Process Synchronization (CS-351)

Agenda

- What is process synchronization?
- Producer Consumer Problem: Race Condition
- The critical region problem.
- Software Solutions: Semaphores, Peterson's solution
- Hardware Solutions: TestAndSet(), Swap()
- The Deadlock, Starvation, Priority inversion problems

What is Process Synchronization?

- Cooperating processes can share data.
- Concurrent access to data may result in data inconsistency.
- Process synchronization: ensuring an orderly execution of cooperating processes that share data, in order to maintain data consistency.
  - Necessary to prevent race conditions.
- Race condition: when multiple processes (or threads) access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place.
- Recall the bounded buffer implementation of the producer consumer problem (next slide).

Race Conditions: Producer Consumer Problem

- Bounded buffer implementation (a wrap-around buffer):
  - Store the following variables in shared memory:

```
#define BUFFER_SIZE 10
typedef struct {
   . . .
} item;
item buffer[BUFFER_SIZE];
int counter = 0;  //Keeps track of the # of available slots.
int in = 0;    //First empty position in "buffer"
int out = 0; //First full position in "buffer"
//The buffer is empty when in == out
//The buffer is full when ((in+1) % BUFFER_SIZE) == out.
```

Race Conditions: Producer Consumer Problem

Producer code:

```
while (true)
{
    /* do nothing -- no free buffers */
    while ((( (in + 1) % BUFFER_SIZE == out);
     //Produce an item
      …..
      //Save the produced item
      buffer[in] = item;
      //Compute the next free index
      in = (in + 1) % BUFFER_SIZE;
      counter++;
 }
```

Consumer code:

```
while (true)
{
    //No items to consume
     while (in == out);
      // Consume an item
     item = buffer[out];
     // Compute the index of the next
     // item to consume
     out = (out + 1) % BUFFER SIZE;
     counter--;
     return item;
   }
```

Race Conditions: Producer Consumer Problem

- Machine code for updating the counter:

Producer code:

PI_1. $register_1$ = counter

PI_2. $register_1$ = $register_1$ + 1

PI_3. counter = $register_1$

Consumer code:

CI_1. $register_2$ = counter

CI_2. $register_2$ = $register_2$ - 1

CI_3. counter = $register_2$

- PI = producer instruction
- CI = consumer instruction
- The concurrent execution of the above codes, may correspond to many different sequential executions.
- The final value of the counter depends on the order of execution.

Race Conditions: Producer Consumer Problem

Producer code:

PI_1. $register_1$ = counter

PI_2. $register_1 = register_1 + 1$

PI_3. counter = $register_1$

Consumer code:

CI_1. $register_2$ = counter

CI_2. $register_2 = register_2 - 1$

CI_3. counter = $register_2$

PI = producer instruction

CI = consumer instruction

- PI_1: producer: $register_1$ = counter {$register_1$ = 5}
- PI_2: producer: $register_1 = register_1 + 1$ {$register_1$ = 6}
- CI_1: consumer: $register_2$ = counter {$register_2$ = 5}
- CI_2: consumer: $register_2 = register_2 - 1$ {$register_2$ = 4}
- PI_3: producer: counter = $register_1$ {counter = 6 }
- CI_3: consumer: counter = $register_2$ {counter = 4}
- What if the order of PI_3 and CI_3 is swapped (next slide)?

Race Conditions: Producer Consumer Problem

Producer code:

PI_1. $register_1 = counter$

PI_2. $register_1 = register_1 + 1$

PI_3. $counter = register_1$

Consumer code:

CI_1. $register_2 = counter$

CI_2. $register_2 = register_2 - 1$

CI_3. $counter = register_2$

PI = producer instruction

CI = consumer instruction

- PI_1: producer: $register_1 = counter$ $\{register_1 = 5\}$
- PI_2: producer: $register_1 = register_1 + 1$ $\{register_1 = 6\}$
- CI_1: consumer: $register_2 = counter$ $\{register_2 = 5\}$
- CI_2: consumer: $register_2 = register_2 - 1$ $\{register_2 = 4\}$
- CI_3: consumer: $count = register_2$ $\{counter = 4\}$
- PI_3: producer: $counter = register_1$ $\{counter = 6\}$
- The value of counter is now 6 (4 in the previous slide)!

