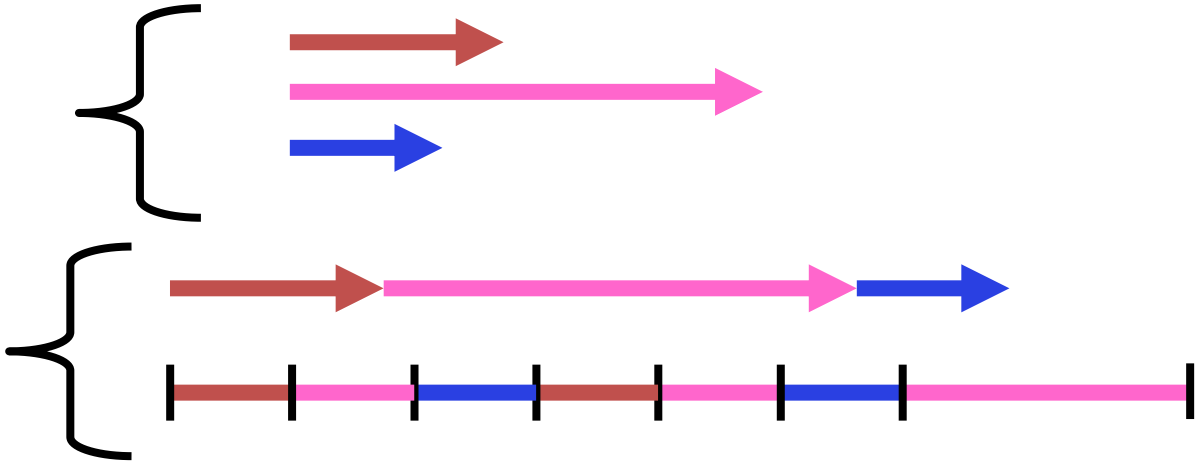
Process Synchronization

Chapter 6 & 7

Background



- **Cooperating process (threads)** can affect or be affected by other processes (threads).
- Concurrent access to shared data may result in data inconsistency
- What does it mean to run two threads "concurrently"?
  - Scheduler is free to run threads in any order and interleaving: FIFO, Random, …
  - Dispatcher can choose to run each thread to completion of time-slice in big chunks or small chunks
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes

Concurrency Problem at Program Execution level

- Concurrency problems can arise at the program execution level.
- Example: Suppose two processes A and B run concurrently and each statement in the code assumed to be atomic.

Process A

```
cout << "a";
cout << "c";
```

Process B

```
cout << "b";
cout << "d";
```

- Assume atomic program instruction execution.
- **Question:** How many different outputs are possible if Process A and B execute concurrently?

Recall the Bounded Buffer problem

- Producer/Consumer problem: Producer writes to a buffer and the Consumer reads from the buffer.
- Shared-memory solution to bounded-buffer problem (Chapter 3) allows at most *n* – 1 items in buffer at the same time.
- Suppose that we wanted to provide a solution to the consumer-producer problem that fills all the buffers. We can do so by having an integer count that keeps track of the number of full buffers.
  - Initially, count is set to 0.
  - It is incremented by the producer after it produces a new item and is decremented by the consumer after it consumes an item.

Bounded-Buffer

- Shared data

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

Producer

```
while (true) {

     /*  produce an item and put in nextProduced  */
     while (count == BUFFER_SIZE) {}          // busy waiting, do nothing
    /* add one item */
     buffer [in] = nextProduced;
     in = (in + 1) % BUFFER_SIZE;
     count++;
}
```

Consumer

```
while (true)  {
        while (count == 0)  {}  // busy waiting,do nothing
          /*  consume the item in nextConsumed */
           nextConsumed =  buffer[out];
           out = (out + 1) % BUFFER_SIZE;
          count--;
  }
```

Accessing count concurrently

- What happens when the statements
  count++;
  count--;
  are performed concurrently?

- count++ could be implemented as:

```
register1 = count
register1 = register1 + 1
count = register1
```

- count-- could be implemented as :

```
register2 = count
registe2  = register2 - 1
count = register2
```

Race Condition

- Suppose counter initially set to 5. Execution of counter++ and counter-- consecutively should leave the value at 5.
- Concurrent execution could leave inconsistent data:
  - S0: producer execute register1 = count {register1 = 5}
  - S1: producer execute register1 = register1 + 1 {register1 = 6}
  - S2: consumer execute register2 = count {register2 = 5}
  - S3: consumer execute register2 = register2 - 1 {register2 = 4}
  - S4: producer execute count = register1 {count = 6 }
  - S5: consumer execute count = register2 {count = 4}

- Could end up with counter value of 4, 5 or 6
- There is no way to predict the relative speed of process execution, so you cannot guarantee that one will finish before the other.

Race Condition

- **Race condition**: The situation where several processes access – and manipulate shared data concurrently. The final value of the shared data depends upon the order of process accessing the data.
- To prevent race conditions, concurrent processes must be **synchronized**.
- e.g. The statements
  count++;
  count--;
  must be performed *atomically*.
- **Atomic operation** means an operation that completes in its entirety without interruption.

