Lab # 09

Process Synchronization

This lab examines aspects of Process Synchronization techniques. The primary objective of this lab is to implement difference Synchronization techniques:

- Locking Technique
- Peterson's Algorithm
- Test and Set
- Compare and Swap
- Semaphores
- Monitors

1. Locking Technique (Using Bool Variable/Mutex)

Example 1:

```
#include<pthread.h>
#include<stdlib.h>
#include<unistd.h>
#include<stdio.h>
#include <stdbool.h>
bool lock = false;
int myGlobal = 0;
void *threadFunction()
{
int i, j;
for (i = 0; i<10; i++)
{
while(lock);
lock=true;
 j = myGlobal;
 j = j+1;
 myGlobal = j;
 printf("\n My Global Is: %d\n", myGlobal);
 lock=false;
 sleep(1);
}
}
int main()
{
pthread_t myThread1,myThread2;
int i,k;
pthread_create(&myThread1, NULL,threadFunction,NULL);
pthread_create(&myThread2, NULL,threadFunction,NULL);
```

```
pthread_join(myThread1, NULL);
pthread_join(myThread2, NULL);
exit(0);}
```

Example 2:

```
#include<pthread.h>
#include<stdlib.h>
#include<unistd.h>
#include<stdio.h>
int myGlobal = 0;
pthread_mutex_t myMutex;
void *threadFunction()
{
int i, j;
for (i = 0; i<10; i++)
{
pthread_mutex_lock(&myMutex);
 j = myGlobal;
 j = j+1;
 myGlobal = j;
 printf("\n My Global Is: %d\n", myGlobal);
pthread_mutex_unlock(&myMutex);
 sleep(1);
}
}
int main()
{
pthread_t myThread1,myThread2;
int i,k;
pthread_create(&myThread1, NULL,threadFunction,NULL);
pthread_create(&myThread2, NULL,threadFunction,NULL);
pthread_join(myThread1, NULL);
pthread_join(myThread2, NULL);
printf("\nMy Global Is: %d\n", myGlobal);
exit(0);
}
```

2. Peterson's Algorithm

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
bool flag[2]={false,false};
double balance = 0;
double temp1 = 0;
double temp2 = 0;
int turn;
void *credit(void * arg)
{
```

```
int a = *(int *) arg;
flag[0] =true;
turn = 1;
while(flag[1] && turn == 1) ;
balance = balance + a;
flag[0]=false;
}
void *debit(void * arg)
{
int a = *(int *) arg;
 flag[1] = true;
 turn = 0;
 while(flag[0] && turn == 0) ;
 balance = balance-a;
 flag[1] = false;
}
int main()
{
int choice;
pthread_t credit_thread, debit_thread;
while(1)
{
system("clear");
printf("Name: Student \nAcc No.: 420\n");
printf("Available Balance: Rs. %f/-\n", balance);
printf("Enter amount to credit\n");
scanf("%d", &temp1);
printf("Enter amount to Debit\n");
scanf("%d", &temp2);
pthread_create(&credit_thread, NULL,credit, &temp1);
pthread_create(&debit_thread, NULL, debit, &temp2);
}
pthread_join(credit_thread, NULL);
pthread_join(debit_thread, NULL);
printf("Account Summary:\n----------------\n");
printf("Available Balance: Rs. %f/-\n", balance);
exit(0);
}
```

3. Test and Set Instruction

```
#include<pthread.h>
#include<stdlib.h>
#include<unistd.h>
#include<stdio.h>
#include <stdbool.h>
bool lock1 = false;
int myGlobal = 0;
bool TestAndSet (bool *lock)
{
bool ret = *lock;
```

```
*lock = true;
return ret;
}
void *threadFunction()
{
int i, j;
for (i = 0; i<10; i++)
{
while (TestAndSet(&lock1));
 j = myGlobal;
 j = j+1;
 myGlobal = j;
 printf("\n My Global Is: %d\n", myGlobal);
 lock1=false;
 sleep(1);
}
}
int main()
{
pthread_t myThread1,myThread2;
int i,k;
pthread_create(&myThread1, NULL,threadFunction,NULL);
pthread_create(&myThread2, NULL,threadFunction,NULL);
pthread_join(myThread1, NULL);
pthread_join(myThread2, NULL);
printf("\nMy Global Is: %d\n", myGlobal);
exit(0);
}
```

4. Compare and Swap Instruction

