHORE::set\_sembuf\_p(int k, int op, int flg){

(pset+k)->sb.sem\_num = (short) k;

(pset+k)->sb.sem\_op = op;

(pset+k)->sb.sem\_flg = flg;

}

void SEMAPHORE::set\_sembuf\_v(int k, int op, int flg){

(vset+k)->sb.sem\_num = (short) k;

(vset+k)->sb.sem\_op = op;

(vset+k)->sb.sem\_flg = flg;

}

int SEMAPHORE::init() {

pset = **new** mysembuf[size];

vset = **new** mysembuf[size];

**for**(int k=0; k<size; k++) {

set\_sembuf\_p(k, -1, 0 */\*suspend\*/*);

set\_sembuf\_v(k, 1, 0 */\*suspend\*/*);

}

*// initialize all to zero*

semun arg;

ushort initv[size];

**for**(int k=0; k<size; k++) {

initv[k]=0;

}

arg.array = initv;

**return** semctl(semid, size, SETALL, arg);

}

SEMAPHORE::~SEMAPHORE() {

**delete** pset;

**delete** vset;

}