CIS 415
Operating Systems
Concurrency and Synchronization

Prof. Allen D. Malony
Department of Computer and Information Science
Spring 2020

UNIVERSITY OF OREGON

Logistics

- Congratulations on Project 1!
- Project 2 to be posted very soon
  - Due Tuesday, May 12, 11:59pm
  - Try to make good progress in the next 2 weeks
  - Lab sessions (3 of them) will be focused on project
  - Reserve 5 minutes at end of class to discuss
- Lab 4 posted
  - For future labs (including Lab 4) solutions will be provided after everyone's work has been evaluated
- For future labs and projects, we will schedule an Zoom meeting after the lab/project is assigned to have discussion about things that are unclear and any other questions that you might have
- Read Chapter 6
- Slides for Lecture 7 and 8 are combined (have been updated)
- Midterm exam in 2 weeks

Outline

- Background
- Critical-Section Problem
- Peterson's Solution
- Synchronization Hardware
- Mutual Exclusion (Mutex) Locks
- Semaphores
- Monitors
- Classic Problems of Synchronization

Objectives

- To present the concept of process synchronization
- To introduce the critical-section problem, whose solutions can be used to ensure the consistency of shared data
- To present both software and hardware solutions of the critical-section problem
- To examine several classical process-synchronization problems
- To explore several tools that are used to solve process synchronization problems

Roadmap

Concurrent Applications

Shared Objects

Bounded buffer Barrier

Synchronization Variables

Semaphores Locks Condition Variables

Atomic Instructions

Interrupt disable Test-and-Set Compare-and-Swap

Hardware

Multiple processors Hardware interrupts

Background

- Early operating systems research was where fundamental problems of *concurrency* first arose
- "Cooperating Sequential Processes," E.W. Dijkstra, Technical Report, TU Eindhoven, the Netherlands,1965
  - See link on schedule (appears in a chapter)
  - Dijkstra was the 1972 Turing Award winner!
  - This paper introduced the critical section problem

- OS must function as a concurrent system … Why?
- Concurrent systems must address problems of how concurrent processes work consistently together

Concurrency and Synchronization

- Processes can execute concurrently (logically, physically)
  - May be interrupted at any time, partially completing execution
  - OS must concurrently execute its own software!!!
- There are different kinds of resources that are shared between processes:
  - Physical (terminal, disk, network, …)
  - Logical (files, sockets, memory, …)
  - Memory
- Concurrent access to shared resources must be done in a consistent manner or else errors arise
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
- This is the role of synchronization

Resources

- There are different kinds of resources that are shared between processes:
  - Physical (terminal, disk, network, …)
  - Logical (files, sockets, memory, …)
- For the purposes of this discussion, let us focus on "memory" as the shared resource
  - Processes can all read and write into memory
  - Suppose multiple processes share memory
    - use IPC shared memory segments mechanisms
  - It might be easier to think of multiple threads
    - sharing memory is natural

Example Problem Due to Sharing

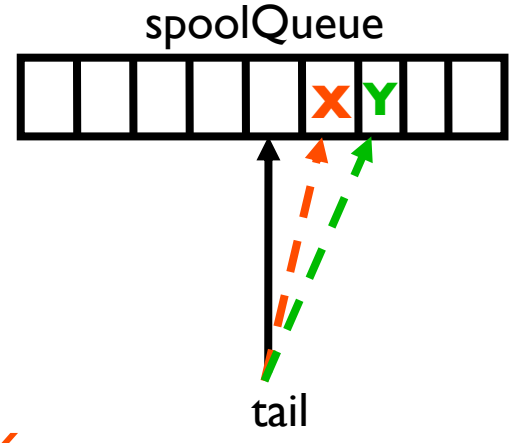
- Consider a shared printer queue
  - *spoolQueue[N]*
- 2 processes want to enqueue an element each to this queue
- *tail* points to the current end of the queue
  - It is also a shared variable
- Each process needs to do

  tail = tail + 1;

  spoolQueue[tail] = "element";

What are trying to do?

- Want to have "consistent" and "correct" execution



Process 1

tail = tail + 1;
spoolQueue[tail] = X

Process 2

tail = tail + 1;
spoolQueue[tail] = Y

- What could go wrong?

What is the problem?

- *tail = tail + 1* is NOT a single machine instruction
  - So, what? Why do we care?
- What assembly code does the compiler produce?