Atomic Operations

- To understand a concurrent program, we need to know what the underlying indivisible operations are!
- Atomic Operation: an operation that always runs to completion or not at all
  - It is *indivisible:* it cannot be stopped in the middle and state cannot be modified by someone else in the middle
  - Fundamental building block – if no atomic operations, then have no way for threads to work correctly together
- On most machines, memory references and assignments (i.e. loads and stores) of words are atomic
- Many instructions are not atomic
  - Double-precision floating point store often not atomic
  - VAX and IBM 360 had an instruction to copy a whole array

Correctness Requirements

- Threaded programs must work for all interleavings of thread instruction sequences
  - Bugs of cooperating threads inherently non-deterministic and non-reproducible
  - Really hard to debug unless carefully designed!
- Example: Therac-25
  - Machine for radiation therapy
    - Software control of electron accelerator and electron beam/ Xray production
    - Software control of dosage
  - Software errors caused the death of several patients
    - A series of race conditions on shared variables and poor software design
    - "They determined that data entry speed during editing was the key factor in producing the error condition: If the prescription data was edited at a fast pace, the overdose occurred."

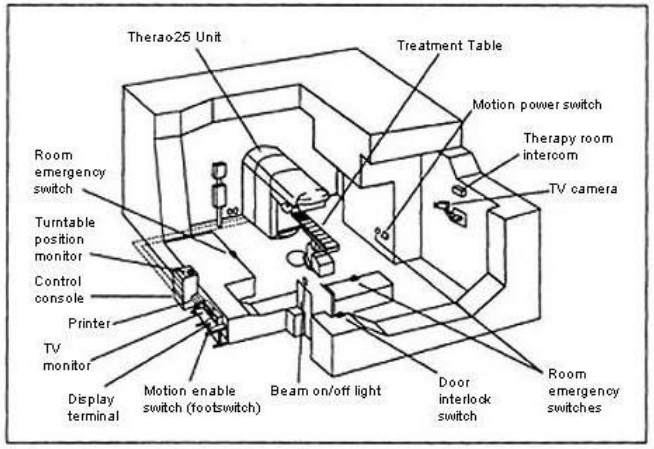


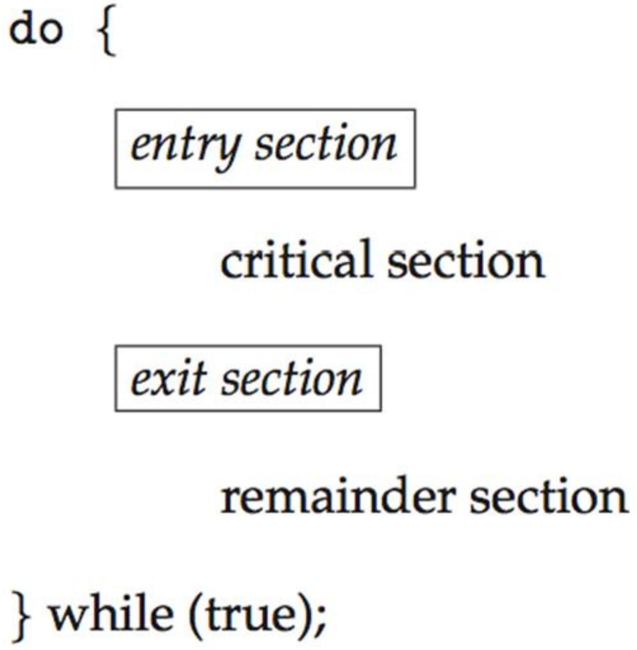
Figure 1. Typical Therac-25 facility

Critical Section Problem

- Consider system of ***n*** processes $\{p_0, p_1, \ldots p_{n-1}\}$
- Each process has **critical section** segment of code
  - Process may be changing common variables, updating table, writing file, etc
  - When one process in critical section, no other may be in its critical section
- ***Critical section problem*** is to design protocol to solve this
- Each process must ask permission to enter critical section in **entry section**, may follow critical section with **exit section**, then **remainder section**

Critical Section

- General structure of process ***P**$_i$*



```
do {

        entry section

            critical section

        exit section

            remainder section

} while (true);
```

Solution to Critical-Section Problem

1. Mutual Exclusion - If process P_i is executing in its critical section, then no other processes can be executing in their critical sections
2. 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
3. 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

Initial Attempts to Solve Problem

- 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
        remainder section
} while (1);
```

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

Algorithm 1

- Shared variables:
  - **int turn**;
    initially **turn = 0**
  - **turn = i** $\Rightarrow P_i$ can enter its critical section
- Process P_i

```
do {
    while (turn != i) ;
    critical section
    turn = j;
    remainder section
} while (1);
```

- Satisfies mutual exclusion, but not progress

Algorithm 2

- Shared variables
  - **boolean flag[2]**;
    initially **flag [0] = flag [1] = false.**
  - **flag [i] = true** $\Rightarrow$ P_i ready to enter its critical section
- Process P_i

```
do {
    flag[i] = true;
    while (flag[j]) ;
    critical section
    flag [i] = false;
    remainder section
} while (1);
```

- Satisfies mutual exclusion, but not progress requirement.

Algorithm 3

- Combined shared variables of algorithms 1 and 2.
- Process P_i

