Operating Systems (CSE531)
Lecture # 13

Manish Shrivastava
LTRC, IIIT Hyderabad

Process Syncronization

Manish Shrivastava

Race Condition

- **Race condition**: The situation where several processes access – and manipulate shared data concurrently. The final value of the shared data depends upon which process finishes last.

- To prevent race conditions, concurrent processes must be **synchronized**.

Critical Section Problem

- *n* processes all competing to use some shared data
- Each process has a code segment, called *critical section*, in which the shared data is accessed.
- Problem – ensure that when one process is executing in its critical section, no other process is allowed to execute in its critical section.

Background

- Concurrent access to shared data may result in data inconsistency.
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes.
  - Shared-memory solution to bounded-buffer problem allows at most *n* – 1 items in buffer at the same time. A solution, where all *N* buffers are used is not simple.
    - Suppose that we modify the producer-consumer code by adding a variable *counter*, initialized to 0 and incremented each time a new item is added to the buffer

Bounded-Buffer

- Shared data

```
#define BUFFER_SIZE 10
typedef struct {
    . . .
} item;
item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
int counter = 0;
```

Bounded-Buffer

- Producer process

```
item nextProduced;

while (1) {
        while (counter == BUFFER_SIZE)
                ; /* do nothing */
        buffer[in] = nextProduced;
        in = (in + 1) % BUFFER_SIZE;
        counter++;
}
```

Bounded-Buffer

- Consumer process

```
item nextConsumed;

while (1) {
        while (counter == 0)
                ; /* do nothing */
        nextConsumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        counter--;
}
```

Bounded Buffer

- Although, both the producer and consumer routines are correct separately they may not function correctly when executed concurrently.
- The statements

  counter++;
  counter--;

  must be performed *atomically*.

- Atomic operation means an operation that completes in its entirety without interruption.

Bounded Buffer

- The statement “**counter++**” may be implemented in machine language as:

  register1 = counter

  register1 = register1 + 1
  counter = register1

- The statement “**count—**” may be implemented as:

  register2 = counter
  register2 = register2 – 1
  counter = register2

Bounded Buffer

- If both the producer and consumer attempt to update the buffer concurrently, the assembly language statements may get interleaved.
- Interleaving depends upon how the producer and consumer processes are scheduled.

Bounded Buffer

- Assume **counter** is initially 5. One interleaving of statements is:

  producer: **register1 = counter** (*register1 = 5*)
  producer: **register1 = register1 + 1** (*register1 = 6*)
  consumer: **register2 = counter** (*register2 = 5*)
  consumer: **register2 = register2 – 1** (*register2 = 4*)
  producer: **counter = register1** (*counter = 6*)
  consumer: **counter = register2** (*counter = 4*)

- The value of **count** may be either 4 or 6, where the correct result should be 5.

Solution

- A solution to critical section problem must satisfy the following conditions.
  - **Mutual Exclusion**. If process P_i is executing in its critical section, then no other processes can be executing in their critical section.
  - **Progress**. At least one process requesting entry into CS will be able to enter it if there is no other process in it..
  - **Bounded Waiting**. No process waits indefinitely to enter CS once it has requested entry.
    - Assume that each process executes at a nonzero speed
    - No assumption concerning relative speed of the *n* processes.

Approaches

- Several kernel-level processes may active at a time
  – Example: Data structure “List of open files”
- Kernel developers should ensure that OS is free from race conditions.
- Two approaches are used

Non-preemptive

- Non-preemptive kernel
  - A non-preemptive kernel does not allow a process running in the kernel mode to be preempted.
  - Kernel mode process runs until it exists kernel mode, blocks, or voluntarily yields the control of CPU
  - Free from race conditions

Preemptive

- Preemptive kernel
  - A preemptive kernel allows a process to be pre-empted while it is running in kernel mode.
  - Should be carefully designed
  - Difficult to design especially in SMP
- Why we prefer preemptive kernels ?
  - Suitable for real-time programming
  - More responsive as kernel mode process can not run for a longer time.
- WINDOWS XP, WINDOWS 2000, Prior to LINUX 2.6 are non-preemptive
- Solaris and IRIX are preemptive

Mutual exclusion: Software approaches

- Software approaches can be implemented
- Assume elementary mutual exclusion at the memory access level.
  - Simultaneous access to the same location in main memory are serialized in some order.
- Beyond this, no other support in the hardware, OS, programming language is assumed.