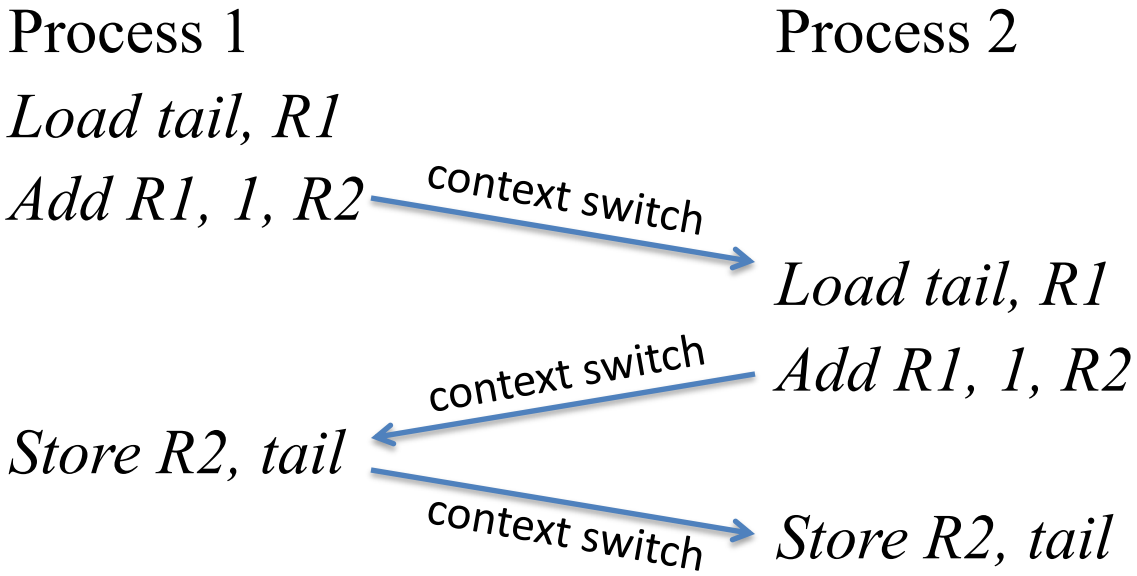
  Load tail, R1

  Add R1, 1, R2

  Store R2, tail
- These 3 machine instructions might NOT be executed *atomically* … Why not?
  - To execute atomically means to execute multiple instructions logically together as if they were a single instruction without being interrupted
- So, what is the problem?

Interleaving

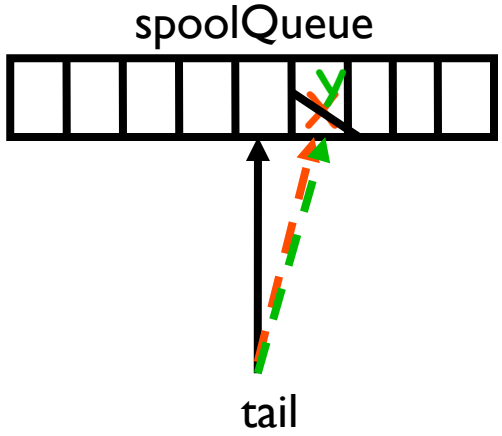
- Each process executes this set of 3 instructions
- Interrupts might happen at any time!!!
  - Thus, a context switch can happen at any time
  - Uh, ok, so what?
- Suppose we have the following scenario:



See any problems?

Leading to …

- Incorrect execution



Process 1

tail = tail + 1;
spoolQueue[tail] = X

Process 2

tail = tail + 1;
spoolQueue[tail] = Y

- This is not what we want
- Hmm, how can we fix it?

Race Conditions

- Situations like this that can lead to erroneous execution are called *race conditions*
- Definition: a *race condition* in concurrent execution is when the outcome of the execution depends on the particular interleaving of concurrent instructions
- Debugging race conditions can be fun! Not!
- Race conditions are timing dependent!
  - Errors can be non-repeatable (not fun!)
- Race conditions CAN cause the "state" of the execution to be inconsistent (incorrect)
  - It does not mean that because there is a race condition that the state WILL become inconsistent

Ok, how do we avoid race conditions?

- Definition: A set of instructions is *atomic* if it executed *as if* it was a single instruction (logically)
- What does this mean exactly?
  - Executing the instructions can not be interrupted?
  - It has more to do with the outcomes of executing the instructions with respect to other processes
- Suppose the 3 assembly instructions we were looking at were atomic
- Does this avoid the race condition?
- Definition: When executing a set of instructions is vulnerable to a race condition, that set of instructions are said to constitute a *critical section*

Critical Section Problem (Dijkstra, 1965)

- Consider system of *n* processes $\{P_0, P_1, \ldots P_{n-1}\}$
- Each process has critical segment of code
  - Process may be changing common variables, updating table, writing file, … (in its critical section)
  - When one process is in its critical section, no other may be in its critical section (*mutual exclusion*)
- *Critical section problem* is to design a *protocol* between the processes to solve this
  - Each process must enter the critical section (*entry section)*
  - Each process then executes the critical section instructions
  - Each process must exit the critical section (*exit section)*
  - Each process executes outside the critical section

Critical Section

- General structure of process P_i

```
do {
    code outside critical section
    code to enter the critical section
        CRITICAL SECTION
    code to exit from critical section
    code outside critical section
} while (true);
```

Only 1 process can be in critical section at a time

It is the entry and exit code that defines the critical section protocol.

The infinite do loop is just to suggest that a process will possibly want to enter its critical section multiple #s of times, including never (0 times) or just once (1 time).

Requirements for Solution to CS Problem

- *Mutual exclusion*
  - If process P_i is executing in its critical section, then no other processes can be executing in their critical sections
- *Progress*
  - If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely
- *Bounded waiting*
  - A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted
  - Assume that each process executes at a nonzero speed
  - No assumption concerning relative speed of the *N* processes

How to Implement Critical Sections

- Implementing critical section solutions follows three fundamental approaches

1. Disable Interrupts
   - Effectively stops the scheduling of other processes
   - Does not allow another process to get the CPU
2. Busy-wait/spinlock solutions
   - Pure software solutions
   - Integrated hardware-software solutions
3. Blocking Solutions