The Critical Section Problem

- Critical section: a segment of process **code**, in which the process modifies shared resources e.g. updating a table or writing to a file.
- No two processes may execute in the critical section at the same time e.g. no two processes may update the same variable at the same time.
- The critical section problem: ensuring that only one process is executing in its critical section.

The Critical Section Problem: Solution Requirements

- Requirements for solving the critical section problem:
  - Mutual exclusion: only one process may enter a critical section at a time.
  - Progress: no process executing outside of its critical section, may block other processes from entering theirs.
  - Bounded waiting: process cannot be perpetually barred by other processes, from entering its critical section.
- Existing solutions:
  - Peterson's solution: software-based solution.
  - Synchronization hardware: hardware-based solution.
  - Semaphores: specialized integer variables.

Semaphores

- Are easier to use than hardware-based synchronization.
- An integer variable S, accessible only through wait() and signal() operations: for example:

```
wait(S)
{
    //Do nothing
    while( S <= 0);

    S--;
}
```

```
signal(S)
{
        S++;
}
```

- All modifications to S are atomic.

Semaphores: Different Types

- Binary semaphore:
  - The value can range only between 0 and 1.
  - Also known as mutex (i.e. mutual exclusion) locks.
  - The mutex is initialized to 1.
  - Processes use the mutex as follows:

```
// Code of process P
do
{
   wait(mutex);

    // Critical section

    signal(mutex);

    // Remainder section

 } while (TRUE);
```

```
wait(S)
{
     //Do nothing
     while( S <= 0);

     S--;
}
```

```
signal(S)
{
          S++;
}
```

Student Participation: Disadvantages of Mutex (Discussion)

- What are the possible issues of using mutex?

Semaphores: Implementation

- The following implementation of mutex requires busy waiting i.e. constantly polling S in a **while loop**:

```
wait(S)
{
    //Do nothing
    while( S <= 0);

    S--;
}
```

- Fine if the critical section is short; otherwise it is inefficient.
- If there are **multiple instances** of the same resource, the other processes still need to wait.
- There are alternatives to busy waiting (next slide).

Semaphores: Different Types

- Counting Semaphore:
  - The value is unrestricted.
  - Useful for coordinating access to a resource with finite number of instances.
  - The semaphore is initialized to the maximum # of instances.
  - When the process calls wait() it decrements the value of S:
    - If S remains positive, then the process may continue executing.
    - Otherwise, the process waits in a loop until S becomes positive.
  - When the process calls signal(), it increments the value of S.

Semaphores: Counting Semaphore Implementation

- With each semaphore associate a wait queue of processes (represented as linked list).
- Each entry in queue comprises:
  - Value (of type integer)
  - Pointer to next record in the list

```
//Code of the semaphore
typedef struct
{  int value;
   struct process *list;
} semaphore;
```

- Redefine wait() and signal() functions as follows:

```
wait(semaphore *S)
 {   S->value--;
     if (S->value < 0)
     {  add this process to S->list;
        block();
      }
}
```

```
signal(semaphore *S)
{   S->value++;
     if (S->value <= 0)
     {    remove a proc. P from S->list;
          wakeup(P);
     }
}
```

- wait(): if S->value is non-positive: the calling process is added to the list and is blocked (switched to waiting state).
- wakeup(): if S->value <= 0, then a process is removed from the list and is woken up i.e. allowed to execute.

