Process Synchronization

These slides were compiled from the OSC textbook slides (Silberschatz, Galvin, and Gagne) and the instructor’s class materials.

Highlights

- Introduction to the Critical-Section (C-S) Problem
- Discuss HW & SW solutions to C-S Problem
- Atomic Transactions and mechanisms
- Synchronization techniques:
  - Mutex
  - Semaphores
  - Monitors

Background

- **Concurrent** access to shared data may result in data **inconsistency**
- Maintaining data **consistency** requires mechanisms to ensure the **orderly** execution of cooperating processes
- **Producer-Consumer** problem:
  - Suppose that we wanted to provide a solution to the consumer-producer problem that fills all the buffers.
  - Keep an integer count that tracks the number of full buffers.
  - Initially, count is set to 0.
  - It is incremented by the producer after it produces a new buffer and is decremented by the consumer after it consumes a buffer.

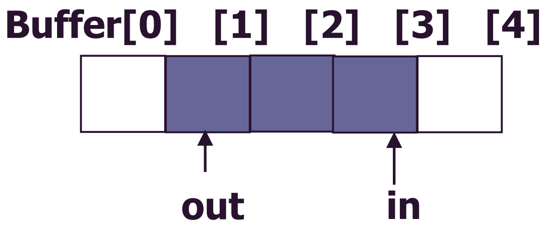
Producer and Consumer Example

roducer Process

or(int i = 0; ; i++)

BoundedBuffer.enter(new Integer(i))

```
public void enter( Object item )
{
   while ( count == BUFFER_SIZE )
     ; // buffer is full! Wait till buffer is consumed
   ++count;
   buffer[in] = item; // add an item
   in = ( in + 1 ) % BUFFER_SIZE;
}

public object remove( )
{
   Object item;
   while ( count == 0 )
     ; // buffer is empty! Wait till buffer is filled
   --count;
   item = buffer[out]; // pick up an item
   out = ( out + 1 ) % BUFFER_SIZE;
}
```



Consumer Process

```
for(int i = 0; ; i++ )
{
  BoundedBuffer.remove( );
}
```

Race Condition Example

Assume count=5, which will be incr/decr

```
++
        reg1 = mem[count];
        reg1 = reg1 + 1;
        mem[count] = reg1;

--
        reg2 = mem[count];
        reg2 = reg2 – 1;
        mem[count] = reg2;
```

```
Producer: reg1 = mem[count];        {reg1=5}
Producer: reg1 = reg1 + 1;          {reg1=6}

Consumer: reg2 = mem[count];        {reg2=5}
Consumer: reg2 = reg2 – 1;          {reg2=4}

Producer: mem[count] = reg1;        {count=6}

Consumer: mem[count] = reg2;        {count=4}
```

The outcome of concurrent thread execution depends on the particular order in which the access takes place = ***race condition***.

Revisit Producer-Consumer Problem

- Producer-Consumer Problem with two threads

```
#include <pthread.h>
#include <iostream>
#define SIZE 10
using namespace std;

class Queue {
private:
  int jobs[SIZE];
  int count, nextIn, nextOut;
public:
  Queue( ) {
    count = nextIn = nextOut = 0;
  }
  void put( int job ) {
    while ( count == SIZE );
    count++;
    jobs[nextIn] = job;
    nextIn = ( nextIn + 1 ) % SIZE;
  }
  int get( ) {
    while ( count == 0 );
    --count;
    int job = jobs[nextOut];
    nextOut = ( nextOut + 1 ) % SIZE;
    return job;
  }
};
```

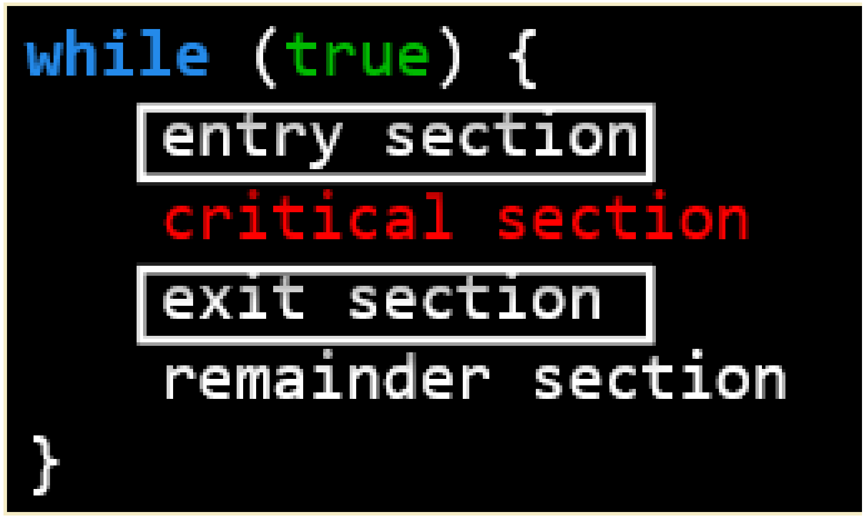
```
void* producer( void *args ) {
  cout << "producer thread: args = " << args << endl;
  Queue *queue = (Queue *)args;
  for ( int i = 0; ; i++ )
    queue->put( i );
}
```

```
void* consumer( void *args ) {
  cout << "consumer thread: args = " << args << endl;
  Queue *queue = (Queue *)args;
  for ( int i = 0; ; i++ ) {
    int job = queue->get( );
    cout << job << endl;
    if ( job != i ) {
      cout << "NAH!!!" << endl;
      exit( -1 );
    }
  }
}
```

```
int main( ) {
  Queue *queue = new Queue( );
  pthread_t producer_t, consumer_t;

  pthread_create( &producer_t, NULL, producer, (void
*)queue );
  pthread_create( &consumer_t, NULL, consumer, (void
*)queue );

  pthread_join( producer_t, NULL );
  pthread_join( consumer_t, NULL );
}
```

The Critical Section (“CS”)

The *critical section* is a block of code in which no two processes can be executing their instructions at the same time.



```
while (true) {
    entry section
    critical section
    exit section
    remainder section
}
```

Critical Section Requirements

1. Mutual Exclusion.

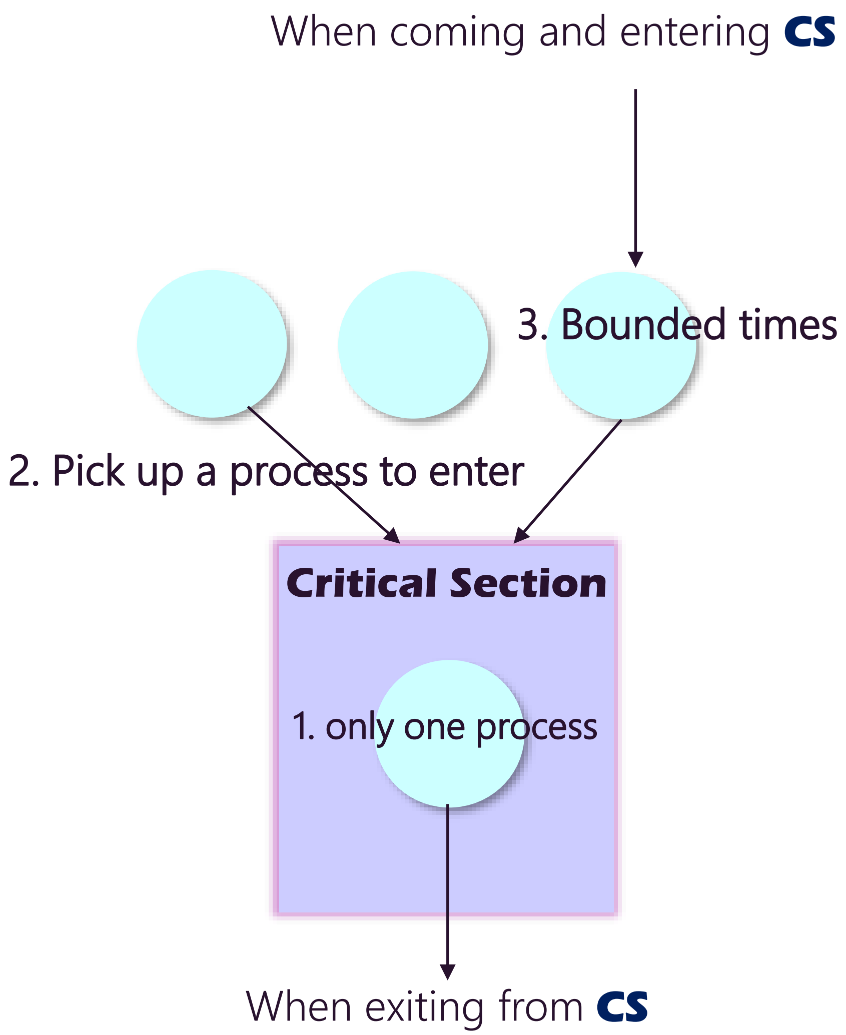
If process *Pi* is executing in its critical section(**CS**) =>no other processes can be executing in their critical sections.