- Think about whether the critical section solution requirements are being met

Disabling Interrupts

- We already know how to prevent another process from interrupting the current running process
  - Do not allow interrupts to occur by disabling them
- Advantages:
  - Simple to implement (single instruction)
- Disadvantages:
  - Do not want to give such power to user processes
  - Does not work on a multiprocessor … Why?
  - Disables multiprogramming even if another process is NOT interested in critical section

Note: WHILE Loops

- In the following slides, WHILE loops with empty statements are use to mean infinite loops until the WHILE condition is met
- These examples are equivalent

```
while (lock == TRUE) {
    ;
}
```

```
while (lock == TRUE)
    ;
```

```
while (lock == TRUE) ;
```

; represents a NULL instruction

Busy Waiting (aka Spinning)

- Overall philosophy:
  - Keep checking some state (variables) until they indicate other process(es) are not in critical section

```
locked = FALSE; // initial value

P1 {                                    P2 {

while (locked == TRUE)                  while (locked == TRUE)
    ;                                       ;
locked = TRUE;                          locked = TRUE;

/************                           /************
(critical section code)                 (critical section code)
/************                           /************

locked = FALSE;                         locked = FALSE;

}                                       }
```

- Is there an interleaving where this fails?

Remember, P1 and P2 might do this multiple #s of times, including 0.

Reading, Writing, and Testing Locks

- Is the instruction below atomic?

```
while (locked == TRUE)
```

A: load locked, R1
cmp R1, 1
beq A

- How about these instructions?

```
locked = TRUE;

locked = FALSE;
```

- Generally, if the high-level statement compiles to a single machine instruction, it is atomic
- Need reading, writing, testing of locks to be atomic

Try Strict Alternation

- Consider this code
- Idea is to take turns using the critical section
  - Variable *turn* is used for this
- Does it work?
- What problems do you see?
  - Is there mutual exclusion?
  - Is there progress?
  - Is there bounded waiting?

Remember, P1 and P2 might do this multiple #s of times, including 0.

```
turn = 0; // initial value
P0 { // proces P0
  while (turn != 0);
    /*********/
    critical section
    /*********/
  turn = 1;
}
P1 { // proces P1
  while (turn != 1);
    /*********/
    critical section
    /*********/
  turn = 0;
}
```

Fixing the "progress" requirement

- What about this code?
- Each process has a flag to say that they want to enter the critical section
- Problems?
- Deadlocked!
  - They got mutual exclusion, for sure!
- For this reason, it does NOT meet the progress or bounded waiting requirements either

Remember, P1 and P2 might do this multiple #s of times, including 0.

```
bool flag[2]; // initialize to FALSE

P0 {
    flag[0] = TRUE;
    while (flag[1] == TRUE)
        ;

    /* critical section */

    flag[0] = FALSE;
}

P1 {
    flag[1] = TRUE;
    while (flag[0] == TRUE)
        ;

    /* critical section */

    flag[1] = FALSE;
}
```

Peterson's Solution

- Consider 2 processes
- Assume that the LOAD and STORE instructions are atomic and cannot be interrupted
- The two processes share two variables:
  - *int turn;*
  - *boolean flag[2]*
- Variable *turn* indicates whose turn it is to enter the critical section
- The *flag* array is used to indicate if a process is ready to enter the critical section
  - *flag[i] = true* implies that process P_i is ready!

Algorithm for Process P_i and Process P_j

Process P_i

```
while (TRUE) {
   . . .

   flag[i] = TRUE;
   turn = j;
   while (flag[j]&&turn == j);

      critical section

   flag[i] = FALSE;

   . . .
}
```

Process P_j

```
while (TRUE) {
   . . .

   flag[j] = TRUE;
   turn = i;
   while (flag[i]&&turn == i);

      critical section

   flag[j] = FALSE;

   . . .
}
```

The infinite do loop is just to suggest that a process will possibly want to enter its critical section multiple #s of times, including 0 times and 1 time.

Does Peterson's Solution work?

- Prove that the 3 CS requirements are met:
  - Mutual exclusion is preserved
    P_i enters CS only if:
    either *flag[j] = false* or *turn = i*
    - if both processes are interested in entering, then 1 condition is false for one and true for the other process
  - Progress requirement is satisfied
    - a process wanting to enter will be able to do so at some point
    - Why?
  - Bounded-waiting requirement is met
    - eventually it will be P_i's turn if P_i wants to enter

Multiple Processes

- Peterson's solution ONLY works for 2 process solutions
- How do we extend for multiple processes?
- Is there a way to modify Peterson's approach?
- Consider the following enter /exit routines:

```
int turn;
int flag[N]; /* all set to FALSE initially */

enterCS(int myid) {
  otherid = (myid+1) % N;
  turn = otherid;
  flag[myid] = TRUE;
  while (turn == otherid && flag[otherid] == TRUE) ;
  /* proceed if turn == myid or flag[otherid] == FALSe
  */
}
leave_CS(int myid) {
  flag[myid] = FALSE;
}
```

Remember, processes might do this multiple #s of times, including 0.

Bakery Algorithm (Leslie Lamport, 1974)

- We need to enforce a sequence in some manner that everyone will follow and contribute to making progress
- Think about a bakery
- See link to paper in schedule

```
Notation: (a,b) < (c,d) if a<c  or  a=c and b<d

Every process has a unique id (integer) Pi

bool choosing[0..n-1]; /* all set to FALSE */
int number[0..n-1]; /

enter_CS(myid) {
  choosing[myid] = TRUE;
  number[myid] =
    max(number[0],number[1],…,number[n-1]) + 1;
  choosing[myid] = FALSE;
  for (j=0 to n-1) {
    while (choosing[j])
      ;
    while (number[j] != 0)
      && ((number[j],Pj)<(number[myid],myid))
      ;
  }
}
leave_CS(myid) {
  number[myid] = 0;
}
```

Remember, processes might do this multiple #s of times, including 0.