Student Participation: Properties of the semaphores solution to the CS problem

- 1) The semaphore mutex can only have the values 0 or 1.
  - True
  - False
- 2) When each process runs on a separate CPU then mutual exclusion cannot be guaranteed.
  - True
  - False
- 3) If p1 runs on a faster CPU than the other processes then p1 can execute CS multiple times before another process gets a turn.
  - True
  - False

Deadlocks

- Multiple processes waiting indefinitely for an event that can be triggered by only one of the waiting processes.
- Example: Let S and Q be two semaphores, set to the value of 1:

P_0	P_1
wait (Q);	wait (S);
wait (S);	wait (Q);
.	.
.	.
.	.
signal(S)	signal(Q)
signal(Q)	signal(S)

- What can possibly go wrong?

Starvation

- A process is never removed from the semaphore queue in which it waits.
- May happen if the other processes keep grabbing the shared resource before the queued process.

Priority Inversion

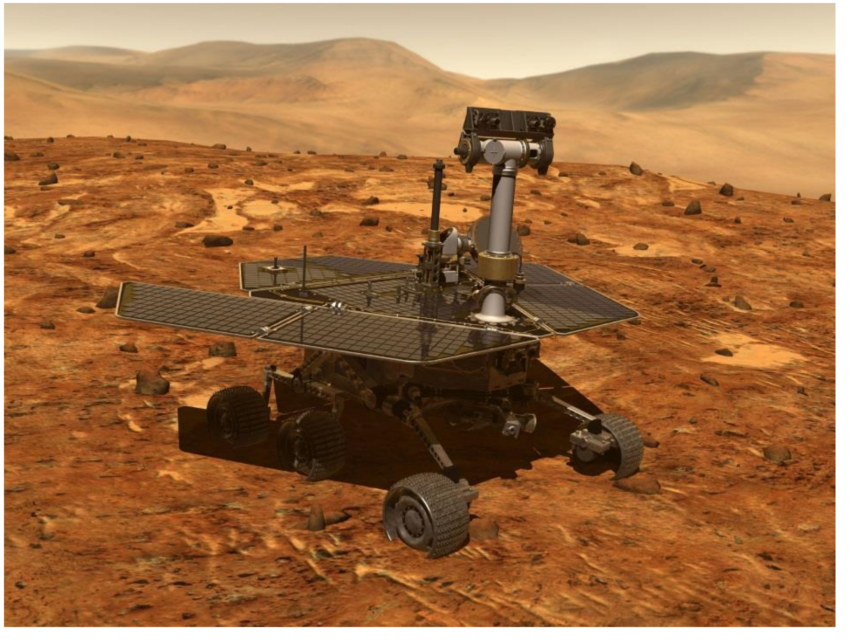
- When a lower priority process holds a lock and prevents a higher priority process from acquiring the lock (so it may access a critical region).
- Example: 3 processes L, M, H with priority: L < M < H.
  - H waits for a lock held by L, meanwhile process M becomes runnable (and does not require a lock).
  - Since M is the only non-blocked task, it will be scheduled to run and preempt L (L cannot finish its critical region unless it runs)! Hence, H has to wait until L is scheduled to run again, finishes the critical section, and releases the lock!
  - Solution: while L has holds lock, let it inherit the priority of H so it cannot be preempted. This is called priority inheritance protocol.

Priority Inversion: Priority Inheritance

- If a higher priority process waits for a lower priority process to release the lock, the lower priority process temporarily inherits the priority of the waiting process.
- When the process releases the lock, its priority is reverted to the original.

Priority Inversion

- Example: The Mars Pathfinder (1997) Mission:
  - The Sojourner rover would frequently hang because
    - A high priority task called bc_dist was blocked by a lower priority task "ASI/MET", waiting to use a shared resource.
    - ASI/MET, in turn, was pre-empted by multiple medium priority tasks.
  - Solution: enable priority inheritance on all semaphores (change a global variable in the VxWorks OS).