```
#include<pthread.h>
#include<stdlib.h>
#include<unistd.h>
#include<stdio.h>
int myGlobal = 0;
int lock = 0;
int compare_and_swap (int* v, int exp, int new)
{
int temp = *v;
if (*v == exp)
*v = new;
return temp;
}
void *threadFunction()
{
int i, j;
for (i = 0; i<10; i++)
{
while(compare_and_swap(&lock, 0, 1));
 j = myGlobal;
```

```
 j = j+1;
 myGlobal = j;
 printf("\n My Global Is: %d\n", myGlobal);
 compare_and_swap(&lock, 1, 0);
 sleep(1);
}
}
int main()
{
pthread_t myThread1,myThread2;
int i,k;
pthread_create(&myThread1, NULL,threadFunction,NULL);
pthread_create(&myThread2, NULL,threadFunction,NULL);
pthread_join(myThread1, NULL);
pthread_join(myThread2, NULL);
printf("\nMy Global Is: %d\n", myGlobal);
exit(0);
}
```

5. Semaphores

Example 1:

```
#include<pthread.h>
#include<stdlib.h>
#include<unistd.h>
#include<stdio.h>
int myGlobal = 0;
int S=1;
wait(int* s)
{
while (*s<= 0);
*s--;
}
signal(int* s)
{
*s++;
}
void *threadFunction()
{
int i, j;
for (i = 0; i<10; i++)
{
 wait(&S);
 j = myGlobal;
 j = j+1;
 myGlobal = j;
 signal(&S);
 printf("\n My Global Is: %d\n", myGlobal);
 sleep(1);
}
```

```
}
int main()
{
pthread_t myThread1,myThread2;
int i,k;
pthread_create(&myThread1, NULL,threadFunction,NULL);
pthread_create(&myThread2, NULL,threadFunction,NULL);
pthread_join(myThread1, NULL);
pthread_join(myThread2, NULL);
printf("\nMy Global Is: %d\n", myGlobal);
exit(0);
}
```

Example 2:

```
#include<pthread.h>
#include<stdlib.h>
#include<unistd.h>
#include<stdio.h>
#include<semaphore.h>
int myGlobal = 0;
sem_t m;
void *threadFunction()
{
int i, j;
for (i = 0; i<10; i++)
{
 sem_wait(&m);
 j = myGlobal;
 j = j+1;
 myGlobal = j;
 printf("\n My Global Is: %d\n", myGlobal);
 sem_post(&m);
 sleep(1);
}
}
int main()
{
pthread_t myThread1,myThread2;
if (sem_init(&m, 0, 1) == -1) {
perror("Could not initialize mylock semaphore");
exit(2);
}
int i,k;
pthread_create(&myThread1, NULL,threadFunction,NULL);
pthread_create(&myThread2, NULL,threadFunction,NULL);
pthread_join(myThread1, NULL);
pthread_join(myThread2, NULL);
printf("\nMy Global Is: %d\n", myGlobal);
exit(0);
}
```

Semaphores

- OS guarantees that Wait() + Signal() are atomic
- No busy-waiting
- Machine independent Instructions
- Can be scaled to N processes
- Can have as many critical regions as you want by assigning different semaphore to each critical region

Monitors:

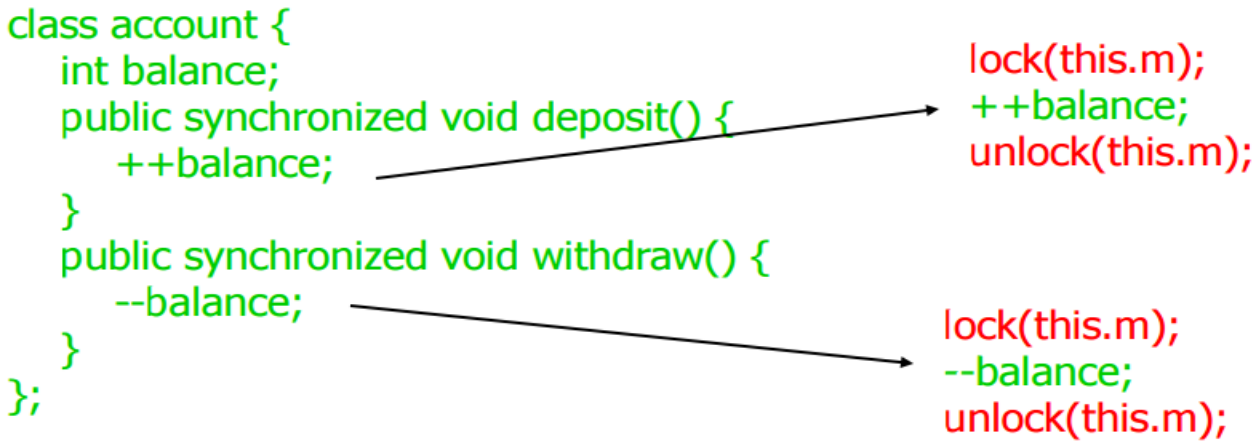
A way of introducing OOP techniques in concurrent Programming. A way to make concurrent programming more structured. You might wonder why monitors were invented at all, instead of just using explicit locking. At the time, object-oriented programming was just coming into fashion. Thus, the idea was to gracefully blend some of the key concepts in concurrent programming with some of the basic approaches of object orientation. Nothing more than that. **Monitor:** object with a set of monitor procedures and only one thread may be active (i.e. running one of the monitor procedures) at a time. Compiler automatically inserts lock and unlock operations upon entry and exit of monitor procedures. Monitor uses Condition Variables (Conceptually associated with some conditions).

Operations on condition variables:

wait(): suspends the calling thread and releases the monitor lock. When it resumes, reacquire the lock. Called when condition is not true

signal(): resumes one thread waiting in wait() if any. Called when condition becomes true and wants to wake up one waiting thread

broadcast(): resumes all threads waiting in wait(). Called when condition becomes true and wants to wake up all waiting threads.



General Form of Monitors for Banking Problem Problem:

