SE350: Operating Systems

Mahmoud Salem

m4salem@uwaterloo.ca

Please email through Learn

Office hours will be decided if needed

Tutorial

- Chapter 5 :
  - Review Questions: 5.1 – 5.7
  - Problems: 5.1, 5.3, 5.7, 5.8, 5.12, 5.21

7th and 8th Editions have same numbering for those problems.

Chapter 5 Concurrency: Mutual Exclusion and Synchronization

Reference:

Operating Systems "Internals and Design Principles" 8th Edition – William Stallings

Design issues raised by existence of concurrency – handled by OS

- Keep track of various processes
  - this is handled by process control blocks
- OS must allocate and de-allocate resources
  - processing time, memory, files, I/O, ..
- Protection of data and resources from unintended interference by other processes.
- Function of process and output produced must be independent of the speed relative to the speed of other concurrent processes.

Concurrency context

- Multiple applications [Multiprogramming]
  - processing time is dynamically shared among active applications
- Structure applications
  - some applications can be effectively programmed as a set of concurrent processes
- OS Structure
  - OS itself often implemented as set of processes and threads

Mutual Exclusion

- The basic requirement for execution of concurrent processes is to enforce mutual-exclusion.
- It’s the requirement that when one process is in the critical section that access shared resources, no other process may be in a critical section that access any of those shared resources.

Degrees of awareness between processes

- Processes unaware of each other
  - independent, non intended to work together
  - OS regulate their **competition** for resources
- Processes indirectly aware of each other
  - not necessarily aware of each other w.r.t to IDs
  - but might share access to same object as I/O buffer, have to **cooperate**
- Processes directly aware of each other
  - communicate with each other **(cooperate)** by process ID to work jointly on some activity

Cooperation vs Competition

- Competing processes:
  - need access to the same resource at the same time as file or printer
- Cooperating processes:
  - Either share access to common object
    - Such as memory buffer
  - Or able to communicate with each other
    - performance of some application

Check Table 5.2 in text book !

Control problems with competing processes

- Mutual Exclusion
  - need access to same resource at same time
  - critical section code (as printing code) - critical resource (as a printer)
- Deadlock
  - competing process need exclusive access to more than one resource
  - situation where two or more processes are unable to proceed because each is waiting for one of the others
- Starvation
  - a competing process which is ready to proceed may be indefinitely denied access to needed resource because of other processes monopolizing this resource. (overlooked by the dispatcher)

Problems

Problem 5.1

- The same problems of concurrency are present for both multiprogramming and multiprocessing.
  - Mention two differences in terms of concurrency between multiprogramming and multiprocessing ?

Problem 5.1 Solution

- Multiprogramming
  - Interleaving of execution on uniprocessor
  - Concurrency is handled by disabling interrupts
- Multiprocessing
  - Overlapping of execution on multiple processor
  - Concurrency is handled by locking shared objects
  - Ex: Memory accessed by multiple processors at the same time.

Problem 5.3

```
   P1: {
shared int x;
x = 10;
while (1) {
    x = x - 1;
    x = x + 1;
    if (x != 10)
      printf("x is %d",x)
    }
  }
}
```

```
   P2:{
shared int x;
x = 10;
while ( 1 ) {
    x = x - 1;
    x = x + 1;
    if (x!=10)
      printf("x is %d",x)
    }
  }
}
```

- Consider this program running on a uniprocessor system, the scheduler can run the instructions in pseudo parallel fashion.
  - Show an execution trace to print "x is 10" ?

Solution 5.3a

- The key here is to find a scenario that reach the check before the print statement where x !=10, then by the time the print statement is executed the x should have been equal to 10.

P1	P2	Value of X
X = X-1		9
X = X+1		10
	X = X-1	9
If(x != 10)		9
	X= X+1	10
print X		10
	If(x != 10)	10

Problem 5.3 - continued

- Show a sequence such that the statement “x is 8” printed.

Knowing that the increment and decrement are not done atomically in the assembly language code as follows:

```
LD    R0,X  /* load R0 from memory location x */
INCR  R0    /* increment R0 */
STO   R0,X  /* store the incremented value back in X */
```

Solution 5.3b

- The key here is to find a scenario where you a the loop iteration decreases the value of X so eventually you will print the value of 8.

<table>
<tr><th colspan="2">P1</th><th colspan="2">P2</th><th>Values</th></tr>
<tr><td colspan="2">X = X -1</td><td colspan="2"></td><td>V(X) = 9</td></tr>
<tr><td colspan="2"></td><td colspan="2">X = X-1</td><td>V(X) = 8</td></tr>
<tr><td>X= X+1</td><td>LD R01,X</td><td colspan="2"></td><td>V(RO1) = 8</td></tr>
<tr><td></td><td>INC RO1</td><td colspan="2"></td><td>V(RO1) = 9</td></tr>
<tr><td colspan="2"></td><td>X= X+1</td><td>LD R02,X</td><td>V(RO2) = 8</td></tr>
<tr><td colspan="2"></td><td></td><td>INC RO2</td><td>V(RO2) = 9</td></tr>
<tr><td colspan="2"></td><td></td><td>STO RO2,X</td><td>V(X) = 9</td></tr>
<tr><td></td><td>STO RO1,X</td><td colspan="2"></td><td>V(X) = 9</td></tr>
<tr><td colspan="2">Print X</td><td colspan="2"></td><td>V(X) = 9</td></tr>
<tr><td colspan="2"></td><td colspan="2">Print X</td><td>V(X) = 9</td></tr>
</table>