Two process solution : Dekker's algorithm

- Reported by Dijkstra, 1965.
- Only 2 processes, P_0 and P_1
- General structure of process P_i (other process P_j)

```
do {
    entry section
        critical section
    exit section
        reminder section
} while (1);
```

- Processes may share some common variables to synchronize their actions.

Algorithm 1

- Shared variables:
  - **int turn**;
    initially **turn = 0**
- Turn variable

P0	P1
while (turn != 0) ; ***/* Do nothing */*** ***critical section*** **turn = 1;** **remainder section**	**while (turn != 1);** ***/* Do nothing */*** ***critical section*** **turn = 0;** **remainder section**

  - Shared variable *turn* indicates who is allowed to enter next, can enter if *turn = me*
  - On exit, point variable to other process
  - Deadlock if other process never enters

- +Satisfies mutual exclusion: Only one process can enter in CS
- -It does not satisfy the progress requirement, as it requires strict alternation of processes to enter CS.
- The pace of execution is dictated by slower process.
- If turn=0, P1 is ready to enter into CS, P1 can not do so, even though P0 may be in the RS.
- If one process fails in CS or RS, other process is blocked permanently.

Algorithm 2

- Problem with Alg1
  - It does not retain sufficient information about the state of each process.
  - Alg1 remembers only which process is allowed to enter the CS.
- To solve this problem, variable turn is replaced by **boolean flag[2]**; flag[0] is for P0; and flag[1] is for P1.
- Each process may examine the other's flag but may not alter it.
- When a process wishes to enter CS, it periodically checks other's flag until that flag is false (other process is not in CS)
- The process sets its own flag true and enters CS.
- When it leaves CS, it sets its flag to false.

Algorithm 2…

- initially **flag [0] = flag [1] = false.**
- **P0** **P1**

P0	P1
while (flag[1]) ;	**while (flag[0])**
/* Do nothing */	***/* Do nothing */***
flag[0] = true;	**flag[1] = true;**
critical section	***critical section***
flag[0] = false;	**flag[1] = false;**

- Mutual exclusion is satisfied.
- If one process fails outside CS the other process is not blocked.
- Sometimes, the solution is worst than previous solution.
  - It does not even **guarantee ME.**
    - P0 executes the **while** statement and finds flag[1] set to false.
    - P1 executes the **while** statement and finds flag[0] set to false.
    - P0 sets flag[0] to true and enters its CS.
    - P1 sets flag[1] to true and enters its CS.

Algorithm 3

- Interchange the first two statements.
- Busy Flag Modified

P0	P1
flag[0] = true; while (flag[1]); */* Do nothing */* *critical section* flag[0] = false;	flag[1] = true; while (flag[0]); */* Do nothing */* *critical section* flag[1] = false;

- Guarantees ME
- Both processes set their flags to true before either has executed the **while** statement, then each will think the other has entered CS causing deadlock.

Correct solution (1)

- Combining the key ideas of previous algorithms
- Dekker's Algorithm
  - Use *flags* for mutual exclusion, *turn* variable to break deadlock
  - Handles mutual exclusion, deadlock, and starvation

Dekker’s Algorithm

- Initial state: flag[0]=flag[1]=false; turn=1

P0

```
flag[0] = true;
while ( flag[1] )
    if (turn==1)
    {
        flag[0]=false;
            while (turn==1)
                /* do nothing */
        flag[0]=true;
    }
/* critical section */
turn=1;
flag[0] = false;
remainder section
```

P1

```
flag[1] = true;
while ( flag[0])
if  (turn==0)
    {
flag[1]=false;
        while (turn==0)
                /* do nothing */
        flag[1]=true;
    }
/* critical  section */
turn=0;
flag[1] = false;
remainder section
```