Evaluation of the Bakery Algorithm

- Does it work?
- What do you need to prove?
- Show that it meets
  - Mutual exclusion
  - Progress
  - Bounded waiting requirements
- Need to know that maximum # processes

Looking to the Hardware

- Complications arose because we had atomicity only at the granularity of a machine instruction
  - What a machine instruction could do is (was) limited
- Can we provide specialized instructions in hardware to provide additional functionality (with an instruction still being atomic)?
  - Looking again to hardware to help solve a OS problem
- Many systems (now) provide hardware support for implementing the critical section code
- All solutions below are based on idea of locking
  - Protecting critical regions via locks
  - But without special instructions

Synchronization Hardware

- Uniprocessors – could disable interrupts
  - Currently running code executes without preemption
  - Generally too inefficient on multiprocessor systems
    - operating systems using this not broadly scalable
- Modern CPUs provide atomic hardware instructions (2 general types)
  - *Test-and-Set*
    - test memory word and set value
  - *Compare-and-Swap*
    - swap contents of 2 memory words
- How does this help?

Critical-section Solution Using Locks

```
while (TRUE){
 acquire lock
    critical section
 release lock
    remainder section
}
```

The infinite do loop is just to suggest that a process will possibly want to enter its critical section multiple #s of times, including 0 times and 1 time.

- The problem before was that "acquire" and "release" could not be done in a single instruction
- Suppose it could with a single atomic instruction

test_and_set Instruction

- Definition:

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

boolean types have values of 0 or 1

- Assume it executes atomically
- Returns the original value of passed parameter
- Set the new value of passed parameter to TRUE

Solution using test_and_set()

- Shared boolean variable *lock*, initialized to FALSE
- Solution:

```
while (TRUE){
    . . .
    while (test_and_set(&lock) == TRUE)
        ;

    /* critical section */

    lock = FALSE;
    . . .
}
```

- Does it work?

The infinite do loop is just to suggest that a process will possibly want to enter its critical section multiple #s of times, including 0 times and 1 time.

compare_and_swap Instruction

- Definition:

```
int compare_and_swap(int *value, int expected,
                     int newvalue) {
  int temp = *value;
  if (*value == expected)
    *value = newvalue;
  return temp;
}
```

- Assume it executes atomically
- Returns the original value of passed parameter *value*
- Set the variable *value* to the value of the passed parameter *newvalue*
  - Only if *value* == *expected*
- That is, the swap takes place only under this condition

Solution using compare_and_swap

- Shared integer *lock* initialized to 0
- Solution:

```
while (TRUE){
  . . .
  while(compare_and_swap(&lock, 0, 1) != 0)
    ;

    /* critical section */

  lock = 0;
   . . .
} while (true);
```

- Does it work?

Bounded-waiting with test_and_set

```
while (TRUE){
   . . .
    waiting[i] = TRUE;
    key = TRUE;
    while (waiting[i] && key)
       key = test_and_set(&lock);
    waiting[i] = FALSE;
     /* critical section */
    j = (i + 1) % n;
    while ((j != i) && !waiting[j])
       j = (j + 1) % n;
    if (j == i)
       lock = FALSE;
    else
       waiting[j] = FALSE;
   . . .
}
```

N processes:
P_0 , P_1 , …, P_{N-1}

This is the code for Pi (i.e., each process is executing this code with *i* set to the process ID ($0 \leq i \leq$ N-1)

Spinning vs. Blocking

- In the previous solutions, we are spinning (*busy-waiting*) for some condition to change
  - This change should be effected by some other process
  - We are "presuming" that this other process will eventually get the CPU
    - is this a reasonable assumption?
    - needs a preemptive scheduler … Why?
- This can be inefficient because:
  - You are wasting your time quantum spinning
  - Sometimes, your programs may not work!
    - suppose if the OS scheduler is not preemptive

Blocking Approaches

- If instead of busy-waiting, the process relinquishes the CPU at the time when it cannot proceed, the process is said to *block*
  - It is still wanting to get into the critical section
  - It is put in the blocked queue.
- It is the job of the process changing the *condition* to wake up a blocked process
  - Moves it from blocked back to ready queue
- Advantage:
  - Do not unnecessarily occupy CPU cycles
- Disadvantages?

Blocking Example

```
Enter_CS(L) {
    Disable Interrupts
    Check if anyone is using L
    If not {
         Set L to being used
    }
    else {
         Move this PCB to Blocked
              queue for L
         Select another process to run
              from Ready queue
         Context switch to that process
    }
    Enable Interrupts
}
```

```
Exit_CS(L) {
    Disable Interrupts
    Check if blocked queue
       for L is empty
    if so {
       Set L to free
    }
    else {
        Move PCB from head of
             Blocked queue of L to
             Ready queue
    }
    Enable Interrupts
}
```

Think of L as a lock

NOTE: These must be OS system calls! Why?