2. Progress.

If no process is executing in its **CS** && there exist some processes that wish to enter their **CS** => the selection of the processes that will enter the **CS** next cannot be postponed indefinitely. Only processes not executing "remainder" section can participate

3. Bounded Waiting.

A bound must exist on the number of times that other processes are allowed to enter their **CS**- after a process has made a request to enter its **CS** and before that request is granted.



Solutions

- **User** level algorithms, e.g.,
  - Peterson
- **OS** level synchronization primitives
  - Semaphores
- **Language** level synchronization primitives
  - Monitor
- **Hardware** synchronization
  - Interrupt making, test_and_set, compare_and_swap

Solutions

- **User** level algorithms
- **OS** level synchronization primitives
- **Language** level synchronization primitives
- **Hardware** synchronization

Keep in mind…

Valid solutions to the Critical section problem must demonstrate that:

① Mutual Exclusion is preserved.
② Progress requirement is satisfied.
③ Bounded Waiting requirement is met

Kind reminders and logistics

- Please use your UW email, do not use your personal email
  - See communication guidelines in the Notebook
- When submitting your in-class discussions, include your FULL NAME, our graders have a hard time finding the right student (e.g., there are 3 Steves)
- Canvas will mark your assignment as LATE even if you submit at 11:59
- Always plan on things going wrong right before the submission deadline

Algorithm 1: Yielding by Turn

```
int turn = 0; // or 1
      producer
t0:
  while ( turn != 0 )
        // busy wait ;
  // critical section
  turn = 1;
```

```
      consumer
t1:
  while ( turn != 1 )
        // busy wait ;
  // ciritical section
  turn = 0;
```

What if producer is t1?

Violates **CS** rules 2,3 – *progress*

Both thread 0 and 1 cannot enter **CS** consecutively.

Algorithms 2: Declare "I'm using"

```
bool flag[0] = false;
bool flag[1] = false;

t0:
   flag[0] = true;
   while ( flag[1] == true )
        // busy wait;
   // critical section
   flag[0] = false;

t1:
  flag[1] = true;
   while ( flag[0] == true )
        // busy wait;
   // critical section
   flag[1] = false;
```

What if both t0 and t1 are declared simultaneously?

1. Thread 0 sets flag[0] true
2. A context switch occurs
3. Thread 1 sets flag[1] true
4. Thread 1 sees flag[0] is true, and waits for Thread 0
5. A context switch occurs
6. Thread 0 sees flag[1] is true, and waits for Thread 1

Violates **CS** rules 2,3 – *progress*

Both thread 0 and 1 cannot enter **CS** consecutively.

Algorithm 3: Dekker's Algorithm

- Mix of 1 and 2
- Declaring "I'm using" as well as yielding by turn
- Available for two processes

Still confused?
Try working it out
on your own…
On paper

```
int turn = 0; // or 1
bool flag[0] = false;  // wants to enter
bool flag[1] = false;

t0:
  flag[0] = true;
  while ( flag[1] == true ) {
    if ( turn != 0 ) {
      flag[0] = false;
      while ( turn != 0 )
            // busy wait ;
      flag[0] = true;
    }
  }

  // critical section

  turn = 1;
  flag[0] = false;
```

```
t1:
  flag[1] = true;
  while ( flag[0] == true ) {
   if ( turn != 1 ) {
      flag[1] = false;
      while ( turn != 1 )
             // busy wait ;
      flag[1] = true;
    }
  }

  // critical section

  turn = 0;
  flag[1] = false;
```

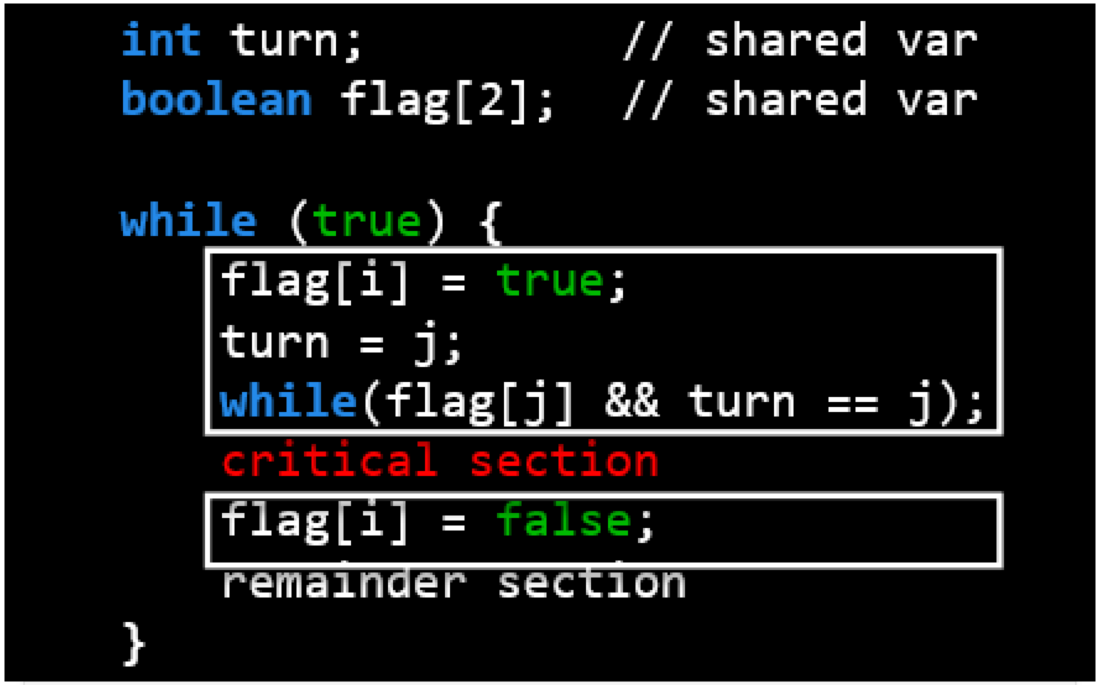
Declare I want to enter
if counterpart is using (wants to enter)
if my turn, wait for the other to turn-down declaration
otherwise, wait till it is my turn

Complies with **CS** rules 2, 3 – progress

- Even if both threads declared to enter **CS** , *turn* eventually points to either A or B!