Repeat this scenario for more iteration to print “X is 8”

Problem 5.7

- Lamport's Bakery Algorithm

```
boolean choosing[n];
int number[n];
while (true) {
   choosing[i] = true;
   number[i] = 1 + getmax(number[], n);
   choosing[i] = false;
   for (int j = 0; j < n; j++){
     while (choosing[j]) { };
     while ((number[j] != 0) && (number[j],j) < (number[i],i)) { };
   }
   /* critical section */;
   number [i] = 0;
   /* remainder */;
}
```

- *choosing*[ ] is initialized to false, and *number*[ ] is initialized to 0.
- The i^{th} element of an array is only writeable by process i, can be read by any.
- The notation (a,b) < (c,d) defined as:

  (a < c) or (a = c and b < d)

Problem 5.7 - continued

- Describe the algorithm in words ?
- Show that this algorithm avoids deadlock ?
- Show that it enforces mutual exclusion ?

5.7 Solution

- For a process i:
  - choosing[i] = true means the process is taking a number
  - number[i] = 1 + max() means largest number assigned to process after increment.
  - choosing[i] = false means the process took a number
  - Loop to check all processes:
    - No process is in procedure of taking a number
    - For the processes holding a number, wait until you are having the lowest number. If you have number equal to other process, the process with lower index proceed.
  - proceed with the critical section
  - process finished the critical section and reset its ticket

5.7 Solution - continued

- Deadlock

  Assuming that the processes are assigned a unique index, no two processes can block on each other. Always we will have a process that proceed.

5.7 Solution - continued

- Mutual Exclusion

Let's say P1 in critical section, that means:

- P1 has found choosing[p2index] is false
  - Either P2 just entered main loop – didn't get number [OK]
  - Or P2 was already assigned a number [go to next check]
- P1 has found it got smaller number than P2, or if got the same number it already have lower index [OK]

P2 can't enter the critical section as P1 is the only satisfier to the conditions until it resets its number[p1index] to zero at end of critical section.

Problem 5.21

Any effect on program meaning if statements are as follows ?

- semWait(s);
  semWait(e);
- semSignal(n);
  semSignal(s);
- semWait(s);
  semWait(n);
- semSignal(e);
  semSignal(s);

```
/* program boundedbuffer */
const int sizeofbuffer = /* buffer size */;
semaphore s = 1;
semaphore n= 0;
semaphore e= sizeofbuffer;
void producer()
{
      while (true)
      {
            produce();
            semWait(e);
            semWait(s);
            append();
            semSignal(s);
            semSignal(n)
      }
}
void consumer()
{
      while (true)
      {
            semWait(n);
            semWait(s);
            take();
            semSignal(s);
            semSignal(e);
            consume();
      }
}
void main()
{
      parbegin (producer, consumer);
}
```

5.21 Solution

a) Deadlock might occur when buffer is full (e=0).
b) No issues, although optimally you should have the critical section with the main functionality (append/take).
c) Deadlock as the producer wont reach the signal call when the buffer is empty (n=0)
d) Same as b.

Additional Problems

Problem 5.8

```
int number[n];
while (true) {
   number[i] = 1 + getmax(number[], n);
   for (int j = 0; j < n; j++){
     while ((number[j] != 0) && (number[j],j) < (number[i],i)) { };
   }
   /* critical section */;
   number [i] = 0;
   /* remainder */;
}
```

- Modified version of Lamport Bakery algorithm without using *choosing variable. Does this violate mutual exclusion ?*

5.8 Solution

- The getmax() will have to read the number[]elements before assigning the incremented value to a process.
- Assume P0 and P1 both read the max number[] as 0 (initial value) at the same time
- P1 proceeds with value 1 and enter C.S
- P0 proceeds with value 1 and enter C.S because it has lower index and same number as P1.
- Mutual Exclusion is violated.
- The *choosing* variable with it's loop prevents a process from performing the C.S check while another process is being assigned a number.

Problem 5.12

- Any difference for using this semaphore definition compared to the one in Fig 5.3 ?

```
void semWait(s)
{
    if (s.count > 0) {
      s.count--;
    }
    else {
      place this process in s.queue;
      block;
    }
}
void semSignal (s)
{
    if (there is at least one process blocked on
        semaphore s) {
        remove a process P from s.queue;
        place process P on ready list;
    }
    else
        s.count++;
}
```

5.12 Solution

- The definition in Fig 5.3 allows negative values for the count variable, which counts the waiting processes. The provided definition doesn’t have this info.
- Using this semaphore is the same as to the one in Fig 5.3 as they function in the same way.

Helping tips for understanding semaphores

- Understand requirements of semaphores in page 210 in 8th edition or page 209 in 7th edition.
- Understand the effect of counter initialization.
- Understand the effect of Signal/Wait order

Questions ?