

Computer Operating Systems, Practice Session 7

Example Synchronization Problems

Mustafa Ersen (ersenm@itu.edu.tr)

March 30, 2016

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

```
1 #include <pthread.h>
2 #include <stdio.h>
3
4 void* printThis(void* typ){
5     int i,j;
6     char* str = (char)typ=="a"? "abcdefghij": "0123456789"; // 2 types of contents
7     for(i=0; i<100; i++) // 100 separate print jobs
8         for(j=0; j<10; j++) // of 10 pages each
9             printf("%c", str[j]); // each character represents a page
10    pthread_exit(NULL);
11 }
12
13 int main(void){
14     printf("I'm the NO-SYNC printer manager.\n");
15     setvbuf(stdout, (char*)NULL, _IONBF, 0); // no-buffer printf
16     pthread_t a,n; // create two threads (a thread for each set of documents)
17     pthread_create(&a, NULL, printThis, (void *)'a');
18     pthread_create(&n, NULL, printThis, (void *)'n');
19     // wait for the threads to finish
20     pthread_join(a, NULL);
21     pthread_join(n, NULL);
22     pthread_exit(NULL);
23     return 0;
24 }
```

I'm the NO_SYNC printer manager

[illegible]

Synchronization Attempt

```
1 #include <pthread.h>
2 #include <stdio.h>
3
4 int s;
5
6 void* printThis(void* typ){
7     int i,j;
8     // create two types of contents for print jobs
9     char* str = (char)typ=='a'?"abcdefghij":"0123456789";
10    for(i=0; i<100; i++) // 100 separate print jobs
11        if(s>0){
12            s--; // lock other threads
13            for(j=0;j<10;j++)
14                printf("%c", str[j]);
15            s++; // unlock other threads
16        }
17    pthread_exit(NULL);
18 }
```

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

Synchronization Attempt

```
1 int main(void){
2     printf("I'm the DUMMY-SYNC printer manager.\n");
3     setvbuf(stdout, (char*)NULL, _IONBF, 0); // no-buffer printf
4     s = 1; // printer is initially available
5     pthread_t a,n; // create two threads (a thread for each set of documents)
6     pthread_create(&a, NULL, printThis, (void *)'a');
7     pthread_create(&n, NULL, printThis, (void *)'n');
8     // wait for the threads to finish
9     pthread_join(a, NULL);
10    pthread_join(n, NULL);
11    pthread_exit(NULL);
12    return 0;
13 }
```

Example Output (Synchronization Attempt)

I'm the DUMMY-SYNC printer manager.

```
0123456789012345678901234567890123456789012345678901234567890123456789
0123456789012345678901234567890123456789012345678901234567890123456789
0123456789012345678901234567890123456789012345678901234567890123456789
0123456789012345678901234567890123456789012345678901234567890123456789
0123456789012345678901234567890123456789012345678901234567890123456789
0123456789012345678901234567890123456789012345678901234567890123456789
0123456789012345678901234567890123456789012345678901234567890123456789
0123456789012345678901234567890123456789012345678901234567890123456789
0123456789012345678901234567890123456789012345678901234567890123456789
0123456789012345678901234567890123456789012345678901234567890123456789
0123456789012345678901234567890123456789012345678901234567890123456789
0123456789012345678901234567890123456789012345678901234567890123456789
0123456789012345678901234567890123456789012345678901234567890123456789
0123456789012345678901234567890123456789012345678901234567890123456789
0123456789012345678901234567890123456789012345678901234567890123456789
0123456789012345678901234567890123456789012345678901234567890123456789
0123456789012345678901234567890123456789012345678901234567890123456789
```

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

Successful Synchronization

```
1 #define _GNU_SOURCE
2 #include <sys/ipc.h>
3 #include <sys/sem.h>
4 #include <sys/types.h>
5 #include <pthread.h>
6 #include <unistd.h>
7 #include <stdio.h>
8 #include <stdlib.h>
9 #include <string.h>
10
11 // to create a semaphore for mutual exclusion(value=1)
12 int s;
13 void create(char *argv[]) {
14     // The ftok() function returns a key based on path and id that
15     // is usable in subsequent calls to semget() and shmget()
16     int someKey = ftok(strcat(get_current_dir_name(), argv[0]), 1);
17     s = semget(someKey, 1, 0700|IPC_CREAT);
18     semctl(s, 0, SETVAL, 1); // semaphore value = 1
19 }
20
```