Peterson's Solution (Algorithm)

P_i, $i=0$
P_j, $j=1$



```
int turn;            // shared var
boolean flag[2];     // shared var

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

Must demonstrate:

① Mutual Exclusion is preserved.
② Progress requirement is satisfied.
③ Bounded Waiting requirement is met

Peterson's algorithm (solution)

G.L. Peterson White Paper

No Guarantees!

There no guarantees that Peterson’s solution will work correctly on modern computer architectures due to complex load and store instructions

Notes on Software Solutions

- Various algorithms like Peterson's solution...
  - Work only for a pair of threads
  - How about a mutual execution among three or more threads? Check Lamport's Algorithm (See Appendix).
- Interrupt Masking
  - It disables even ***time interrupts***, thus **not** allowing ***preemption***.
  - Not broadly scalable (ok for uniprocessor systems).
  - Malicious user program may hog CPU forever.

Busy Waiting?

- "Spinlocks"
  - Thread acquires the lock and waits in a loop repeatedly checking
  - Continuous waiting
    - CON: Uses CPU cycles
    - PRO: Doesn't require context switch
    - Use if wait time is short (because preemption would be expensive)

Solutions

- **User** level algorithms
- **OS** level synchronization primitives
- **Language** level synchronization primitives
- **Hardware** synchronization

Hardware Solutions

- Many systems provide hardware support for critical section
- Modern machines provide special atomic hardware instructions
- Atomic (non-interruptible) set of instructions provided
  - test_and_set (or read_modify_write)
  - compare_and_swap
- They are an atomic combination of memory read and write operations, e.g.,
  - Either test memory word and set value
  - Or swap contents of two memory words

Lock using Test and Set

- Atomic operation:
  - Test the value of **flag**
    - If **flag** set, leave it (and wait till it is reset by the other)
    - Else set it (i.e., enter CS)

- Example:

```
while ( testAndSet( flag ) == true )
    // busy wait ;
// enter CS
```

Solutions

- **User** level algorithms
- **OS** level synchronization primitives
- **Language** level synchronization primitives
- **Hardware** synchronization

Synchronization Primitives

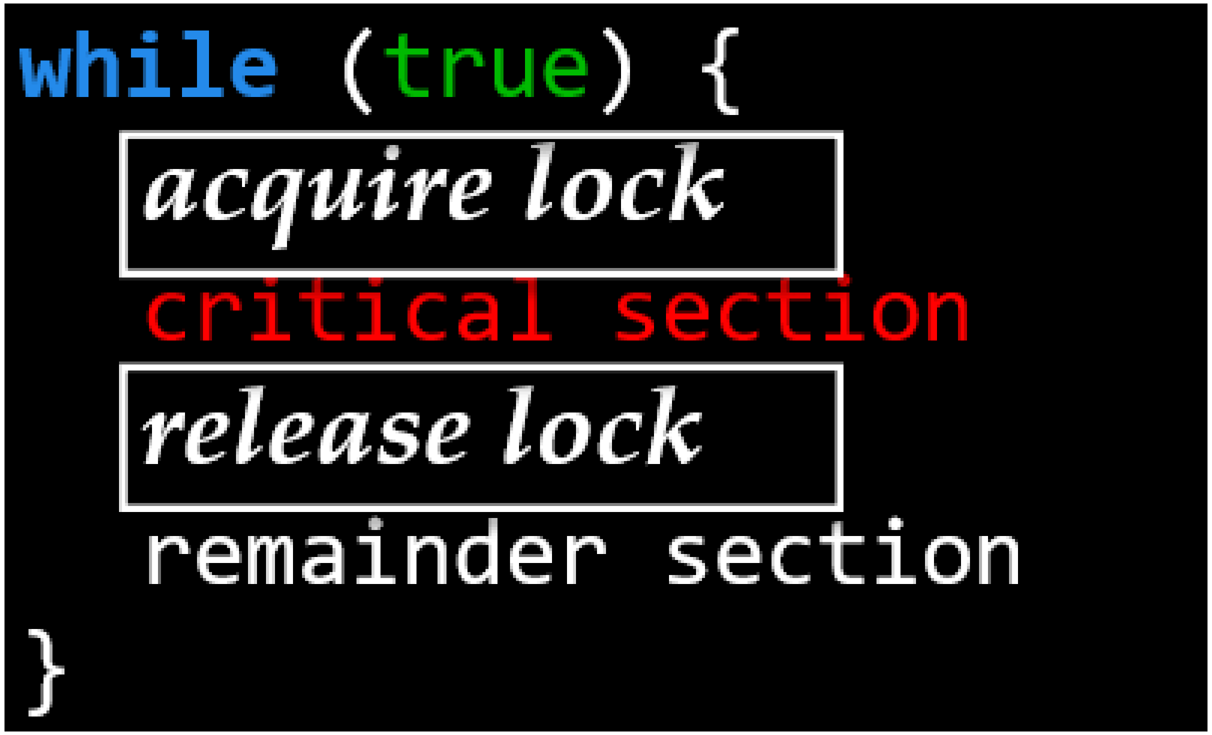
Locks

Mutexes

Semaphores

Monitors

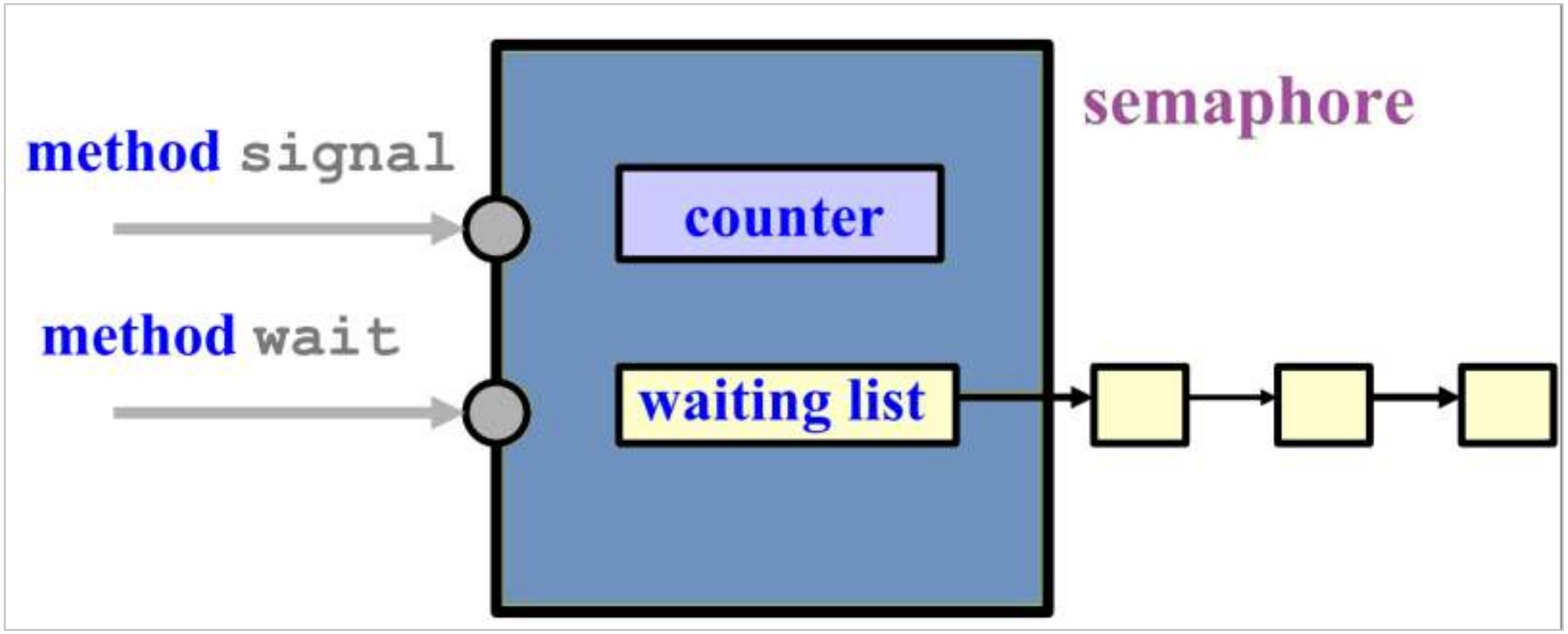
Recall Critical Section (“CS”) *Locks*



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

Semaphore

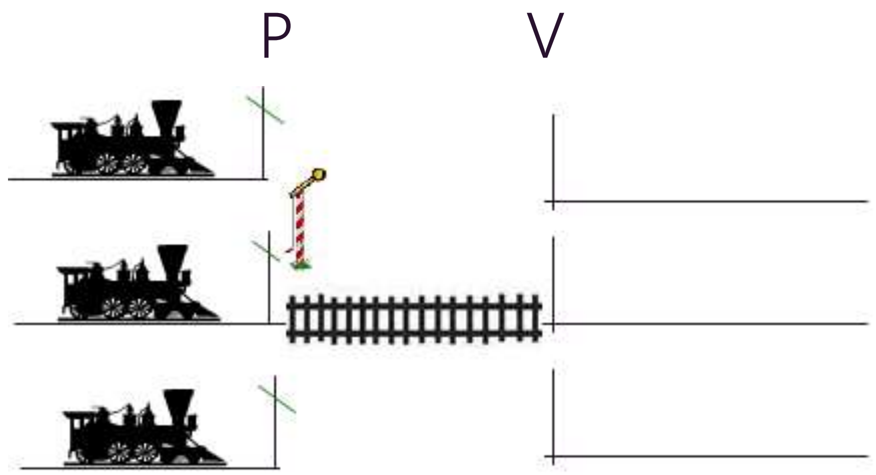
- A semaphore is an object that consists of a ***counter***, a ***waiting list*** of processes and two methods (e.g., functions): ***signal*** and ***wait***.

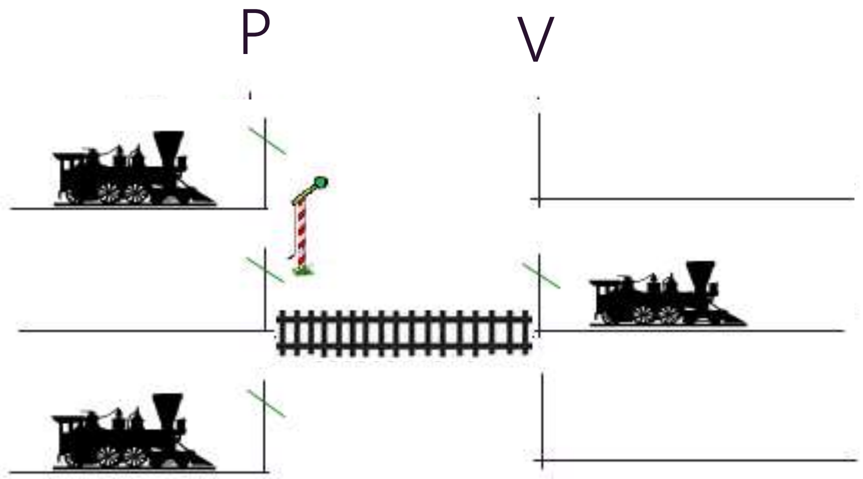


Semaphore (OS-Level solution)

- Synchronization tool that does not require busy waiting at a user level
- Semaphore *S* – integer variable

- Two standard indivisible operations modify *S*: `acquire()` and `release()`
  - Originally called `P()` and `V()` [Dutch P proberen, meaning "to test " and V from verhogen, meaning "to increment"]
- A type of lock initialized to *n*, where *n* is the number of threads that have access to it (or number of "wake-ups" needed)
- Checking and changing the value and going to sleep are done *atomically*





Semaphores

- Mutual exclusion using a (binary) semaphore