```
do {
    flag [i] = true;
    turn = j;
    while (flag [j] and turn == j) ; // do nothing
    critical section
    flag [i] = false;
    remainder section
} while (1);
```

- Meets all three requirements; solves the critical-section problem for two processes.
- Peterson’s Algorithm. Not guaranteed to work on modern computer architectures.

Where are we going with synchronization?

Programs	Shared Programs
Higher-level API	Locks Semaphores Monitors Send/Receive
Hardware	Load/Store Disable Ints Test&Set Comp&Swap

- We are going to implement various higher-level synchronization primitives using atomic operations
  - Everything is pretty painful if only atomic primitives are load and store
  - Need to provide primitives useful at user-level

Synchronization Hardware

- Many systems provide hardware support for critical section code
- Uniprocessor – could disable interrupts in kernel
  - Currently running code would execute without preemption
  - Generally too inefficient on multiprocessor systems
    - Operating systems using this not broadly scalable
- Modern machines provide special **atomic** hardware instructions
  - Either test memory word and set value (`TestAndSet`)
  - Or swap contents of two memory words: `compare_and_swap`
- We can use these special instructions to implement a “lock” and solve the critical section problem.

Mutex Lock

- Suppose we have some sort of implementation of a lock (more in a moment).
  - `Lock.Acquire()` – wait until lock is free, then grab
  - `Lock.Release()` – Unlock, waking up anyone waiting
  - These must be atomic operations – if two threads are waiting for the lock and both see it's free, only one succeeds to grab the lock
- Then, our critical section problem is easy:

```
lock.Acquire();
  Critical section
lock.Release();
```

- Once again, section of code between `Acquire()` and `Release()` called a "Critical Section"

TestAndSet Instruction

- Definition:

```
boolean TestAndSet (boolean *var)
{
     boolean temp = *var;
     *var = TRUE;
     return temp;
}
```

atomic

1. Executed atomically
2. Returns the original value of passed parameter
3. Set the new value of passed parameter to “TRUE”.

Solution using TestAndSet

- Shared Boolean variable lock., initialized to FALSE.
- Solution:

```
while (true) {
    // aquire lock
    while ( TestAndSet (&lock ));   /* do nothing */

         critical section

    // release lock
    lock = FALSE;
         remainder section
  }
```

compare_and_swap Instruction

- Definition:

```
int compare_and_swap(int *var, int expected, int new_value) {
      int temp = *var;
      if (*var == expected)
           *var = new_value;
      return temp;
}
```

1. Executed atomically
2. Returns the original value of passed parameter "var"
3. Set the variable "var" the value of the passed parameter "new_value" but only if "var ==expected". That is, the swap takes place only under this condition.

Solution using compare_and_swap

- Shared Boolean variable lock initialized to 0;
- Solution:

```
do {
      while (compare_and_swap(&lock, 0, 1) != 0); /*do nothing
   */
      /* critical section */
      lock = 0; /* unlock */
      /* remainder section */
} while (true);
```

Atomic Variables

- Typically, instructions such as `compare_and_swap` are used as building blocks for other synchronization tools.
- One tool is an **atomic variable** that provides *atomic* (uninterruptible) updates on basic data types such as integers and booleans.
- For example, the **`increment()`** operation on the atomic variable **`sequence`** ensures **`sequence`** is incremented without interruption:

```
increment(&sequence);
```

Atomic Variables

- The **`increment()`** function can be implemented as follows:

```
void increment(atomic_int *v)
{
int temp;

do {
   temp = *v;
}
while (temp != (compare_and_swap(v,temp,temp+1));
}
```

Busy Waiting with TestAndSet

- Now we have a simple solution for a lock:

```
int lock = 0; // Free
acquire() {
   while (TestAndSet(lock)); // while busy
}
release() {
   lock = 0;
}
```

- But this implementation requires **busy waiting**
  - This lock therefore called a **spinlock**
  - This can be very inefficient because the busy-waiting thread will consume cycles waiting
  - Waiting thread may take cycles away from thread holding lock (no one wins!)
  - Priority Inversion: If busy-waiting thread has higher priority than thread holding lock $\Rightarrow$ no progress!

Notes on TestAndSet

- The previous algorithm satisfies mutual exclusion.
- The algorithm works for multiple processes. (But there is no ordering of processes waiting to enter the CS).
- It does not satisfy bounded waiting. Starvation is possible.

Solution with no starvation

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

```
boolean lock;
boolean waiting[n]; //list of processes waiting for CS
```

- Process P_i

```
while (true) {
    waiting[i] = true;
    key = true;
    while (waiting[i] && key) key = TestAndSet(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;
    remainder section
}
```

Satisfying Requirements

- The previous algorithm satisfies the three requirements
- Mutual Exclusion: The first process to execute TestAndSet(lock) when lock is false, will set lock to true so no other process can enter the CS.
- Progress: When a process exits the CS, it either sets lock to false, or waiting[j] to false, allowing the next process to proceed.
- Bounded Waiting: When a process exits the CS, it examines all the other processes in the waiting array in a circular order. Any process waiting for CS will have to wait at most n-1 turns.

Better Locks using TestAndSet

Can we build TestAndSset locks without busy-waiting?

Can't entirely, but can minimize!

Idea: only busy-wait to atomically check lock value

```
int guard = 0;
int lock = FREE;

Acquire() {
    // Short busy-wait time
    while (TestAndSet(guard));
    if (lock == BUSY) {
        put thread on wait queue;
        go to sleep() & guard = 0;
    } else {
        lock = BUSY;
        guard = 0;
    }
}
```

