Computer Operating Systems, Practice Session 7

Example Synchronization Problems

Resul Tugay (tugayr@itu.edu.tr)

April 5, 2017

Today

Computer Operating Systems, PS 7

- Step by Step Semaphore Usage
- Example Synchronization Problems

Problem

Purpose: Control a Printer System

- System has 1 printer
- Printer prints out (to the screen) a single character for each page it actually prints out
- Within a short period, jobs are given to this printer as to print out 100 copies (for each doc.) of two different documents
- These documents should be printed by protecting the integrity of each document

Without Synchronization

```c
#include <pthread.h>
#include <stdio.h>

void* printThis(void* typ){
  int i,j;
  char* str = (char)typ=='a'?"abcdefghij":"0123456789"; // 2 types of contents
  for(i=0; i<100; i++) // 100 separate print jobs
      for(j=0; j<10; j++) // of 10 pages each
        printf("%c", str[j]); // each character represents a page
  pthread_exit(NULL);
}

int main(void){
  printf("I'm the NO-SYNC printer manager.\n");
  setvbuf(stdout, (char*)NULL, _IONBF, 0); // no-buffer printf
  pthread_t a,n;   // create two threads (a thread for each set of documents)
  pthread_create(&a, NULL, printThis, (void *)'a');
  pthread_create(&n, NULL, printThis, (void *)'n');
  // wait for the threads to finish
  pthread_join(a, NULL);
  pthread_join(n, NULL);
  pthread_exit(NULL);
  return 0;
}
```

Example Output (Without Synchronization)

```
I'm the NO-SYNC printer manager.
01234567890123456789012345678901234567890123456789012345678901234567890123456789
01234567890123456789012345678901234567890123456789012345678901234567890123456789
01234567890123456789012345678901234567890123456789012345678901234567890123456789
01234567890123456789012345678901234567890123456789012345678901234567890123456789
01234567890123456789012345678901234567890123456789012345678901234567890123456789
01234567890123456789012345678901234567890123456789012345678901234567890123456789
01234567890123456789012345678901234567890123456789012345678901234567890123456789
01234567890123456789012345678901234567890123456789012345678901234567890123456789
01234567890123456789012345678901234567890123456789012345678901234567890123456789
012345abcdefghijabcdefghijabcdefghijabcdefghijabcdefghijabcdefghijabcdefghijabcd
efghijabcdefghijabcdefghijabcdefghijabcdefghijabcdefghijabcdefghijabcdefghijabcd
efghijabcdefghi67890123456789012345678901234567890123456789012345678901234567890
12345678901234567890123456789012345678901234567890123456789012345678901234567890
12345678901234567890123456789012345678901234567890123456789012345678901234567890
1234567890123456789012345678901234567890123456789jabcdefghijabcdefghijabcdefghij
abcdefghijabcdefghijabcdefghijabcdefghijabcdefghijabcdefghijabcdefghijabcdefghij
abcdefghijabcdefghijabcdefghijabcdefghijabcdefghijabcdefghijabcdefghijabcdefghij
abcdefghijabcdefghijabcdefghijabcdefghijabcdefghijabcdefghijabcdefghijabcdefghij
abcdefghijabcdefghijabcdefghijabcdefghijabcdefghijabcdefghijabcdefghijabcdefghij
abcdefghijabcdefghijabcdefghijabcdefghijabcdefghijabcdefghijabcdefghijabcdefghij
abcdefghijabcdefghijabcdefghijabcdefghijabcdefghijabcdefghijabcdefghijabcdefghij
abcdefghijabcdefghijabcdefghijabcdefghijabcdefghijabcdefghijabcdefghijabcdefghij
abcdefghijabcdefghijabcdefghijabcdefghijabcdefghijabcdefghijabcdefghijabcdefghij
abcdefghijabcdefghijabcdefghijabcdefghijabcdefghijabcdefghijabcdefghijabcdefghij
abcdefghijabcdefghijabcdefghijabcdefghijabcdefghijabcdefghijabcdefghijabcdefghij
```