```
 Semaphore mutex; // initialized to 1
 do {
    wait (mutex);
    // Critical Section
    signal (mutex);
    // remainder section
 } while (TRUE);
```

- Implementation of wait:

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

- Implementation of signal:

```
signal(semaphore *S) {
   S->value++;
   if (S->value <= 0) {
      remove process/thread P from S->list;
      wakeup(P);
   }
}
```

A	B	C
A.wait() S-- : S=0 S<0 ? ✘ A.CS	B.wait() S-- : S=-1 S<0 ? ✓ B blocked	C.wait() S-- : S=-2 S<0 ? ✓ C blocked
A.signal() S++: S=-1 S<=0 ? ✓ wakeup (B)	B.CS	
A.remainder	B.signal() S++ : S=0 S<=0 ? ✓ wakeup (C)	C.CS
	B.remainder	C.signal() S++ : S=1 S<=0 ? ✘ C.remainder

Problems with Semaphores

Correct → acquire() ... release()

Incorrect → release () ... acquire()

Omitting either acquire() or release()

Consider this producer-consumer scenario:

- Reverse order of acquires in the producer
  - semaphore value decremented before `empty`
  - If buffer is full, `producer` would block with `mutex`=0
  - Next time `consumer` tries to access the buffer, it acquires on a value which is now 0, so it would block too
- → Processes are blocked forever!

Solutions

- **User** level algorithms
- **OS** level synchronization primitives
- **Language** level synchronization primitives
- **Hardware** synchronization

Why talk about threads?

- Any modern multi/many core processor uses threads
- Threads is the primary way of implementing and managing parallelism
- Parallelism is at the core of HPC
- While many libraries hide thread management from the programmer, it is crucial to understand what is happening “under the hood”

Synchronization Primitives

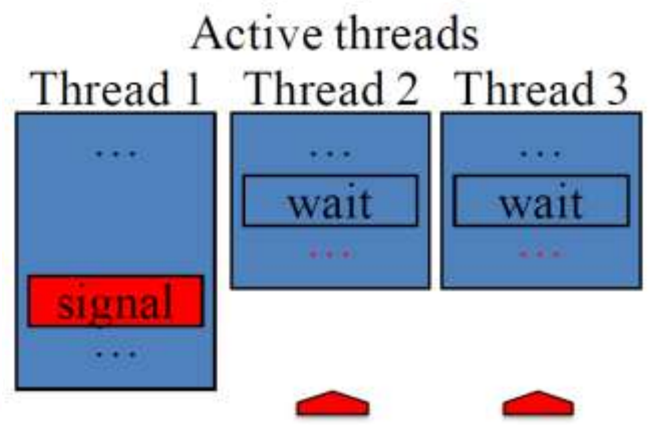
- Mutex (mutual exclusion)
  - Simple lock primitive with 2 states: **lock** and **unlock**
  - Only one thread can lock the mutex.

- Spin vs. sleep locks? -- lock granularity
  - Fine grain – spin mutex
  - Coarse grain – sleep mutex
  - Spin mutex: use CPU cycles and increase the memory bandwidth, but when the mutex is unlocked the thread can continue execution immediately

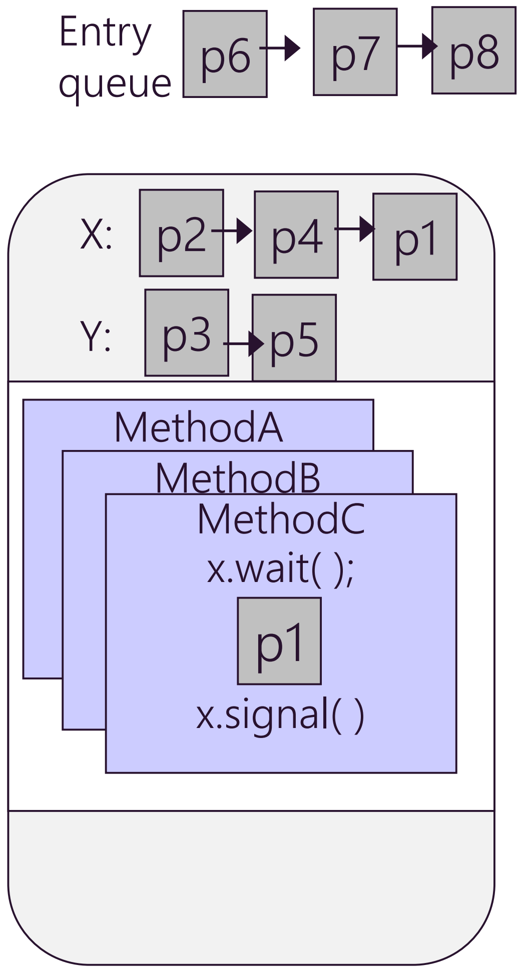
- Shared/Exclusive locks: extension for readers/writer model
  - Multiple threads can hold a shared (reading) lock simultaneously
  - Only one thread can hold exclusive (writing) lock at a time

Synchronization Primitives

- Condition Variable
  - Blocks a thread while waiting for a condition
  - Condition_wait / condition_signal
  - Several threads can wait for the same condition,
    - When the condition is fulfilled, they all get the signal
  - Or, they could wait on different conditions



Monitors



High-level language construct

- Only one process allowed in a **monitor**, thus executing its method
- A process in the **monitor** can wait on a **condition variable**
  - Thus relinquishing the **monitor** and allowing another process to enter
- A process can **signal** another process waiting on a **condition variable** (e.g., X or Y).
- A process signaling another process should exit from the **monitor** , because the signaled process may have begun to work in the **monitor** .

Condition Variable (general)

`pthread_cond_t x;`

- The only operations that can be invoked on a condition variable are **`wait()`** and **`signal()`**

- The operation

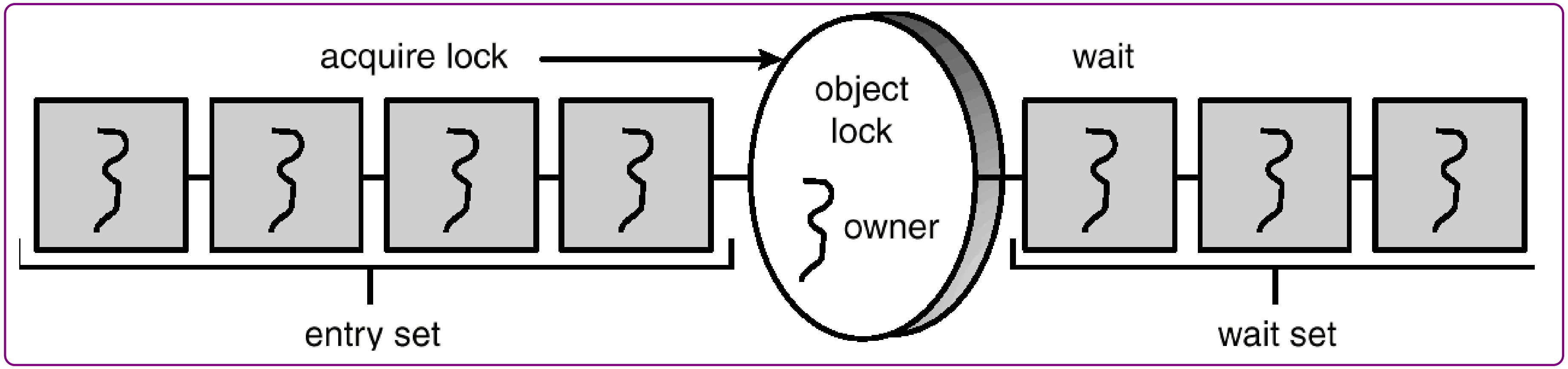
  `pthread_cond_wait();`

means the thread invoking is suspended until another thread invokes

`pthread_cond_signal();`

- CVs are not counters – don't accumulate signals like semaphores
  - Signaling a CV no one is waiting on causes the signal to be lost!
- **`wait`** must come before **`signal`** signal

Case Study: Java Monitor –"synchronized"