```
Release() {
    // Short busy-wait time
    while (TestAndSet(guard));
    if anyone on wait queue {
        take thread off wait queue
        Place on ready queue;
    } else {
        lock = FREE;
    }
    guard = 0;
```

- Note: sleep has to be sure to reset the guard variable
  - Why can't we do it just before or just after the sleep?

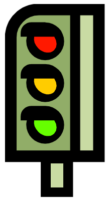
Mutex Locks and Busy Waiting

- OS designers build software tools to solve critical section problem
- Simplest is mutex lock
- Product critical regions with it by first **`acquire()`** a lock then **`release()`** it
  - Boolean variable indicating if lock is available or not
- Calls to **`acquire()`** and **`release()`** must be atomic
  - Usually implemented via hardware atomic instructions
- Mutex lock implementation
  - Busy waiting: spinlock good for multiprocessor and threads holding the lock for short durations.
  - Blocking: waiting threads go to sleep in a wait queue.

Higher-level Primitives than Locks

- Synchronization is a way of coordinating multiple concurrent activities that are using shared state
  - Want right abstractions for synchronizing threads that share memory?
  - Want as high a level primitive as possible
- Good primitives and practices important!
  - Since execution is not entirely sequential, really hard to find bugs, since they happen rarely
  - UNIX is pretty stable now, but up until about mid-80s (10 years after started), systems running UNIX would crash every week or so – concurrency bugs

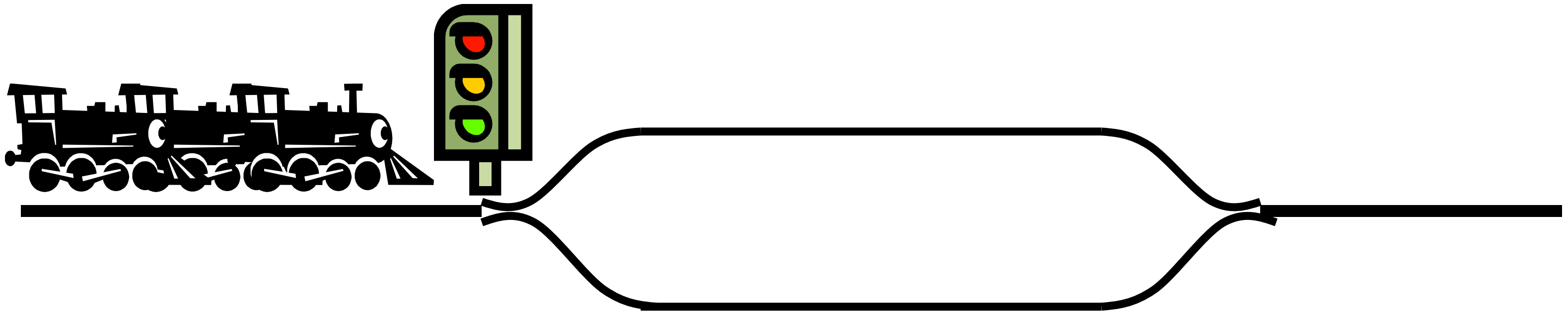
Semaphores



- Semaphores are a kind of generalized lock
  - First defined by Dijkstra in late 60s
  - Main synchronization primitive used in original UNIX
- Definition: a Semaphore has a non-negative integer value and supports the following two operations:
  - P(): an **atomic** operation that waits for semaphore to become positive, then decrements it by 1
    - Think of this as the wait() operation
  - V(): an **atomic** operation that increments the semaphore by 1, waking up a waiting P, if any
    - This of this as the signal() operation
  - Note that P() stands for "*proberen*" (to test) and V stands for "*verhogen*" (to increment) in Dutch

Semaphores Like Integers Except

- Semaphores are like integers, except
  - No negative values
  - Only operations allowed are wait() and signal() – can't read or write value, except to set it initially
  - Operations must be atomic
    - Two waits together can't decrement value below zero
    - Similarly, thread going to sleep in wait won't miss wakeup from signal – even if they both happen at same time
- Semaphore from railway analogy
  - Here is a semaphore initialized to 2 for resource control:



Semaphore as General Synchronization Tool

- **Binary** semaphore – integer value can range only between 0 and 1; can be simpler to implement
  - Equivalent to mutex locks
- "Binary Semaphore" can be used for mutual exclusion (initial value = 1)
  - mutual exclusion:

```
semaphore mutex;    //  initialized to 1
    wait(mutex);
       // Critical section goes here
    signal(mutex);
```

- **Counting** semaphore – integer value can range over an unrestricted domain
- Counting semaphore can be used to control access to a given resource consisting of a finite number of instances

Uses of Semaphores

- Scheduling Constraints (initial value = 0)
  - Locks are fine for mutual exclusion, but what if you want a thread to wait for something?
  - Example: suppose you had to implement ThreadJoin which must wait for thread to terminate:

```
semaphore s = 0// Initial value = 0
ThreadJoin {
  wait(s);
}
ThreadFinish {
  signal(s);
}
```

Semaphore Implementation

- Must guarantee that no two processes can execute wait () and signal () on the same semaphore at the same time
- Don’t want busy waiting on a semaphore
  - A busy-waiting semaphore is also called spin-lock
  - Waste CPU cycles. Note that applications may spend lots of time in critical sections and therefore this is not a good solution.
  - If a semaphore is non-positive, a process should block itself and enter a wait-queue.
- Thus, implementation becomes the critical section problem where the wait and signal code are placed in the critical section.
  - Could now have busy waiting in critical section implementation
    - But implementation code is short
    - Little busy waiting if critical section rarely occupied

Advantages of Semaphores

- Powerful--Can solve a variety of synchronization problems
- Simple solution for multiple processes.
- Solution exists for multiple processors (not as simple).
- No busy waiting.
- No starvation.

What is the Problem here?

- 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);
```

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

Priority Inversion

- Locking can have complex interactions with priorities
- Example:
  - low priority thread A locks mutex M
  - medium priority thread B runs for long
  - high priority thread C waits on M
- Problem: B will complete before C will ever get to run (because B has priority over A, and C waits on a resource that A has)
  - even worse: C may have a time deadline to complete, deadline may expire and C may determine system problem, deadlock, etc… and restart…

Solution

- priority inheritance: once C blocks on M, A (which has M) gets C’s priority (or higher)

What really happened on Mars?