Synchronization Attempt

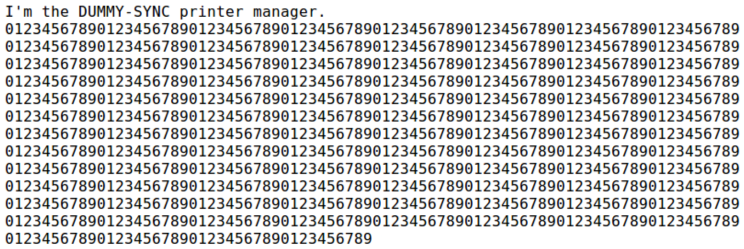
```
#include <pthread.h>
#include <stdio.h>

int s;

void* printThis(void* typ){
    int i,j;
    // create two types of contents for print jobs
    char* str = (char)typ=='a'?"abcdefghij":"0123456789";
    for(i=0; i<100; i++) // 100 separate print jobs
        if(s>0){
            s--; // lock other threads
            for(j=0;j<10;j++)
                printf("%c", str[j]);
            s++; // unlock other threads
        }
    pthread_exit(NULL);
}
```

Problem 1: Using a global variable *s* shared among threads without mutual exclusion

Synchronization Attempt

```
int main(void){
  printf("I'm the DUMMY-SYNC printer manager.\n");
  setvbuf(stdout, (char*)NULL, _IONBF, 0); // no-buffer printf
  s = 1; // printer is initially available
  pthread_t a,n;   // create two threads (a thread for each set of documents)
  pthread_create(&a, NULL, printThis, (void *)'a');
  pthread_create(&n, NULL, printThis, (void *)'n');
  // wait for the threads to finish
  pthread_join(a, NULL);
  pthread_join(n, NULL);
  pthread_exit(NULL);
  return 0;
}
```

Example Output (Synchronization Attempt)



```
I'm the DUMMY-SYNC printer manager.
01234567890123456789012345678901234567890123456789012345678901234567890123456789
01234567890123456789012345678901234567890123456789012345678901234567890123456789
01234567890123456789012345678901234567890123456789012345678901234567890123456789
01234567890123456789012345678901234567890123456789012345678901234567890123456789
01234567890123456789012345678901234567890123456789012345678901234567890123456789
01234567890123456789012345678901234567890123456789012345678901234567890123456789
01234567890123456789012345678901234567890123456789012345678901234567890123456789
01234567890123456789012345678901234567890123456789012345678901234567890123456789
01234567890123456789012345678901234567890123456789012345678901234567890123456789
01234567890123456789012345678901234567890123456789012345678901234567890123456789
01234567890123456789012345678901234567890123456789012345678901234567890123456789
01234567890123456789012345678901234567890123456789012345678901234567890123456789
0123456789012345678901234567890123456789
```

Problem 2: in `printThis()` function, no behavior (e.g., waiting) is defined for the case $s = 0$

Successful Synchronization

```
#define _GNU_SOURCE
#include <sys/ipc.h>
#include <sys/sem.h>
#include <sys/types.h>
#include <pthread.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>


// to create a semaphore for mutual exclusion(value=1)
int s;
void create(char *argv[]){
    // The ftok() function returns a key based on path and id that
    // is usable in subsequent calls to semget() and shmget()
    int someKey = ftok(strcat(get_current_dir_name(),argv[0]),1);
    s = semget(someKey, 1, 0700|IPC_CREAT);
    semctl(s, 0, SETVAL, 1); // semaphore value = 1
}
```

Successful Synchronization

```
void sem_signal(int semid, int val){ // semaphore increment operation
    struct sembuf semaphore;
    semaphore.sem_num=0;
    semaphore.sem_op=val;
    semaphore.sem_flg=1;
    semop(semid, &semaphore, 1);
}

void sem_wait(int semid, int val){ // semaphore decrement operation
    struct sembuf semaphore;
    semaphore.sem_num=0;
    semaphore.sem_op=(-1*val);
    semaphore.sem_flg=1;
    semop(semid, &semaphore, 1);
}

void increase(int sid){ // to increase semaphore value by 1
  sem_signal(s,1);
}

void decrease(int sid){ // to decrease semaphore value by 1
  sem_wait(s,1);
}
```