The Critical Section Problem: Peterson's Solution

- A software-based solution.
- Assumes two processes (P_i and P_j).
- Assumes that the LOAD and STORE operations are atomic i.e. they cannot be interrupted.
  - **Not** a reasonable assumption on modern architectures.
- Two processes share the following variables:
  - int turn;
  - bool flag[2]
- turn: indicates whose turn it is to enter the critical section.
- flag: indicates if a process is ready to enter the critical section.
- flag[i] = true implies that process P_i is ready to enter it's critical section!

The Critical Section Problem: Peterson's Solution

- The two processes (P_i and P_j) share the following variables:
  - int turn: whose turn it is to enter the critical section?
  - bool flag[2]: if a process is ready to enter the critical section?
  - flag[i] = true implies that process P_i is ready!

Code of process P_i

```
do {
        flag[i] = TRUE;
            //j = i - 1
            turn = j;
            while (flag[j] && turn == j);
                        critical section
            flag[i] = FALSE;
                        remainder section
    } while (TRUE);
```

Code of process P_j

```
do {
        flag[j] = TRUE;
            //i = j - 1
            turn = i;
            while (flag[i] && turn == i);
                        critical section
            flag[j] = FALSE;
                        remainder section
    } while (TRUE);
```

The Critical Section Problem: Peterson's Solution

- Observation:
  - Process P_i sets flag[i] to true, indicating that it's ready to enter it's critical section.
  - Process P_i sets turn to j, indicating that P_j may enter its critical section if it wishes to do so.
- Result:
  - If both processes try to enter their critical sections at the same time, then turn will be set to both i and j at roughly the same time.
  - The final value if turn will be determined by the process that gets to update turn last, thus preventing both processes from entering their critical section simultaneously.

The Critical Section Problem: Synchronization Hardware

- Solution for uniprocessor systems: disable interrupts while modifying a shared variable - if no other instructions are being executed, then nothing can interfere with the variable modification operation.
  - Approach taken by nonpreemptive kernels.
  - Difficult on multiprocessor systems.
- Special atomic hardware instructions:
  - Must be atomic i.e. once started, always completes.
  - TestAndSet(): can atomically test and then set the value of a variable.
  - Swap(): can atomically swap the contents of two variables.

The Critical Section Problem: Synchronization Hardware

- TestAndSet() instruction code:

```
boolean TestAndSet (boolean *target)
{
    boolean rv = *target;
    *target = TRUE;
    return rv:
}
```

- Remember: the whole function is guaranteed to be atomic i.e. non-interruptible.

The Critical Section Problem: Synchronization Hardware

- Using TestAndSet() to provide synchronization:
  - Assume both processes share the variable lock:

Code of process P_i

```
do {
      // Do nothing - wait
      while (TestAndSet (&lock ));

       // Critical section
          ...

      lock = FALSE;

       //  Remainder section
       ...

   } while (TRUE);
```

Code of process P_j

```
do {
      // Do nothing - wait
      while (TestAndSet (&lock ));

       // Critical section
          ...

      lock = FALSE;

       //  Remainder section
       ...

   } while (TRUE);
```

The Critical Section Problem: Synchronization Hardware

- Swap() instruction code:

```
void Swap (boolean *a, boolean *b)
{
     boolean temp = *a;
    *a = *b;
    *b = temp:
}
```

- Remember: the whole function is guaranteed to be atomic i.e. non-interruptible.

The Critical Section Problem: Synchronization Hardware

- Using Swap() to provide synchronization:
  - Both processes share a lock variable (initialized to false).
  - Each process has a local variable myWait.

Code of process P_i

```
do
{
    myWait = TRUE;

    while ( myWait == TRUE)
      Swap (&lock, &myWait );

    // Critical section
    lock = FALSE;

   //  Remainder section
 } while (TRUE);
```

Code of process P_j

```
do
{
    myWait = TRUE;

    while ( myWait == TRUE)
      Swap (&lock, &myWait );

    // Critical section
    lock = FALSE;

   //  Remainder section
 } while (TRUE);
```