- Pathfinder contained an "information bus", which you can think of as a shared memory area used for passing information between different components of the spacecraft. A bus management task ran frequently with high priority to move certain kinds of data in and out of the information bus. Access to the bus was synchronized with mutual exclusion locks (mutexes).
- The meteorological data gathering task ran as an infrequent, low priority thread, and used the information bus to publish its data. When publishing its data, it would acquire a mutex, do writes to the bus, and release the mutex. If an interrupt caused the information bus thread to be scheduled while this mutex was held, and if the information bus thread then attempted to acquire this same mutex in order to retrieve published data, this would cause it to block on the mutex, waiting until the meteorological thread released the mutex before it could continue.
- The spacecraft also contained a communications task that ran with medium priority.
- Most of the time this combination worked fine. However, very infrequently it was possible for an interrupt to occur that caused the (medium priority) communications task to be scheduled during the short interval while the (high priority) information bus thread was blocked waiting for the (low priority) meteorological data thread. In this case, the long-running communications task, having higher priority than the meteorological task, would prevent it from running, consequently preventing the blocked information bus task from running. After some time had passed, a watchdog timer would go off, notice that the data bus task had not been executed for some time, conclude that something had gone drastically wrong, and initiate a total system reset.

Classical Problems of Synchronization

- Bounded-Buffer (Producer-Consumer) Problem
- Readers and Writers Problem
- Dining-Philosophers Problem

Critical Section

Producer

```
Lock.acquire()
   /*  produce an item and put in
      nextProduced  */
    while (count == BUFFER_SIZE) {} //
      busy waiting, do nothing
   /* add one item */
   buffer [in] = nextProduced;
    in = (in + 1) % BUFFER_SIZE;
    count++;
Lock.release()
```

Consumer

```
Lock.acquire()
while (count == 0)  {} // busy
      waiting,do nothing
      /*  consume the item in
      nextConsumed */
      nextConsumed =  buffer[out];
      out = (out + 1) % BUFFER_SIZE;
       count--;
Lock.release()
```

Will this code work?

Bounded-Buffer Problem

- Just the producer-consumer problem
  - Size *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.

Bounded Buffer Problem (Cont.)

- The structure of the producer process

```
while (true)  {
        //   produce an item
       wait (empty);
       wait (mutex);

            //  add the item to the  buffer

        signal (mutex);
        signal (full);
}
```

Bounded Buffer Problem (Cont.)

- The structure of the consumer process

```
while (true) {
        wait (full);
        wait (mutex);

                //  remove an item from  buffer

        signal (mutex);
        signal (empty);

                //  consume the removed item

}
```

Discussion about Solution

- Why asymmetry?
  - **Producer does**: **`wait(empty), signal(full)`**
  - **Consumer does**: **`wait(full), signal(empty)`**
- Is order of waits important?
  - Yes! Can cause deadlock
- Is order of signals important?
  - No, except that it might affect scheduling efficiency
- What if we have 2 producers or 2 consumers?
  - Do we need to change anything?

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

Monitors (Shared Objects)

- A high-level abstraction that encapsulate the mechanism for process synchronization
- *Abstract data type*, internal variables only accessible by code within the procedure
- Only one process may be active within the monitor at a time
- using mutex lock and condition variables for managing concurrent access to shared data
  - Some languages like Java provide this natively
  - Most others use actual locks and condition variables

- Only one process may be active within the monitor at a time

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

    Initialization code ( ….) { … }
              …
    }
}
```

Motivation for Condition Variables

- Semaphores are a huge step up; just think of trying to do the bounded buffer with only loads and stores
  - Problem is that semaphores are dual purpose:
    - They are used for both mutex and scheduling constraints
    - Example: the fact that flipping of waits in bounded buffer gives deadlock is not immediately obvious. How do you prove correctness to someone?
- Cleaner idea: Use *locks* for mutual exclusion and *condition variables* for scheduling constraints

Producer Consumer Problem

Producer

```
Lock.acquire()
/*  produce an item and put in
    nextProduced  */
 while (count == BUFFER_SIZE) {} //
    busy waiting, do nothing
      /* add one item */
  buffer [in] = nextProduced;
  in = (in + 1) % BUFFER_SIZE;
  count++;
Lock.release()
```

Consumer

```
Lock.acquire()
while (count == 0)  {} // busy
    waiting,do nothing
    /*  consume the item in
    nextConsumed */
    nextConsumed =  buffer[out];
    out = (out + 1) % BUFFER_SIZE;
     count--;
Lock.release()
```

Busy waiting in Critical Section No Good!

?

Producer Consumer Problem Condition Variables

Producer

```
lock.acquire()
while (count == BUFFER_SIZE){
     wait(&space, &lock)
}
/* add one item */
buffer [in] = nextProduced;
in = (in + 1) % BUFFER_SIZE;
count++;
signal(&item)
lock.release()
```

Condition variables space, item

Consumer