Mutex Locks

- Previous solutions are complicated and generally inaccessible to application programmers
- OS designers build software tools to solve critical section problem
- Simplest is a *mutex* lock ("mutual exclusion")
- Protect a critical section by first *acquire()* a lock then *release()* the lock
  - Boolean variable indicating if lock is available or not
- Calls to *acquire()* and *release()* must be atomic
  - Usually implemented via hardware atomic instructions
- This solution generally requires busy waiting
  - This lock therefore is called a *spinlock*

acquire() and release()

```
acquire() {
  while (!available)
    ; /* busy wait */
  available = false;
}
release() {
  available = true;
}
while (TRUE){
  acquire lock
  critical section
  release lock
  remainder section
}
```

The assumption is that *acquire()* and *release()* will be executed atomically

Synchronization Constructs

- Synchronization requires more than just exclusion
  - If printer queue is full, I need to wait until there is at least 1 empty slot
- Note that *acquire()* / *release()* are not very suitable to implement such synchronization … Why?
- We need constructs to enforce orderings
  - A should be done after B
- We need construction to help keep track of numbers of things

Semaphore (Dijkstra)

- Synchronization tool that provides more sophisticated ways (than mutex locks) for processes to synchronize their activities.
- A semaphore *S* is an integer variable
- Can only be accessed via two indivisible (atomic) operations
  - *wait()* and *signal()*
- Definition of the *wait()* operation

```
wait(S) { // P()
  while (S <= 0)
    ; // busy wait
  S--;
}
```

Originally called *P()* and *V()* by Dijkstra
P = Probeer ('try' in Dutch)
V = Verhoog ('increment' in Dutch)

- Definition of the *signal()* operation

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

https://en.wikipedia.org/wiki/Semaphore_(programming)

Semaphore Usage

- *Binary semaphore*
  - Integer value can range only between 0 and 1
  - Same as a mutex lock
- *Counting semaphore*
  - Integer value can range over an unrestricted domain
- Can solve various synchronization problems with semaphores
- Consider P_1 and P_2 that require S_1 to happen before S_2
  - Create a semaphore *synch* initialized to 0

```
P1:
   S1;
   signal(synch);
P2:
   wait(synch);
   S2;
```

- Can implement a counting semaphore *S* be a binary semaphore?

Semaphore Implementation

- Must guarantee that no two processes can execute the *wait()* and *signal()* …
  - … on the same semaphore …
  - … at the same time
- Thus, the implementation becomes the critical section problem where the *wait* and *signal* code are placed in the critical section
  - Could have busy waiting in critical section implementation
    - but implementation code is short
    - little busy waiting if critical section rarely occupied
- Note that applications may spend lots of time in critical sections and therefore this is not a particularly good solution

Semaphores with no Busy waiting

- With each semaphore there is an associated waiting queue
- Each entry in a waiting queue has two data items:
  - value (of type integer)
  - pointer to next record in the list
- Two operations:
  - *block* – place the process invoking the operation on the appropriate waiting queue
  - *wakeup* – remove one of processes in the waiting queue and place it in the ready queue

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

Implementation with no Busy waiting

```
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 process P from S->list;
    wakeup(P);
  }
}
```

NOTE: These are involving OS system calls, and there is no atomicity lost during the execution of these routines because interrupts are being disabled.

Problems with Semaphores

- Incorrect use of semaphore operations:
  - *signal (mutex) .... wait (mutex)*
  - *wait (mutex) ... wait (mutex)*
  - Omitting of *wait (mutex)* or *signal (mutex)* (or both)
- Deadlock and starvation are possible

Deadlock and Starvation

- Definition: A *deadlock* situation occurs if 2 or more processes are waiting indefinitely for an event that only one of the waiting processes can cause
- 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);`