```
public void synchronized method1 ( ) {
  // If a calling thread does not own the lock, it is placed in the entry set.
  while ( condition == false )
    try {
  // The thread releases a lock and places itself in the wait set.
       wait ( );
    } catch ( InterruptedException e ) { }
  }
   ...;
  // The calling thread resumes one of threads waiting in the wait set.
  notify ( );
}
```

- *Every object has an <u>intrinsic lock</u> associated with it.*
- *Calling a synchronized method requires "owning" the lock.*

Case Study: Java Monitor -"synchronized"

- The call to `wait()` moves a thread into the wait set.
- The call to `notify()` selects an arbitrary thread from the wait set.
  - It is possible the selected thread is in fact not waiting upon the condition for which it was notified
- The call `notifyAll()` selects all threads in the wait set and moves them to the entry set.
- In general, `notifyAll()` is a more *conservative* strategy than `notify()`

Case Study: Java Monitor –“synchronized”

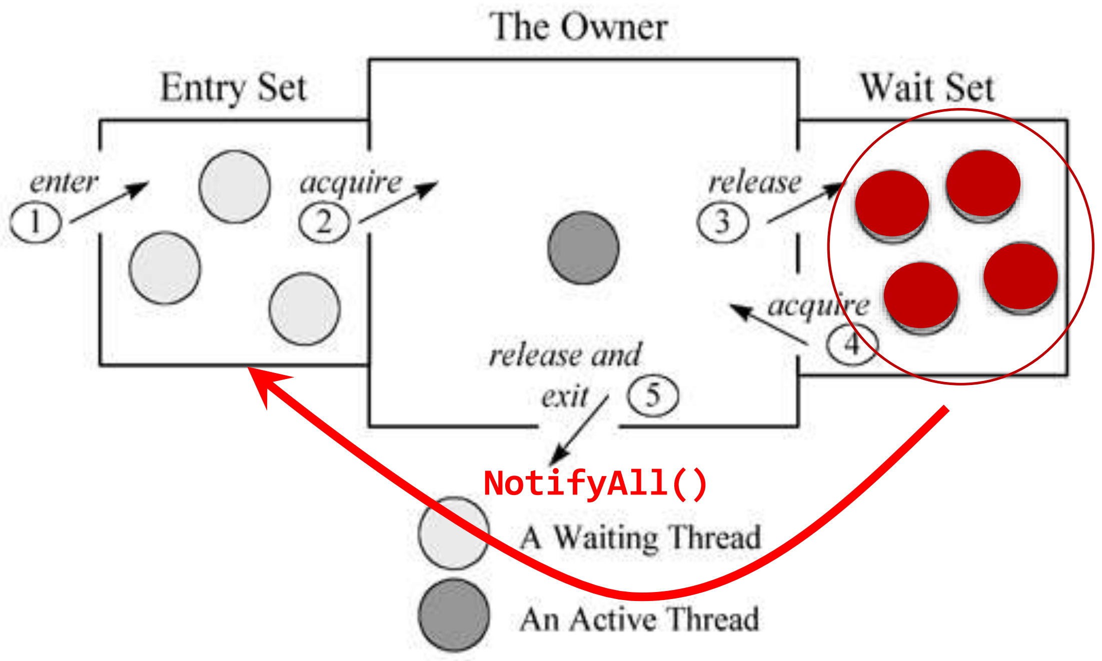


Figure 20-1. A Java monitor.

Discussion

1. Non-interruptible execution of CPU instructions is not enough to implement TestAndSet and Swap. Why? What else should hardware support?

2. Can you implement P and V functions using the TestAndSet instruction? If so, how? Briefly design the algorithm your algorithm.

3. Fill out the following table.

	Advantage	Disadvantage	Implementation (HW, OS, or Language)
Test and set Swap			
Semaphore			
Monitor			

Deadlock and Starvation

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