```
lock.acquire()
while (count == 0)  {
   wait(&item, &lock)
}
/*  consume the item  */
nextConsumed =  buffer[out];
out = (out + 1)
   %BUFFER_SIZE;
count--;
signal(&space);
lock.release()
```

Condition Variables

- Condition Variable: maintain a queue of threads waiting for something *inside* a critical section
  - Key idea: allow sleeping inside critical section by atomically releasing lock at time when thread goes to sleep
- Condition variable's Operations:
  - **`wait(&cond, &lock)`**: Atomically release lock and go to sleep. Re-acquire lock later, before returning.
  - **`signal(&cond)`**: Wake up one waiting thread (if any)
  - **`broadcast(&cond)`**: Wake up all waiting threads
- Rule: Must hold lock when doing condition variable ops!

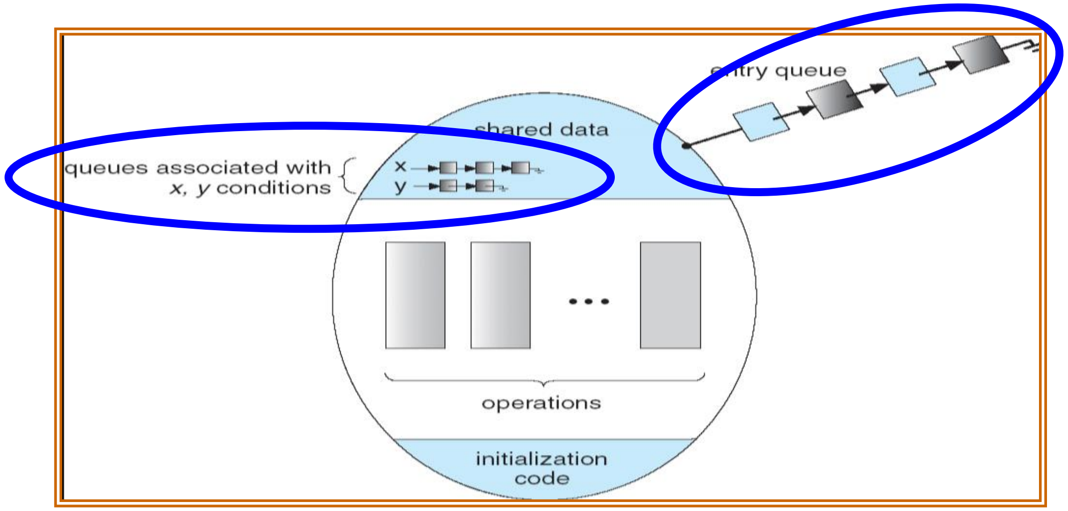
Condition Variables vs. Semaphore

- Condition variable has no internal values
- Condition variable wait() always suspends the calling thread
- Condition variable signal() wakes up one waiting thread (if any), and has no effect if no waiting thread

- Semaphore has non-negative integer value
- If semaphore value is positive, wait() decrements its value and continues. Wait() suspends the thread if semaphore value is 0.
- Semaphore signal() increments its value, and wakes up one waiting thread (if any).

Monitor with Condition Variables

- Lock: the lock provides mutual exclusion to shared data
  - Always acquire before accessing shared data structure
  - Always release after finishing with shared data
  - Lock initially free
- Condition Variable: a queue of threads waiting for something *inside* a critical section



Simple Monitor Example (version 1)

- Here is an (infinite) synchronized queue

```
class SynchronizedQueue {

  Lock lock;
  Queue queue;
  AddToQueue(item) {
    lock.Acquire();                     // Lock shared data
    queue.enqueue(item);                // Add item
    lock.Release();                     // Release Lock
  }

  RemoveFromQueue() {
    lock.Acquire();                     // Lock shared data
    item = queue.dequeue();             // Get next item or null
    lock.Release();                     // Release Lock
    return(item);                       // Might return null
  }}
```

How do we change the RemoveFromQueue() routine to wait until something is on the queue?

Complete Monitor Example (with condition variable)

- Here is an (infinite) synchronized queue

```
Lock lock;
Condition notempty;
Queue queue;
AddToQueue(item) {
    lock.Acquire();              // Get Lock
    queue.enqueue(item);         // Add item
    signal(&notempty);           // Signal a waiting thread
    lock.Release();              // Release Lock
}

RemoveFromQueue() {
    lock.Acquire();               // Get Lock
    while (queue.isEmpty()) {
       wait(&notempty, &lock); // If nothing in queue, sleep
    }
    item = queue.dequeue();       // Get next item
    lock.Release();               // Release Lock
    return(item);
}
```

Can we change while to if statement?

Using Monitor

It is made easy to use monitor in multi-threaded code

```
SynchronizedQueue squeue; // shared queue
```

Thread 1:

```
…
squeue.AddToQueue(x)
…
```

Thread 2:

```
…
y = squeue.RemoveFromQueue()
...
```

Suppose we're using message passing, will this code operate correctly?

```
/* NO SHARED MEMORY */

Producer                           Consumer
int item;                          int item;

while (TRUE) {                     while (TRUE) {
    item = Produce ();                 receive (Producer, &item);
    send (Consumer, &item);            Consume (item);
}                                  }
```

A. No, there is a race condition.

B. No, we need to protect *item* with a lock.

C. Yes, this code is correct.

This code is correct and relatively simple. Why don’t we always just use message passing (vs shared memory)

```
/* W/O SHARED MEMORY */

Producer                        Consumer
int item;                       int item;

while (TRUE) {                  while (TRUE) {
    item = Produce ();              receive (Producer, &item);
    send (Consumer, &item);         Consume (item);
}                               }
```

A. Message passing copies more data.

B. Message passing only works across a network.

C. Message passing is a security risk.

Readers-Writers Problem

- A data set is shared among a number of concurrent processes
  - Readers – only read the data set; they do not perform any updates
  - Writers – can both read and write.
- Problem – allow multiple readers to read at the same time. Only one single writer can access the shared data at the same time.
- Shared Data
  - Data set
  - Semaphore mutex initialized to 1.
  - Semaphore wrt initialized to 1.
  - Integer readcount initialized to 0.

Readers-Writers Problem (Cont.)

The structure of a writer process