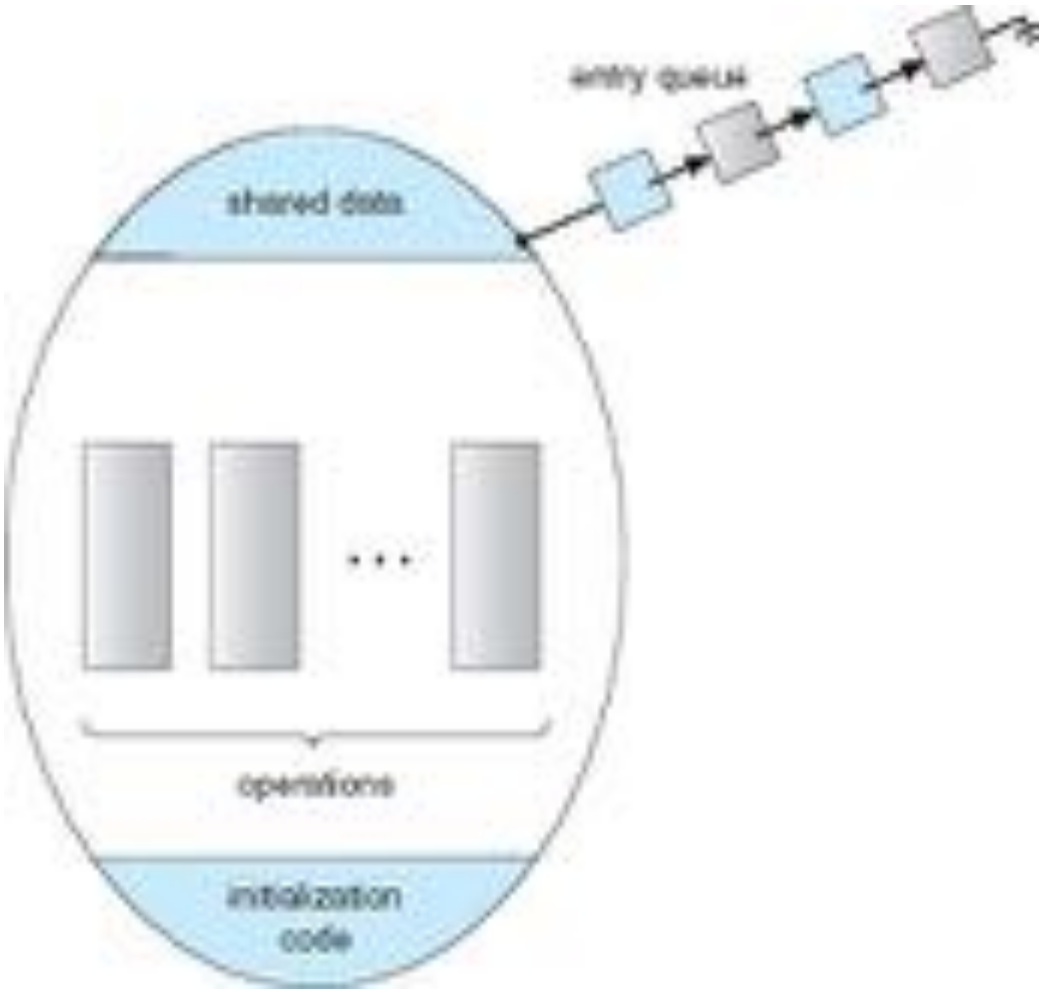
- *Starvation* (indefinite blocking)
  - A process may never be removed from the semaphore queue in which it is suspended
- *Priority inversion*
  - Scheduling problem when lower-priority process holds a lock needed by higher-priority process
  - Solved via priority-inheritance protocol

Monitors

- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- Abstract data type
  - Internal variables only accessible by code within the procedure
  - Routines (operations) that operate on the internal variables (shared)
  - External world only sees these operations (not the shared data or how the operations and synchronization are implemented)
- Only one process may be active within the monitor at a time
  - All the processes that are executing monitor code, there can be at most 1 process in ready queue (rest are either blocked or not in monitor!)

```
monitor monitor-name {
  // shared variable declarations
  procedure P1 (…) { …. }
  procedure Pn (…) {……}
  Initialization code (…) { … }
}
```