Correct solution (2)

- Peterson's Algorithm
- Initial state: flag[0]=flag[1]=false;

P0	**P1**
flag[0] = true;	flag[1] = true;
turn = 1;	turn = 0;
while (flag[1] && turn==1)	while (flag[0] && turn==0)
/* Do Nothing */;	/* Do nothing */;
critical section	*critical section*
flag[0] = false;	flag[1] = false;
remainder section	***remainder section***

Correct solution

- We need to show that
  - ME is preserved
  - The progress requirement is satisfied
  - The bounded-waiting requirement is met.
- **ME is preserved**
  - If both processes enter the CS both flad[0]==flag[1]==true
  - Both could not execute while loop successfully as turn is either 0 or 1.
- **Progress.**
  - While P1 exits CS it sets flag[1]=false, allowing P0 to enter CS.
  - P1 and P0 will enter the CS (Progress)
- **Bounded waiting:** P1 will enter the CS after at most one entry by P0 and vice versa.

Multi-process solution: Bakery Algorithm

- Based on scheduling algorithm commonly used in bakeries.
  - On entering the store the customer receives the number.
  - The customer with the lowest number is served.
  - Customers may receive the same number, then the process with the lowest name is served first.
- Before entering its critical section, process receives a number. Holder of the smallest number enters the critical section.
- If processes P_i and P_j receive the same number, if $i < j$, then P_i is served first; else P_j is served first.
- The numbering scheme always generates numbers in increasing order of enumeration; i.e., 1,2,3,3,3,3,4,5...

Bakery Algorithm

- var: choosing: array[0…n-1] of boolean.
- Notation <≡ lexicographical order (ticket #, process id #)
  - $(a,b) < c,d)$ if $a < c$ or if $a = c$ and $b < d$
  - max $(a_0,\dots, a_{n-1})$ is a number, k, such that $k \geq a_i$ for i =0, …, $n - 1$
- Shared data

  boolean choosing[n];

  int number[n];

  Data structures are initialized to **false** and **0** respectively

Bakery Algorithm

```
do {
        choosing[i] = true;
        number[i] = max(number[0], number[1], …, number [n – 1])+1;
        choosing[i] = false;
        for (j = 0; j < n; j++) {
                    while (choosing[j]) ;
                    while ((number[j] != 0) && (number[j,j] < number[i,i])) ;
        }
            critical section
        number[i] = 0;
            remainder section
} while (1);
```

- **Consider Pi in its CS and Pk is trying to enter CS**
- When Pk enters second while statement for j=I, it finds that
  - number[i] $\neq 0$
  - (number[i],i) < (number[k].k)
  - So it leaves until Pi leaves CS
- FCFS is followed.

Mutual exclusion: hardware solution

- In the uni-processor system, it is sufficient to prevent a process from being interrupted.

```
while (true){
/* disable interrupts */
/* Critical section */
/* enable interrupts */
/* remainder */
}
```

- **Since CS can not be interrupted ME is guaranteed.**
- **The efficiency decreases**
- **It can not work in multi-processor environments**
    - **More than one process is executing at a time.**

Special machine instructions

- In multi-processor configuration, several processes share access to a common main memory.
- At the hardware level, access to a memory location excludes any other access to that same memory location.
- Processor designers have proposed several machine instructions to carry out two actions atomically (single cycle).
  - **Reading and writing**
  - **swapping**

Test and set instruction

- Test and modify the content of a word atomically

```
boolean testset  (int    i)
{
 if (i==0)
   {
       i=1;
       return true;
   }
   else
     {
        return false
      }
}
```

- This instruction sets the value of ‘i’, if the value=0 and returns true. Otherwise the value is not changed and false is returned.