```
monitor class account {
private:
int balance = 0;
public:
void deposit(int amount) {
balance = balance + amount;
}
void withdraw(int amount) {
balance = balance - amount;
}
};
```

C/C++ don't provide monitors; but we can implement monitors using pthread mutex and condition variable

```
class account {
private:
int balance = 0;
pthread_mutex_t monitor;
public:
void deposit(int amount) {
pthread_mutex_lock(&monitor);
balance = balance + amount;
pthread_mutex_unlock(&monitor);
}
void withdraw(int amount) {
pthread_mutex_lock(&monitor);
balance = balance - amount;
pthread_mutex_unlock(&monitor);
}
};
```

Monitor for Producer and Consumer Problem:

```
monitor class BoundedBuffer {
private:
int buffer[MAX];
int fill, use;
int fullEntries = 0;
cond_t empty;
cond_t full;
public:
void produce(int element) {
if (fullEntries == MAX) // line P0
wait(&empty); // line P1
buffer[fill] = element; // line P2
fill = (fill + 1);
fullEntries++; // line P4
```

```
signal(&full); // line P5
}
int consume() {
if (fullEntries == 0) // line C0
wait(&full); // line C1
int tmp = buffer[use]; // line C2
use = (use + 1); // line C3
fullEntries--; // line C4
signal(&empty); // line C5
return tmp; // line C6
}
};
```

Monitor for Producer and Consumer Problem using Pthreads:

```
class BoundedBuffer {
private:
int buffer[MAX];
int fill, use;
int fullEntries;
pthread_mutex_t monitor; // monitor lock
pthread_cond_t empty;
pthread_cond_t full;
public:
BoundedBuffer() {
use = fill = fullEntries = 0;
}
void produce(int element) {
pthread_mutex_lock(&monitor);
 if(fullEntries == MAX)
pthread_cond_wait(&empty, &monitor);
buffer[fill] = element;
fill = (fill + 1);
fullEntries++;
pthread_cond_signal(&full);
pthread_mutex_unlock(&monitor);
}
int consume() {
pthread_mutex_lock(&monitor);
if(fullEntries == 0)
pthread_cond_wait(&full, &monitor);
int tmp = buffer[use];
use = (use + 1);
fullEntries--;
pthread_cond_signal(&empty);
pthread_mutex_unlock(&monitor);
return tmp;
}
};
```

Exercises:

EX # 01

Three Processes P1, P2, P3, Ignoring Preemption
Two Semaphores S1 = 1, S2 = 0
Which execution order is possible for?

```
P1
do {
    wait(S1);
    ... Print A
    signal(S2);
} while(1);
```

```
P2
do {
    wait(S2);
    ... Print B
    signal(S2);
} while(1);
```

```
P3
do {
    wait(S2);
    ... Print C
    signal(S1);
} while(1);
```

EX # 02

Three Processes Red, Green, Blue
Sequence Required:
Red → Green → Blue
Write Code Hint: Use three semaphores