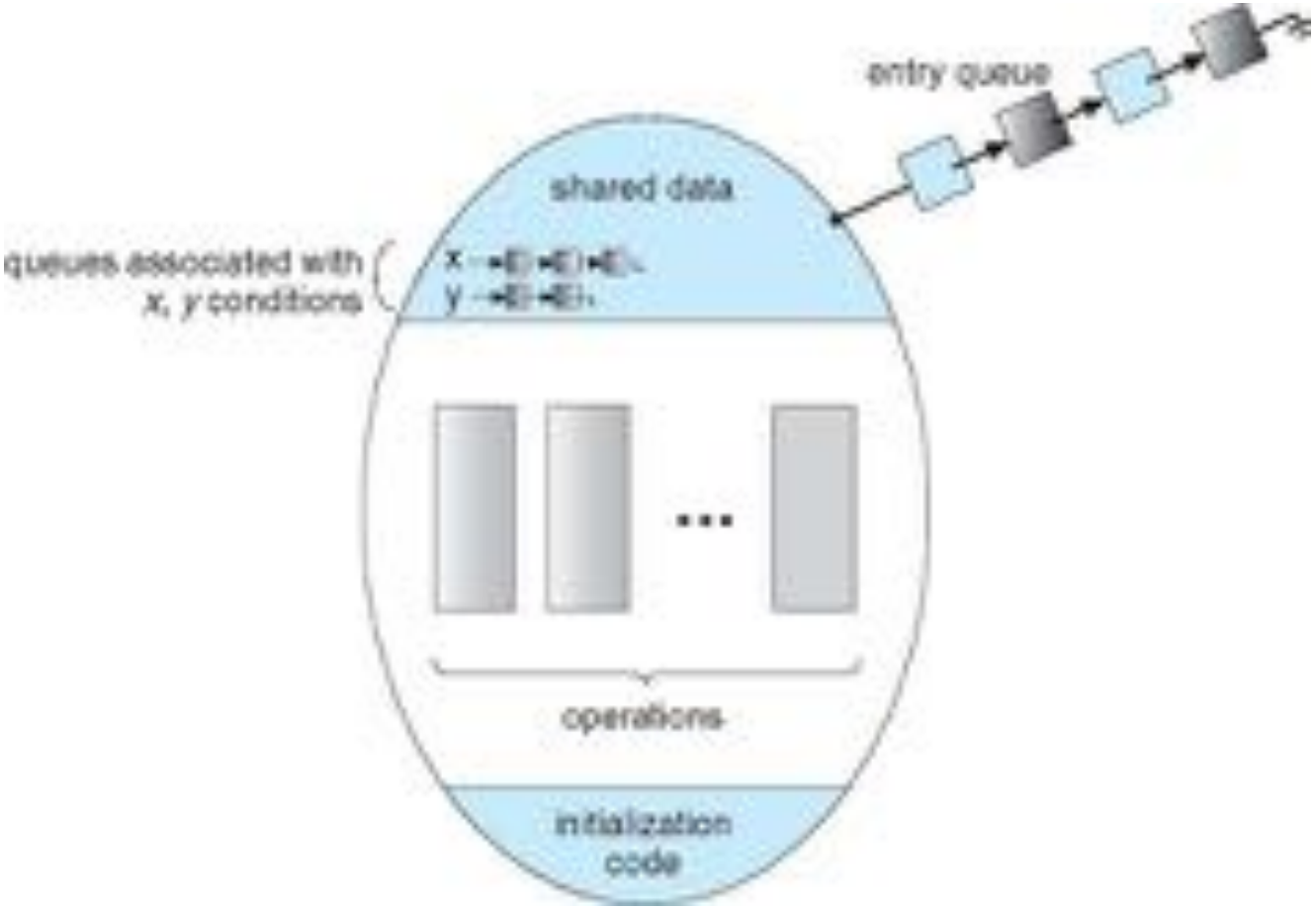
Schematic view of a Monitor



Condition Variables

- *condition x, y;*
- Two operations are allowed on a condition variable:
  - *x.wait()* – a process that invokes the operation is suspended until *x.signal()*
  - *x.signal()* – resumes one of the processes (if any) that invoked *x.wait()*
    - if no *x.wait()* on the variable, then it has no effect on the variable
- NOTE: If the signal comes before the wait, the signal gets lost!!! – You need to be careful since signals are not stored unlike semaphores

Monitor with Condition Variables



Condition Variables Choices

- If process *P* invokes *x.signal()*, and process *Q* is suspended in *x.wait()*, what should happen next?
  - Both *Q* and *P* cannot execute in parallel
  - If *Q* is resumed, then *P* must wait
- Options include
  - *Signal and wait* – *P* waits until *Q* either leaves the monitor or it waits for another condition
  - *Signal and continue* – *Q* waits until *P* either leaves the monitor or it waits for another condition
  - Both have pros and cons – language implementer can decide
  - Monitors implemented in Jave, C#, Concurrent Pascal, …

Mesa versus Hoare Semantics

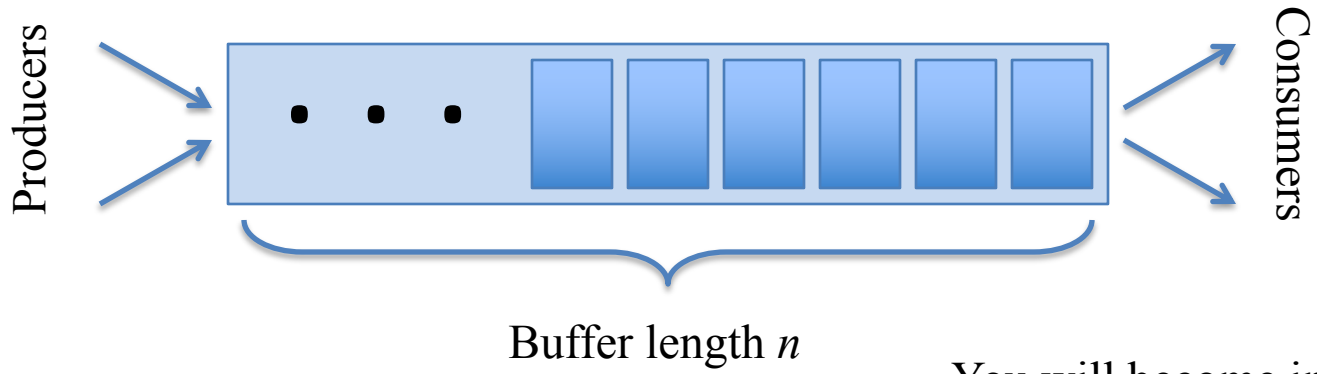
- Can implement signal in two ways
- *Mesa* (signal and continue)
  - Signal puts waiter on the ready list
  - Signaller keeps lock and the processor
- *Hoare* (signal and wait)
  - Signal gives processor and lock to waiter
  - Waiter gives processor and lock back to the signaller when it finishes
  - It is possible to support nested signalling

Classical Problems of Synchronization

- Classical problems used to test newly-proposed synchronization schemes
  - Bounded-Buffer Problem
  - Readers and Writers Problem
  - Dining-Philosophers Problem

Bounded-Buffer Problem

- *n* buffers, each can hold one item
- Semaphore *mutex* initialized to the value 1
- Semaphore *full* initialized to the value 0
- Semaphore *empty* initialized to the value *n*



You will become intimately familiar with bounded buffers in Project 3.

Bounded Buffer Problem – Producer

- The structure of the producer process

```
while (TRUE){
   ...
   /* produce an item in next_produced */
   ...
  wait(empty);  /* what is known afterwards? */
  wait(mutex);
   ...
   /* add next produced to the buffer */
   ...
  signal(mutex); /* release CS access */
  signal(full); /* informs the consumer */
}
```

critical section

Bounded Buffer Problem – Consumer

- The structure of the consumer process

```
while (TRUE){
 wait(full); /* what is known afterwards? */
 wait(mutex);
  ...
  /* remove an item from buffer next_consumed */
  ...
 signal(mutex); /* release CS access */
 signal(empty); /* informs the consumer */
  ...
  /* consume the item in next_consumed */
  ...
}
```

critical section

Does this work for multiple producers and consumers?