Successful Synchronization

```
void* printThis(void* typ){
    int i,j;
    char* str = (char)typ=='a'?"abcdefghij":"0123456789"; // 2 types of contents
    for(i=0; i<100; i++) {// 100 separate print jobs
        decrease(s); // lock other threads
        for(j=0;j<10;j++)
            printf("%c", str[j]);
        increase(s);// unlock other threads
    }
    pthread_exit(NULL);
}
int main(int argc, char *argv[]){
  printf("I'm the SEM-SYNC printer manager.\n");
  setvbuf(stdout, (char*)NULL, _IONBF, 0); // no-buffer printf
  create(argv);   // create a semaphore for mutual exclusion
  pthread_t a,n;   // create two threads (a thread for each set of documents)
  pthread_create(&a, NULL, printThis, (void *)'a');
  pthread_create(&n, NULL, printThis, (void *)'n');
  pthread_join(a, NULL);
  pthread_join(n, NULL);
  semctl(s, 0, IPC_RMID, 0); // removing the created semaphore
  pthread_exit(NULL);
  return 0;
}
```

Example Output (Successful Synchronization)

```
I'm the SEM-SYNC printer manager.
012345678901234567890123456789abcdefghij0123456789abcdefghij0123456789abcdefghij
0123456789abcdefghij0123456789abcdefghij0123456789abcdefghij0123456789abcdefghij
0123456789abcdefghij0123456789abcdefghij0123456789abcdefghij0123456789abcdefghij
0123456789abcdefghij0123456789abcdefghij0123456789abcdefghij0123456789abcdefghij
0123456789abcdefghij0123456789abcdefghij0123456789abcdefghij0123456789abcdefghij
0123456789abcdefghij0123456789abcdefghij0123456789abcdefghij0123456789abcdefghij
0123456789abcdefghij0123456789abcdefghij0123456789abcdefghij0123456789abcdefghij
0123456789abcdefghij0123456789abcdefghij0123456789abcdefghij0123456789abcdefghij
0123456789abcdefghij0123456789abcdefghij0123456789abcdefghij0123456789abcdefghij
0123456789abcdefghij0123456789abcdefghij0123456789abcdefghij0123456789abcdefghij
0123456789abcdefghij0123456789abcdefghij0123456789abcdefghij0123456789abcdefghij
0123456789abcdefghij0123456789abcdefghij0123456789abcdefghij0123456789abcdefghij
0123456789abcdefghij0123456789abcdefghij0123456789abcdefghij0123456789abcdefghij
0123456789abcdefghij0123456789abcdefghij0123456789abcdefghij0123456789abcdefghij
0123456789abcdefghij0123456789abcdefghij0123456789abcdefghij0123456789abcdefghij
0123456789abcdefghij0123456789abcdefghij0123456789abcdefghij0123456789abcdefghij
0123456789abcdefghij0123456789abcdefghij0123456789abcdefghij0123456789abcdefghij
0123456789abcdefghij0123456789abcdefghij0123456789abcdefghij0123456789abcdefghij
0123456789abcdefghij0123456789abcdefghij0123456789abcdefghij0123456789abcdefghij
0123456789abcdefghij0123456789abcdefghij0123456789abcdefghij0123456789abcdefghij
0123456789abcdefghij0123456789abcdefghij0123456789abcdefghij0123456789abcdefghij
0123456789abcdefghij0123456789abcdefghij0123456789abcdefghij0123456789abcdefghij
0123456789abcdefghij0123456789abcdefghij0123456789abcdefghij0123456789abcdefghij
0123456789abcdefghij0123456789abcdefghij0123456789abcdefghij0123456789abcdefghij
0123456789abcdefghij0123456789abcdefghij0123456789abcdefghijabcdefghijabcdefghij
```

Water (H_2O) Production Problem

- We want to model the production of a water molecule using two threads for modelling oxygen (O) and hydrogen (H).
- A water molecule is assembled by using an O thread and 2 H threads (by using the `bond()` function)
- An O thread waits for 2 H threads if they are not ready.
- Similarly, an H thread waits for another H thread and an O thread.