```
        P0          P1
        P(S);       P(Q); { S.value and Q.value are both decr to zero }
        P(Q);       P(S);  { S and Q are both deadlocked (can't be released) }
         ⋮           ⋮
        V(Q);       V(S);  { S and Q release methods can not run }
        V(S);       V(Q);
```

- **Starvation** – indefinite blocking. A process may never be removed from the *semaphore queue* in which it is suspended.
  - What if processes are waiting at P(S) in LIFO order?

Concurrent Programming

"Programming concurrent applications is a difficult and error-prone undertaking" – Dietel**

When thread synchronization is required, you can use the following (in order of complexity):

1. Use existing classes from a language (e.g., Java, C#) API

2. Use synchronized keyword and Object methods `wait`, `notify` and `notifyAll`

3. Use `Lock` and `Condition` interfaces

*** (p.678) Java for Programmers, Deitel & Deitel, 2nd Edition, Prentice Hall, 2011*

Synchronization Classic Problems

① Bounded-Buffer Problem

② Readers and Writers Problem

③ Dining-Philosophers Problem

Recall: Bounded Buffer Problem

Producer Process

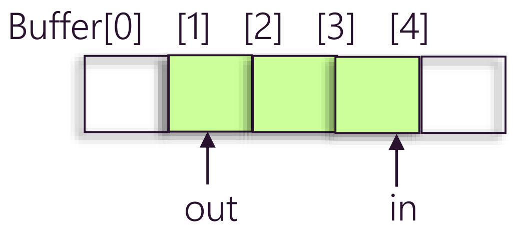
```
for(int i = 0; ; i++ )
{
   BoundedBuffer.enter(new Integer(i));
}
```

```
public void enter( Object item ) {
   while ( count == BUFFER_SIZE )
      ; // buffer is full! Wait till buffer is consumed
   ++count;
   buffer[in] = item; // add an item
   in = ( in + 1 ) % BUFFER_SIZE; // circular buffer
}
```

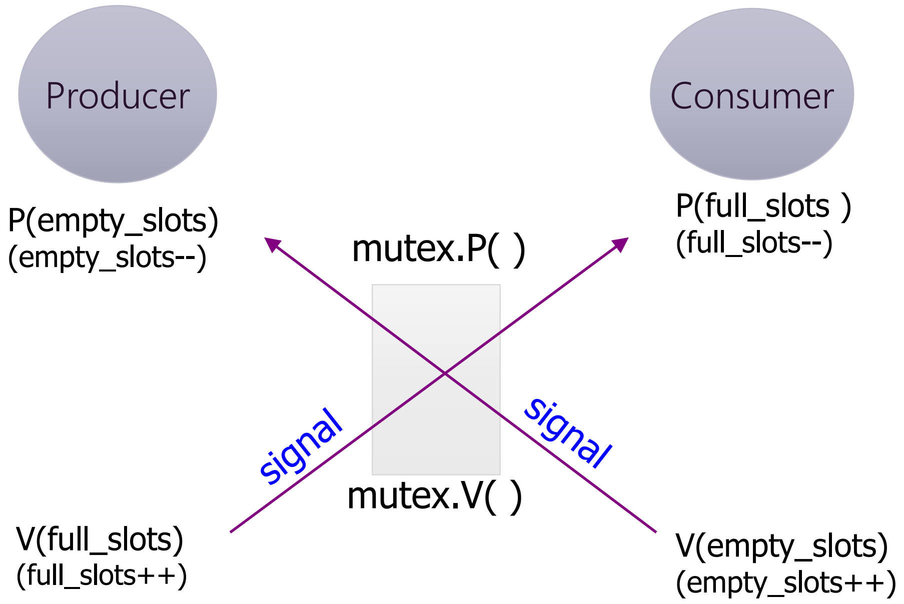
Consumer Process

```
for(int i = 0; ; i++ )
{
   BoundedBuffer.remove();
}
```

```
public object remove( ) {
   Object item;
   while ( count == 0 )
      ; // buffer is empty! Wait till buffer is filled
   -- count;
   item = buffer[out]; // pick up an item
   out = ( out + 1 ) % BUFFER_SIZE; // circular buffer
}
```



Bounded Buffer Problem



```
producer()
{   // push object into the buffer
    while(there are items to produce)
      insert Object item into buffer;
}
```

```
consumer()
{   // fetch item from the buffer
    while(there are items to consume)
      remove an Object item from buffer;
}
```

Semaphores Pseudocode

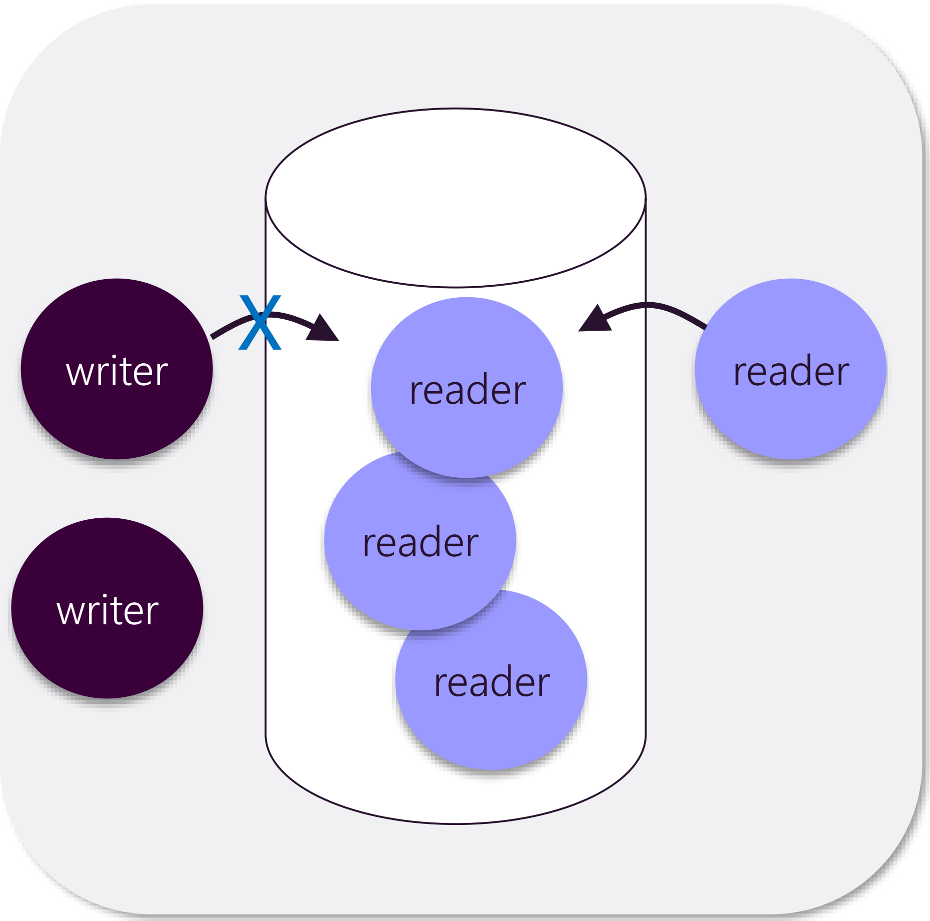
```
in = 0;  out = 0;
buffer     Shared buffer can store BUFFER_SIZE objects,
           initially empty
mutex      Semaphore mutex initialized to 1
empty_slots  Semaphore initialized to BUFFER_SIZE
full_slots   Semaphore full initialized to 0
interface insert, remove
           wait / P, signal / V
```

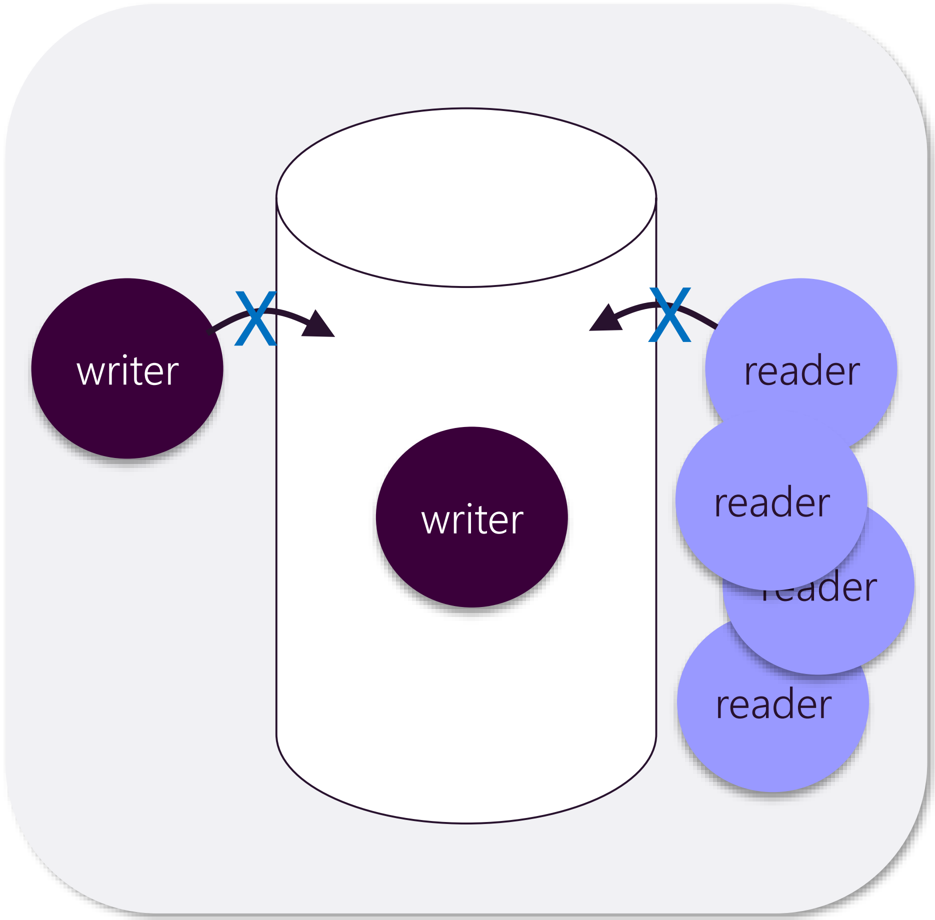
```
void insert(Object item)
{
    wait(empty_slots);    // P
    wait(mutex);
    //add item to buffer
    buffer[in] = item;
    in = (in + 1) % BUFFER_SIZE;
    signal(mutex);
    signal(full_slots); // V
}
```