Mutual Exclusion with Test-and-Set

- Shared data:
  boolean lock = false;
- void *P(int i)*

```
do {
        while (TestAndSet(lock)==false)
            /* do nothing*/;
            critical section
        lock = false;
            remainder section
  }
```

Test-and-Set: Correctness

- **Mutual exclusion**
  - A shared variable lock is set to false
  - The only process Pi that enters CS that finds lock as false and sets it to true.
  - All other processes trying to enter CS so   into a busy waiting mode and finds lock as false.
  - When process leaves C it resents lock to false.
  - When Pi exits lock is set to false so the next process Pj to execute instruction find test-and-set=false and will enter the CS.
- **Progress**
  - **Trivially true**
- **Unbounded waiting**
  - **Possible since depending on the timing of evaluating the test-and-set primitive.**
  - **Does not guarantee fairness.**

Swap instruction

- Atomically swap two variables.

```
void swap(boolean &a, boolean &b) {
  boolean temp = a;
  a = b;
  b = temp;
}
```

Mutual Exclusion with Swap

- Shared data (initialized to **false**):

```
boolean lock;
boolean waiting[n];
```

- Process P_i

```
do {
    key = true;
    while (key == true)
            Swap(lock,key);
        critical section
    lock = false;
        remainder section
}
```

SWAP: Correctness

- **Similar to Test-and-set**
- **Mutual exclusion**
- **Progress**
  - **Trivially true**
- **Unbounded waiting**
  - **Possible since depending on the timing of evaluating the test-and-set primitive.**
  - **Does not guarantee fairness.**

Can we get bounded waiting ?

- **Introduce a boolean array called waiting of size n and boolean variable key**
- **Entry**
  - **waiting[i]:=true;**
  - **key:=true;**
  - **while (waiting[i] and key) do**
    - **Swap(&key,&lock)**
  - **waiting[i]:=false;**
  - **execute CRITICAL SECTION**
- **Exit**
  - **Find the next process j that has waiting[j]=1 stepping through waiting.**
  - **Set waiting[j]:=false;**
  - **Process P_j immediately enter the CS.**
  - **If no process exists, set lock=false;**

Can we get bounded waiting ?....

- **Every (interested) Pi executes test&set at least once.**
- **Pi enters the critical section provided:**
  - **Key is false in which case there is no process in CS.**
- **Or**
  - **If it was waiting, because waiting[i] was reset to false by the unique process that was blocking it in the critical section.**
  - **Either of the above events occur exactly once and hence mutual exclusion.**

Properties of machine instruction approach

- +ve
  - Any number of processes
  - Simple and easy
  - Can support multiple CSs.
- -ve
  - Busy waiting is employed
    - The process is waiting and consuming processor time.
  - Starvation is possible.
    - The selection of waiting process is arbitrary.
  - Deadlock is possible due to priority
    - P1 enters CS and interrupted by higher priority process P2 which is trying to enter CS.
    - P2 can not get CS unless P1 is out and P1 can not be dispatched due to low priority.

Semaphores: Dijkstra; 1965

- Two and more processes can cooperate by means of simple signals, such that a process is forced to stop at a specified place until it has received a specific signal.
- For signaling, special variables called semaphores are used
- A semaphore is a synchronization tool.
- A semaphore is an integer variable that is accessed only through two standard atomic operations: **wait and signal**.
- To transmit a signal to semaphore S, a process executes the primitive *signal(S)* primitive.
- To receive a signal via semaphore S, the process executes *wait(S)* primitive.

Semaphores: Dijkstra 1965
Classical or first definition

- A semaphore is initialized to a non-negative value
- The **wait** operation decrements the semaphore value. If the integer value is negative the process waits.
- The **signal** operation increments the semaphore value. If the value is not positive, then process which is blocked by a wait operation is gets the access to CS.
- The wait and signal are assumed to be atomic.
- Semaphore *S* – integer variable
- can only be accessed via two indivisible (atomic) operations

wait (*S*):

while $S \leq 0$ **do** ***no-op*;**

***S*--;**

signal (*S*):

S++;

Critical Section of *n* Processes

- Shared data:

  semaphore mutex; // initially *mutex* = 1

- Process *Pi:*

```
do {
    wait(mutex);
        critical section
    signal(mutex);
        remainder section
} while (1);
```