Readers-Writers Problem

- A data set is shared among a number of concurrent processes
  - *Readers* – only read the data set (does not perform any updates)
  - *Writers* – can both read and write
- Problem – allow multiple readers to read at the same time
  - Only one writer can access the shared data at the same time
- Several variations of how readers and writers are considered
  - All involve some form of priorities
- Shared data
  - Data set
  - Semaphore *rw_mutex* initialized to 1
  - Semaphore *mutex* initialized to 1
  - Integer *read_count* initialized to 0

Readers-Writers Problem – Writer

- The structure of a writer process

```
while (TRUE){
   wait(rw_mutex);
   …
   /* writing is performed */
   …
   signal(rw_mutex);
}
```

Critical section

Readers-Writers Problem – Reader

- The structure of a reader process

```
while (TRUE){
  wait(mutex);
  read_count++;
  if (read_count == 1) /* what is happening here? */
    wait(rw_mutex);
  signal(mutex);
  …
  /* reading is performed */
  …
  wait(mutex);
  read count--;
  if (read_count == 0)
    signal(rw_mutex)
  signal(mutex)
}
```

Critical section
- Can multiple readers be in?
- Can writer be in?

Readers-Writers Problem Variations

- Do you see any problems?
- First variation (above)
  - No reader kept waiting unless writer has permission to use shared object
- Second variation
  - Once writer is ready, it performs the write ASAP
  - Hmm, how would you implement this?
- Both may have starvation leading to even more variations
- Problem is solved on some systems by kernel providing reader-writer locks

Dining Philosophers Problem (Dijkstra)

- Philosophers spend their lives alternating thinking and eating
- Philosophers do not interact with their neighbors, but occasionally try to pick up 2 forks (one at a time, closest to them) to eat
  - Need both to eat, then release both when done
- In the case of 5 philosophers
  - Shared data
    - bowl of spaghetti (data set)
    - semaphore *fork [5]* initialized to 1

https://en.wikipedia.org/wiki/Dining_philosophers_problem

Dining Philosophers Problem Algorithm

- The structure of Philosopher *i*:

```
While (TRUE){
    wait (fork[i] );
    wait (fork[ (i + 1) % 5] );
    //  eat
    signal (fork[i] );
    signal (fork[ (i + 1) % 5] );
    //  think
}
```

critical section

- What is the problem with this algorithm?

Preventing Starving Philosophers

- Deadlock handling
  - Allow at most 4 philosophers to be sitting simultaneously at the table
    - cheating
  - Allow a philosopher to pick up the forks only if both are available
    - picking must be done in a critical section
  - Use an asymmetric solution
    - an odd-numbered philosopher picks up first the left fork and then the right fork
    - even-numbered philosopher picks up first the right fork and then the left fork

Monitor Solution to Dining Philosophers

```
monitor DiningPhilosophers
{
   enum { THINKING; HUNGRY, EATING) state [5] ;
   condition self [5];

   void pickup (int i) {
     state[i] = HUNGRY;
     test(i);
     if (state[i] != EATING) self[i].wait;
   }

    void putdown (int i) {
     state[i] = THINKING;
     // test left and right neighbors
     test((i + 4) % 5);
     test((i + 1) % 5);
   }
```

Monitor Solution to Dining Philosophers

```
    void test (int i) {
      if ((state[(i + 4) % 5] != EATING)
           && (state[i] == HUNGRY)
           && state[(i + 1) % 5] != EATING) ) {
          state[i] = EATING ;
          self[i].signal () ;
      }
     }

     initialization_code() {
      for (int i = 0; i < 5; i++)
        state[i] = THINKING;
    }
}
```

What does each Philosopher do?

- Each philosopher *i* invokes the operations *pickup()* and *putdown()* in the following sequence:

  DiningPhilosophers.pickup(i);

  EAT

  DiningPhilosophers.putdown(i);

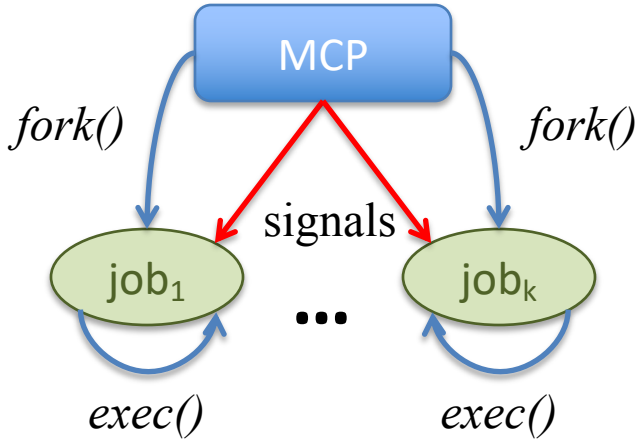
- No deadlock, but starvation is possible

Resuming Processes within a Monitor

- If several processes queued on condition *x*, and *x.signal()* executed, which should be resumed?
- FCFS frequently not adequate
- conditional-wait construct of the form *x.wait(c)*
  - Where *c* is priority number
  - Process with lowest number (highest priority) is scheduled next

Project 2: MCP – Ghost in the Shell

- Implement a program that controls multiple processes in execution
  - Master Control Program (MCP)
- MCP launches jobs
  - Use *fork()* to create processes
  - Use *exec()* to run jobs
- MCP uses signals to control process execution
- Objective is to give you experience working with processes and signals



Next Class

- Deadlocks