```
Object remove()
{
    wait(full_slots);    // P
    wait(mutex);
    //add item to buffer
    Object item = buffer[in];
    out = (out + 1) % BUFFER_SIZE;
    signal(mutex);
    signal(empty_slots); // V
}
```

Classic Problem 2: The Readers-Writers Problem

- Multiple readers or a single writer can use DB.





Readers-Writers Problem (Semaphore)

- A data set is shared among a number of concurrent processes
  - Readers – only read the data set; do NOT perform and updates
  - Writers – can 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 (example)
  - The Data
  - Semaphore **mutex** initialized to 1
  - Semaphore db initialized to 1
  - Integer **`readerCount`** initialized to 0

Readers-Writers Problem Overview

Pseudocode using semaphores for DB example

```
mutex     Semaphore mutex initialized to 1
db        Semaphore initialized to 1
interface read, write, acquireReadLock, acquireWriteLock,
          releaseReadLock, releaseWriteLock
```

```
reader()
{
  acquireReadLock();
  read(); //from DB
  releaseReadLock();
}
```

```
writer()
{
  acquireWriteLock();
  write(stuff); // to DB
  releaseWriteLock();
}
```

Wrapping the semaphore logic

```
acquireReadLock()
{
  wait(mutex);
  readerCount++;
  if (readerCount == 1)
    wait(db);  // P
  signal(mutex);
}
```

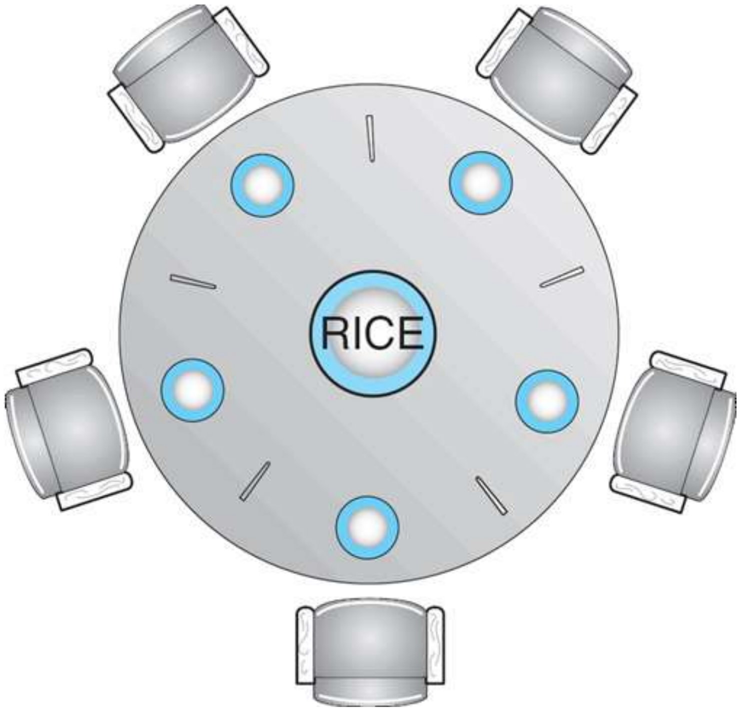
```
releaseReadLock()
{
  wait(mutex);
  readerCount--;
  if (readerCount <= 0)
    signal(db);  // V
  signal(mutex);
}
```

Can you write the logic for the Write locks?