- **Modifications to the integer value of the semaphore in the wait and signal operations must be executed indivisibly.**

Semaphore Implementation

- The classical definition requires busy waiting.
- While a process is in CS, the other process must loop continuously in the entry code.
- Busy waiting wastes CPU cycles.
- This type of semaphore is called spinlock: process spins while waiting for a lock.
  - Advantage of spinlock: no context switch
  - When locks are expected to be held for short times, spinlocks are useful.
- To overcome the need for busy waiting, we can modify the definition of the wait and signal semaphore operations.
- If a process executes wait operation and finds the semaphore operation is not positive, it must wait.
  - Rather than busy waiting it must block itself.
  - The **block** operation puts the process into waiting queue of semaphore and process is switched to waiting state.
- A process that is blocked waiting on a semaphore S, should be restarted when some other process executes signal operation.
- The process is restarted with **wakeup** operation.

Semaphore Implementation

- Define a semaphore as a record

```
typedef struct {
    int value;
    struct process *L;
} semaphore;
```

- Assume two simple operations:
  - **block** suspends the process that invokes it.
  - **wakeup(*P*)** resumes the execution of a blocked process **P**.

Implementation

- Semaphore operations now defined as

wait(*S*):

```
S.value--;
if (S.value < 0) {
        add this process to S.L;
        block;
}
```

signal(*S*):

```
S.value++;
if (S.value <= 0) {
        remove a process P from S.L;
        wakeup(P);
}
```

- **Wait and signal operations are system calls.**

Semaphore as a General Synchronization Tool

- Execute *B* in P_j only after *A* executed in P_i
- Use semaphore *flag* initialized to 0
- Code:

P_i	P_j
$\vdots$	$\vdots$
A	*wait*(*flag*)
signal(*flag*)	*B*

Deadlock and Starvation

- **Deadlock** – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes.
- Let *S* and *Q* be two semaphores initialized to 1

P_0	P_1
wait(*S*);	*wait*(*Q*);
wait(*Q*);	*wait*(*S*);
⋮	⋮
signal(*S*);	*signal*(*Q*);
signal(*Q*)	*signal*(*S*);

- **Starvation** – indefinite blocking. A process may never be removed from the semaphore queue in which it is suspended.

Two Types of Semaphores

- *Counting* semaphore – integer value can range over an unrestricted domain.
- *Binary* semaphore – integer value can range only between 0 and 1; can be simpler to implement.

Binary Semaphores

- A binary semaphore is a semaphore with an integer value that can range only between 0 and 1
- It is simple to implement.
- Type binary semaphore =**record**

  value:(0,1)

  queue: list of processes

  end;
- var s: binary semaphore
- **waitB(s):**

If s.value=1 **then**

s.value=0

else

begin

place this process in s.queue;

block this process;

end;

- **signalB(s):**

If s.queue is empty **then**

s.value=1

else

begin

remove a process from s.queue;

place this process in the ready list.

end;

Implementing *S* as a Binary Semaphore

- Can implement a counting semaphore *S* as a binary semaphore.
- Data structures:

  binary-semaphore S1, S2;

  int C:
- Initialization:

  S1 = 1

  S2 = 0

  C = initial value of semaphore **S**

  – *wait* operation

```
wait(S1);
 C--;
 if (C < 0) {
        signal(S1);
        wait(S2);
              }
 signal(S1);
```

  – *signal* operation

```
 wait(S1);
 C ++;
 if (C <= 0)
   signal(S2);
else
signal(S1);
```

Counting semaphores

***wait*(*S*):**

```
S.value--;
if (S.value < 0) {
      add this process to S.L;
      block;
   }
```

***signal*(*S*):**

```
S.value++;
if (S.value <= 0) {
   remove a process P  from
S.L;
  wakeup(P);
  }
```

Implementing wait() and signal() in Multi-processor Systems

- Disabling interrupts will not work.
- Spinlock is the solution
- With this we have moved busy waiting from entry section to critical sections of application programs.