Prepare the synchronization steps for oxygen and hydrogen threads required for the described behavior above.

Water (H_2O) Production Problem

Required Variables and Initial Values:

- `mutex = Semaphore(1)` Mutex: MUTual EXclusion
- `oxygen = 0` Counter for oxygen atoms, protected by mutex
- `hydrogen = 0` Counter for hydrogen atoms, protected by mutex
- `oxyQueue = Semaphore(0)` Semaphore on which oxygen atoms (threads) wait
- `hydroQueue = Semaphore(0)` Semaphore on which hydrogen atoms (threads) wait
- Initially `hydroQueue` & `oxyQueue` semaphores are locked (both are equal to zero)
- Basic functions related to queuing structures are named as follows:
  - `oxyQueue.wait()` : Enqueue an oxygen thread to oxygen queue
  - `hydroQueue.wait()` : Enqueue an hydrogen thread to hydrogen queue
  - `oxyQueue.signal()` : Dequeue an oxygen thread from oxygen queue
  - `hydroQueue.signal()` : Dequeue an hydrogen thread from hydrogen queue

Oxygen Thread Code Summary

An O thread that has arrived should allow 2 H threads to be processed by increasing `hydroQueue` semaphore by 2

- If there are at least two H threads waiting, it signals them and itself
- If not, it releases the mutex and waits

Only the oxygen thread releases the mutex after bonding to guarantee that the mutex is signalled once (there is only one oxygen in each water molecule).

```
mutex.wait()
oxygen += 1
if (hydrogen >= 2)
  hydroQueue.signal(2)
  hydrogen -= 2
  oxyQueue.signal()
  oxygen -=1
else
  mutex.signal()

oxyQueue.wait()
bond()
mutex.signal()
```

Hydrogen Thread Code Summary

H thread has a similar code:

```
mutex.wait()
hydrogen += 1
if (hydrogen >=2 and oxygen >=1)
  hydroQueue.signal(2)
  hydrogen -= 2
  oxyQueue.signal()
  oxygen -= 1
else
  mutex.signal()

hydroQueue.wait()
bond()
```

Only the oxygen thread releases the mutex after bonding to guarantee that the mutex is signalled once (there is only one oxygen in each water molecule).

Sushi Bar Problem

- Imagine a sushi bar with 5 seats. If you arrive while there is an empty seat, you can take a seat immediately. But if you arrive when all 5 seats are full, that means that all of them are dining together, and you will have to wait for the entire party to leave before you sit down.
- Design a program for simulating customers entering and leaving this sushi bar.

Variables Used in the Solution

```
eating = waiting = 0 // keep track of the number of threads
mutex = Semaphore(1) // mutex protects both counters
block = Semaphore(0) // incoming customers' queue(regular meaning)
must_wait = False // indicates that the bar is full
```

Solution Attempt

```
 1 mutex.wait()
 2 if must_wait:
 3   waiting += 1
 4   mutex.signal()
 5   block.wait()
 6   mutex.wait()    // reacquire mutex
 7   waiting -= 1
 8
 9 eating += 1
10 must_wait = (eating == 5)
11 mutex.signal()
12
13 // eat sushi
14
15 mutex.wait()
16 eating -= 1
17 if eating == 0:
18   n = min(5, waiting)
19   block.signal(n)
20   must_wait = False
21 mutex.signal()
```

The problem is at Line 6. It is possible for newly arrived threads to take all the seats before the waiting threads.

A Solution

The reason a waiting customer reacquires mutex is to update eating/waiting state.