```
acquireWriteLock()
{
  wait(db); // P
}
```

```
releaseWriteLock()
{
  signal(db); // V
}
```

Classic Problem 3: Dining Philosophers Problem



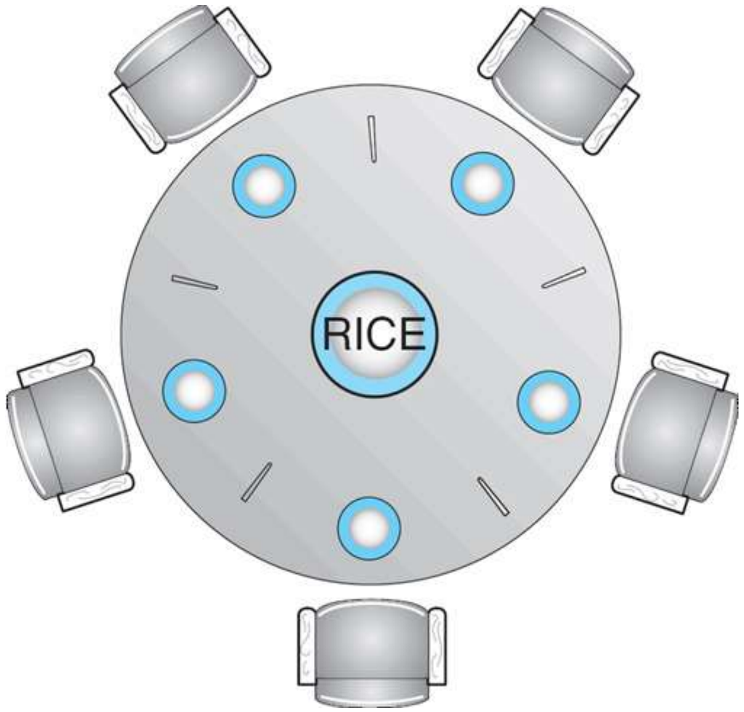
In the inimitable words of my OS professor, Dr. Wolski:

*``Since these are either unwashed, stubborn and deeply committed philosophers or unwashed, clueless, and basically helpless philosophers, there is a possibility for **deadlock**.*

*In particular, if all philosophers simultaneously grab the chopstick on their left and then reach for the chopstick on their right (waiting until one is available) before eating, they will all **starve**.*

*The **challenge** in the dining philosophers problem is to design a protocol so that the philosophers do not deadlock (i.e. the entire set of philosophers does not stop and wait indefinitely), and so that no philosopher starves (i.e. every philosopher eventually gets his/her hands on a pair of chopsticks).''*

Classic Problem 3:
Dining Philosophers Problem



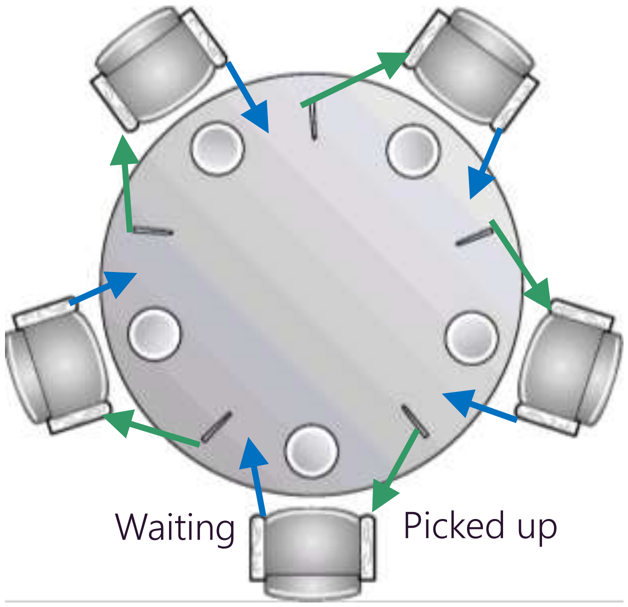
A philosopher can be in 1 of 3 states:

1. THINKING
2. HUNGRY
3. EATING

One approach is: Chopstick is the shared data -- create a semaphore for each

Approach: Synchronize Chopsticks

Structure of Philosopher *i*

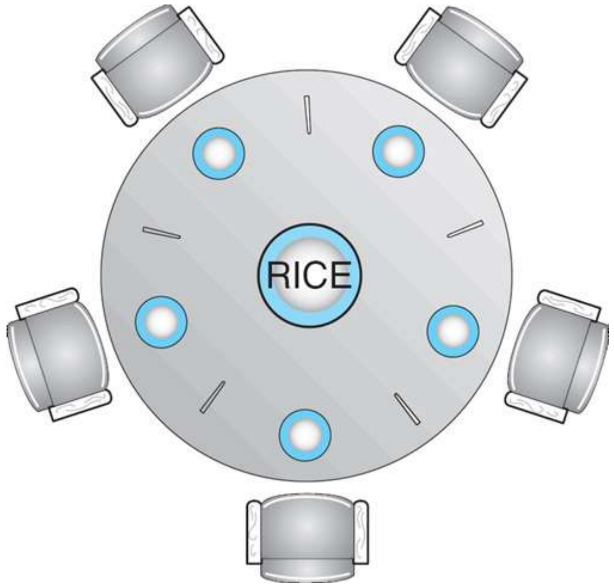


A deadlock occurs!

```
while ( true )
{
  // get left chopstick
  chopStick[i].P();
  // get right chopstick
  chopStick[(i + 1) % 5].P();

  // eat for a while

  //return left chopstick
   chopStick[i].V( );
  // return right chopstick
   chopStick[(i + 1) % 5].V( );

  // think for a while
}
```

Approach: An Asymmetrical Solution

Odd philosopher pick left first, and even philosophers pick right first



```
while ( true )
{ //get chopsticks
  if(i % 2 == 1){
     chopStick[i].P(); // left
     chopStick[(i + 1) % 5].P(); // right
  }
  else {
    chopStick[(i + 1) % 5].P(); // right
    chopStick[i].P(); // left
  }
  // eat for a while

  //return chopsticks
  if(i % 2 == 1){
     chopStick[(i + 1) % 5].V( ); // right
     chopStick[i].V( ); // left
  }
  else {
    chopStick[i].V( ); // left
    chopStick[(i + 1) % 5].V( ); // right
   }
   // think for a while
}
```

Starvation happens!
Thread system: suppose 0 is waiting for 1, when 1 is done there is no guarantee that 0 will get it before 1's thread is rescheduled.
But also because it is unfairly weighed

Dining-Philosophers Problem Using a Monitor

Addressing the deadlock issue:
Approach is to keep track of the philosophers instead of the chopsticks

```
class DiningPhilosophers {
private:
  pthread_mutex_t lock;
  enum State { THINKING, HUNGRY, EATING } state[MAX];
  pthread_cond_t self[MAX];

  void test( int i ) {
    // if phi-i's L is not eating,
    // phi-i  is hungry, and
    // phi-i's right is not eating,
    // then phi-i  can eat!
    // Wake up phi-i
    if ( (state[(i + MAX-1 ) % MAX] != EATING ) &&
         ( state[i] == HUNGRY ) &&
         ( state[ ( i + 1 ) % MAX ] != EATING ) ) {
      state[ i ] = EATING;
      pthread_cond_signal( &self[ i ] );
    }
  }

public:
  DiningPhilosophers( ) {
    pthread_mutex_init( &lock, NULL );
    for ( int i = 0; i < MAX; i++ ) {
      pthread_cond_init( &self[i], NULL );
      state[i] = THINKING;
    }
  }
```

```
 void pickUp( int i ) {
    pthread_mutex_lock( &lock );

    state[ i ] = HUNGRY;   // I got hungry
    test( i );             // Can I have my L and R chopsticks?
    if ( state[ i ] != EATING ) // I can't => I should wait
      pthread_cond_wait(
             &self[ i ], &lock );
    cout << "philosopher[" << i
             << "] picked up chopsticks."<< endl;

    pthread_mutex_unlock( &lock );
  }


  void putDown( int i ) {
    pthread_mutex_lock( &lock );

    cout << "philosopher[" << i
             << "] put down chopsticks."<< endl;
    state[ i ] = THINKING; // I'm stuffed and now thinking.
                           // test L and R neighbors
    test( ( i + MAX - 1 ) % MAX );  // if possible, wake up my L
    test( ( i + 1 ) % MAX );        // if possible, wake up my R

    pthread_mutex_unlock( &lock );
  }
}
```

More on the Dining-Philosophers Monitor Implementation

- The previous solution prevents deadlock, however…
- In some pathological cases, it may lead to starvation
- What can we do to prevent starvation?
  - How about a queue?
  - Deli ticket approach: pick a number, lowest number eats next
  - Mixed approach? (check how long neighbor has been waiting)

Take-away's

1. When multiple threads or processes access multiple resources exclusively, you must worry about deadlock.

2. You must worry about starvation, and the only way to prevent starvation is to enforce that all threads/processes get unblocked every now and then.
   - This can be using a global queue, or some other ordering strategy,
   - Sometimes you have to be less aggressive about preventing starvation in order to get both good performance, and no starvation.

3. Often you have to worry about treating all threads equally, so that no one thread gets more resources than the others due to your synchronization protocol. This is a problem with an asymmetric solution.

Discussion

- Newer Java compilers have deprecated the use of resume, suspend, and stop. What are these methods? Why do you think they have been deprecated? (Consider an undesired situation incurred when those three are used.)

Discussion

Consider the following five options to implement synchronization between producer and a consumer, both accessing the same bounded buffer. When we run a producer and a consumer on shared-memory-based dual-processor computer, which of the following implementation is the fastest? Justify your selection. Also select the slowest implementation and justify your selection.

(1) User the many-to-one thread mapping model, allocate a user thread to a producer and a consumer respectively, and let them synchronize with each other using test-and-set instructions.
(2) Use the many-to-one thread mapping model, allocate a user thread to a producer and a consumer respectively, and let them synchronize with each other using semaphore.
(3) User the one-to-one thread mapping model, allocate a user thread, (i.e., a kernel thread) to a producer and a consumer respectively, and let them synchronize with each other using test-and-set instructions.
(4) User the one-to-one thread mapping model, allocate a user thread, (i.e., a kernel thread) to a producer and a consumer respectively, and let them synchronize with each other using semaphores.
(5) Allocate a different process to a producer and a consumer respectively, and let them synchronize with each other using semaphores. (Note that a bounded buffer is mapped onto the shared memory allocate by those processes through shmget and shmat.)