```
while (true) {
    wait (wrt) ;

     //   writing is performed

      signal (wrt) ;
 }
```

The structure of a reader process

```
while (true) {
      wait (mutex) ;
      if (readcount == 0)  wait (wrt) ;
       readcount ++ ;
      signal (mutex)

       // reading is performed

       wait (mutex) ;
       readcount  - - ;
       if (readcount  == 0)  signal (wrt) ;
       signal (mutex) ;
 }
```

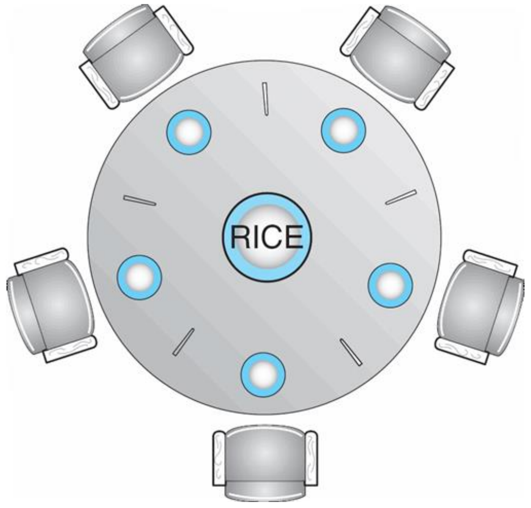
Questions: Which processes does this method favor (readers or writers)? What potential problem does this algorithm have?

Readers-Writers Problem Variations

- *First* variation – no reader kept waiting unless writer has permission to use shared object
- *Second* variation – once writer is ready, it performs write ASAP
- Both may have starvation leading to even more variations
- Problem is solved on some systems by kernel providing reader-writer locks

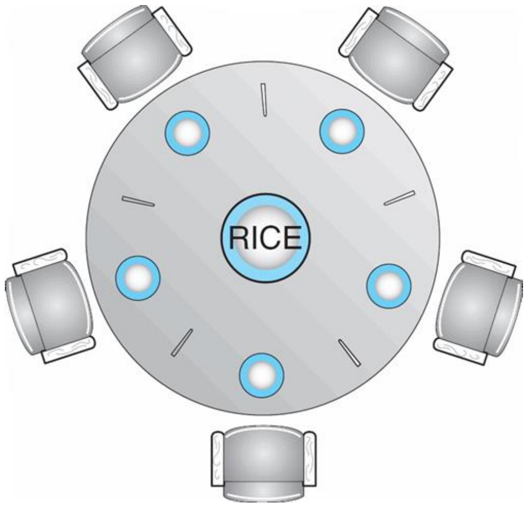
Dining-Philosophers Problem

- Philosophers are seated around a table.
- They alternate thinking and eating.
- One chopstick is placed in between each pair of philosophers.
- A philosopher needs the chopsticks on both sides to eat.
- A philosopher can only pick up one chopstick at a time.
- We want to guarantee that we don't generate deadlock.
- We want to guarantee that no philosopher starves.



Dining-Philosophers Problem

- Shared data
  - Chopsticks
  - Semaphore chopstick [5] initialized to 1



Dining-Philosophers Problem (Cont.)

- The structure of Philosopher *i*:

```
While (true)  {
        wait ( chopstick[i] );
        wait ( chopstick[ (i + 1) % 5] );

                //  eat

        signal ( chopstick[i] );
        signal (chopstick[ (i + 1) % 5] );

                //  think

}
```

Dining-Philosophers Problem

- **Deadlock:** If all philosophers decide to eat at the same time and pick up the chopstick on their left. None will be able to pick up a chopstick on their right.
- **Possible Solutions:**
  - 1)Allow at most 4 philosophers at the table
  - 2)Allow philosopher to pick up chopsticks only if both available.
  - 3)Odd philosophers pick up left first then right. Even philosophers pick up right first then left.
- **Starvation:** A philosopher may never have access to two chopsticks if the philosophers next to him are always eating.

Solution to Dining Philosophers

```
monitor DiningPhilosophters
    {
    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 l and r neighbors
        test((i - 1) % 5);
        test((i + 1) % 5);
    }
```

```
void test (int i) {
   if (
    (state[(i - 1)%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;
    }
}
```

Solution to Dining Philosophers (cont)

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

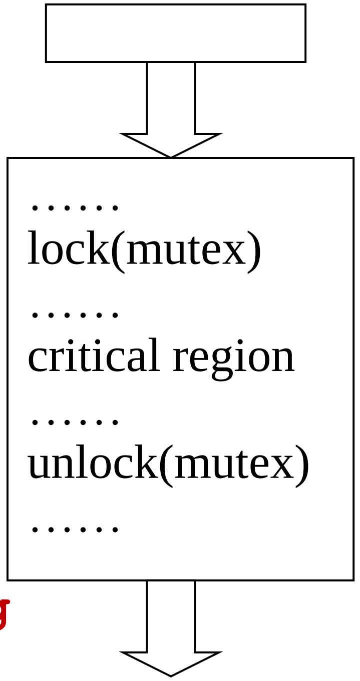
```
        DiningPhilosophters.pickup (i);

            EAT

        DiningPhilosophers.putdown (i);
```

Pthreads Synchronization

- Pthreads API is OS-independent
- It provides:
  - mutex locks
  - condition variables
  - semaphores
- Non-portable extensions include:
  - read-write locks
  - spin locks

Mutex (lock)