- Make the departing customer, who already has the mutex, do the updating.

```
mutex.wait()
if must_wait:
  waiting += 1
  mutex.signal()
  block.wait()
else:
  eating += 1
  must_wait = (eating == 5)
  mutex.signal()

// eat sushi

mutex.wait()
eating -= 1
if eating == 0:
  n = min(5, waiting)
  waiting -= n
  eating += n
  must_wait = (eating == 5)
  block.signal(n)
mutex.signal()
```

Another Solution

- If there are fewer than 5 customers at the bar and no one waiting, an entering customer just increments `eating` and releases the mutex. The fifth customer sets `must_wait`.
- If there are 5 customers at the bar, entering customers `block` until the last customer at the bar clears `must_wait` and signals `block`.
  - The signaling thread gives up the mutex and the waiting thread receives it.
  - This process continues, with each thread passing the mutex to the next until there are no more chairs or no more waiting threads.

Another Solution

```
mutex.wait()
if must_wait:
  waiting += 1
  mutex.signal()
  block.wait()   // when the thread resumes after wait, it has the passed mutex
  waiting -= 1

eating += 1
must_wait = (eating == 5)
if waiting and not must_wait:
  block.signal()    // and pass the mutex (no mutex.signal())
else:
  mutex.signal()

// eat sushi

mutex.wait()
eating -= 1
if eating == 0: must_wait = False

if waiting and not must_wait:
  block.signal()    // and pass the mutex (no mutex.signal())
else:
  mutex.signal()
```

Party Problem

For a party to be held in the dormitory, below constraints are defined:

- Any number of students can be in a room at the same time.
- Dorm manager can step into the room on below two conditions:
  - To search for the room if there is no student inside
  - To terminate the party if there are more than 50 students in the room
- When the manager is inside the room, no other student can come in but insider students can come out.
- After terminating a party, manager leaves the room only when it becomes empty.
- There exists only one manager.

Implement a simulation holding for above constraints defined.

Party Problem

Required Variables and Initial Values:

- `student = 0` Number of students in the room
- `manager = "notInRoom"` Holds the status of the manager (`enum`)
- `mutex = Semaphore(1)` For protecting student and manager status
- `gateLock = Semaphore(1)` Used for holding incoming students when the manager is inside
- `roomReady = Semaphore(0)` Manager is outside and the room is empty or there is a party with 50 or more students.
- `studentsOut = Semaphore(0)` Manager is inside and all students has left the room

Code Summary for the Manager

```
mutex.wait()
if student > 0 and student < 50:
  manager = 'waiting'
  mutex.signal()
  roomReady.wait()     // and get the mutex from the student

if student >= 50:  // student count should be >= 50 for terminating the party
  manager = 'inRoom'
  WarnStudentsToLeave()
  gateLock.wait()        // lock entrance
  mutex.signal()
  studentsOut.wait()     // and get the mutex from the student
  gateLock.signal()     // remove the lock on entrance

else: // student count should be 0 for searching
  searchRoom()     // the manager has the mutex ensuring students can not enter

manager = 'notInRoom'
mutex.signal()
```

- The manager waits until the room is empty or there are 50 or more students.
- The manager terminates the party if there are 50 or more students.
- The manager searches the room if the room is empty.

Code Summary for the Student

```
mutex.wait()
if manager == 'inRoom':
  mutex.signal()
  gateLock.wait()   // wait for the manager to leave the room
  gateLock.signal() // make gateLock 1 so that the manager can lock it again
  mutex.wait()
student += 1
if student == 50 and manager == 'waiting':
  roomReady.signal()     // pass the mutex to manager
else:
  mutex.signal()
party()
mutex.wait()
student -= 1
if student == 0 and manager == 'waiting':
  roomReady.signal()     // pass the mutex to manager
else if student == 0 and manager == 'inRoom':
  studentsOut.signal()     // pass the mutex to manager
else:
  mutex.signal()
```

- If the manager is waiting, then the 50th student in or the last one out has to signal `roomReady`.
- If the manager is in the room, the last student out signals `studentsOut`.

References

- Downey, A. B. (2008). The little book of semaphores. Version 2.1.5. Green Tea Press.