Successful Synchronization

```
1 void sem_signal(int semid, int val){ // semaphore increment operation
2     struct sembuf semaphore;
3     semaphore.sem_num=0;
4     semaphore.sem_op=val;
5     semaphore.sem_flg=1;
6     semop(semid, &semaphore, 1);
7 }
8
9 void sem_wait(int semid, int val){ // semaphore decrement operation
10    struct sembuf semaphore;
11    semaphore.sem_num=0;
12    semaphore.sem_op=(-1*val);
13    semaphore.sem_flg=1;
14    semop(semid, &semaphore, 1);
15 }
16
17 void increase(int sid){ // to increase semaphore value by 1
18     sem_signal(s,1);
19 }
20
21 void decrease(int sid){ // to decrease semaphore value by 1
22     sem_wait(s,1);
23 }
```

Successful Synchronization

```
1 void* printThis(void* typ){
2     int i,j;
3     char* str = (char)typ=='a'?"abcdefghij":"0123456789"; // 2 types of contents
4     for(i=0; i<100; i++) { // 100 separate print jobs
5         decrease(s); // lock other threads
6         for(j=0; j<10; j++)
7             printf("%c", str[j]);
8         increase(s); // unlock other threads
9     }
10    pthread_exit(NULL);
11 }
12 int main(int argc, char *argv[]){
13     printf("I'm the SEM-SYNC printer manager.\n");
14     setvbuf(stdout, (char*)NULL, _IONBF, 0); // no-buffer printf
15     create(argv); // create a semaphore for mutual exclusion
16     pthread_t a,n; // create two threads (a thread for each set of documents)
17     pthread_create(&a, NULL, printThis, (void *)'a');
18     pthread_create(&n, NULL, printThis, (void *)'n');
19     pthread_join(a, NULL);
20     pthread_join(n, NULL);
21     semctl(s, 0, IPC_RMID, 0); // removing the created semaphore
22     pthread_exit(NULL);
23     return 0;
24 }
```

I'm the SEM-SYNC printer manager.

[illegible]

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).

```
1 mutex.wait()
2 oxygen += 1
3 if (hydrogen >= 2)
4     hydroQueue.signal(2)
5     hydrogen -= 2
6     oxyQueue.signal()
7     oxygen -= 1
8 else
9     mutex.signal()
10
11 oxyQueue.wait()
12 bond()
13 mutex.signal()
```

Hydrogen Thread Code Summary

H thread has a similar code:

```
1 mutex.wait()
2 hydrogen += 1
3 if (hydrogen >=2 and oxygen >=1)
4     hydroQueue.signal(2)
5     hydrogen -= 2
6     oxyQueue.signal()
7     oxygen -= 1
8 else
9     mutex.signal()
10
11 hydroQueue.wait()
12 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

```
1 eating = waiting = 0 // keep track of the number of threads
2 mutex = Semaphore(1) // mutex protects both counters
3 block = Semaphore(0) // incoming customers' queue(regular meaning)
4 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.

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

```
1 mutex.wait()
2 if must_wait:
3     waiting += 1
4     mutex.signal()
5     block.wait()    // when the thread resumes after wait, it has the passed mutex
6     waiting -= 1
7
8 eating += 1
9 must_wait = (eating == 5)
10 if waiting and not must_wait:
11     block.signal()    // and pass the mutex (no mutex.signal())
12 else:
13     mutex.signal()
14
15 // eat sushi
16
17 mutex.wait()
18 eating -= 1
19 if eating == 0: must_wait = False
20
21 if waiting and not must_wait:
22     block.signal()    // and pass the mutex (no mutex.signal())
23 else:
24     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

```
1 mutex.wait()
2 if student > 0 and student < 50:
3     manager = 'waiting'
4     mutex.signal()
5     roomReady.wait()    // and get the mutex from the student
6
7 if student >= 50: // student count should be >= 50 for terminating the party
8     manager = 'inRoom'
9     WarnStudentsToLeave()
10    gateLock.wait()    // lock entrance
11    mutex.signal()
12    studentsOut.wait() // and get the mutex from the student
13    gateLock.signal()  // remove the lock on entrance
14
15 else: // student count should be 0 for searching
16     searchRoom()    // the manager has the mutex ensuring students can not enter
17
18 manager = 'notInRoom'
19 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

```
1 mutex.wait()
2 if manager == 'inRoom':
3     mutex.signal()
4     gateLock.wait() // wait for the manager to leave the room
5     gateLock.signal() // make gateLock 1 so that the manager can lock it again
6     mutex.wait()
7     student += 1
8     if student == 50 and manager == 'waiting':
9         roomReady.signal() // pass the mutex to manager
10 else:
11     mutex.signal()
12     party()
13     mutex.wait()
14     student -= 1
15     if student == 0 and manager == 'waiting':
16         roomReady.signal() // pass the mutex to manager
17     else if student == 0 and manager == 'inRoom':
18         studentsOut.signal() // pass the mutex to manager
19 else:
20     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.