```
pthread_mutex_t mutex;
const pthread_mutexattr_t attr;
int status;

status = pthread_mutex_init(&mutex,&attr);

status = pthread_mutex_destroy(&mutex);

status = pthread_mutex_unlock(&mutex);

status = pthread_mutex_lock(&mutex); - blocking

status = pthread_mutex_trylock(&mutex); -
unblocking
```



Condition variables

```
int status;
pthread_condition_t cond;
const pthread_condattr_t attr;
pthread_mutex mutex;
status = pthread_cond_init(&cond,&attr);
status = pthread_cond_destroy(&cond);
status = pthread_cond_wait(&cond,&mutex);
status = pthread_cond_signal(&cond);
status = pthread_cond_broadcast(&cond);
```

Condition variables

```
pthread_mutex_lock(&mutex);
while (a != b)
     pthread_cond_wait(&cond, &mutex);
pthread_mutex_unlock(&mutex);
```

- wait on a condition variable.
- First, a thread needs to have the lock of the mutex.
- Atomically release *mutex* and cause the calling thread to block on the condition variable *cond*
- It would wait until some thread use pthread_signal to notify it that the condition is satisfied.
- It will release the *mutex* while waiting and automatically regain it when awaken.

pthread_cond_signal

```
pthread_mutex_lock(&mutex);
      a = b;
      pthread_cond_signal(&cond_var);
pthread_mutex_unlock(&mutex);
```

- pthread_cond_signal() signals one thread waiting on the condition variable.
- pthread_cond_signal() does not release the mutex lock. It is the subsequent call to pthread_mutex_unlock() that releases the mutex.
- Once the mutex lock is released, the signaled thread becomes the owner of the mutex lock and returns control from the call to pthread_cond_wait().

Semaphore

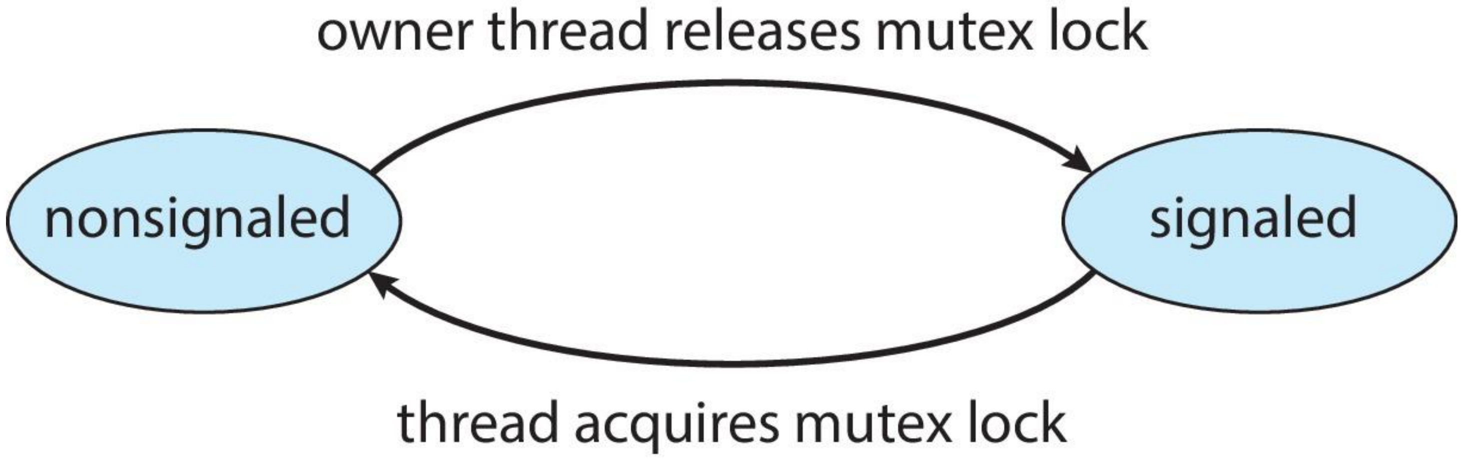
```
int status, pshared;

sem_t sem;

unsigned int initial_value;

status = sem_init(&sem, pshared, initial_value);

status = sem_destroy(&sem);

status = sem_post(&sem);    /* signal*/

status = sem_wait(&sem);   - blocking

status = sem_trywait(&sem); - unblockimg
```

Windows Synchronization

- Kernel accesses a global resources
  - Uses interrupt masks to protect access to global resources on uniprocessor systems
  - Uses spinlocks on multiprocessor systems
- Thread synchronization outside the kernel
  - Also provides **dispatcher objects** which may act as mutexes, semaphores, events, and timers
    - **Events**
      - An event acts much like a condition variable
    - Timers notify one or more thread when time expired
    - Dispatcher objects either **signaled-state** (object available) or **non-signaled state** (thread will block)

Windows Dispatcher Object

- Mutex dispatcher object



Linux Synchronization

- Linux Kernel:
  - Prior to kernel Version 2.6, Linux was a nonpreemptive kernel
  - Version 2.6 and later, fully preemptive kernel

Single Processor	Multiple Processors
Disable Kernel preemption	Acquire spin lock
Enable Kernel preemption	Release spin lock

- Linux kernel provides:
  - semaphores
  - Mutex locks
  - atomic integers `atomic_t counter;`
  - reader-writer versions of both

Summary

- The Critical-Section Problem
- Peterson’s Solution
- Synchronization Hardware
- Semaphores
- Classic Problems of Synchronization
- Monitors