Operating System

Chapter 5 synchronization

Objectives

- 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

Motivation

- Cooperating process/thread:
  - the one that can affect or be affected by other processes executing in system.
  - Processes, threads
- Processes can execute concurrently
  - May be interrupted at any time, partially completing execution
- Problem: Data inconsistency
  - It may occur in the case of concurrent access to shared data
- How to solve?
  - Orderly execution of cooperating processes that share a logical address space

Circular buffer & producer-consumer problem

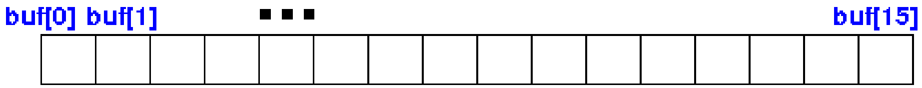
```
#define BUFFER_SIZE 16
typedef struct {
   . . .
} item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

```
item next_produced;
while (true) {
        /* produce an item in next produced */
        while (((in + 1) % BUFFER_SIZE) == out)
                ; /* do nothing */
        buffer[in] = next_produced;
        in = (in + 1) % BUFFER_SIZE;
}
```

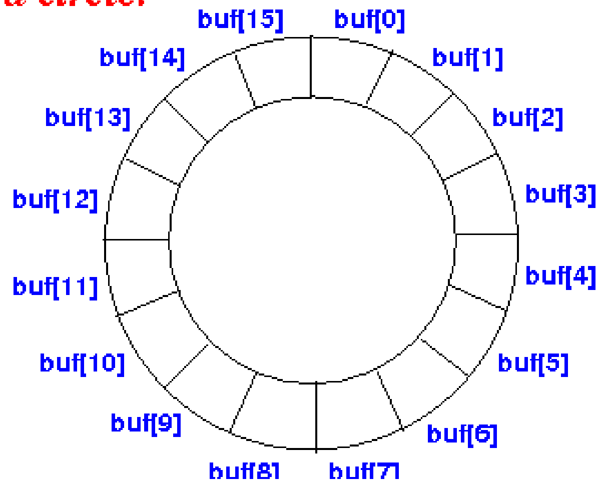
```
item next_consumed;

while (true) {
        while (in == out) ; /* do
nothing */

        next_consumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;

        /* consume the item in next
consumed */
}
```

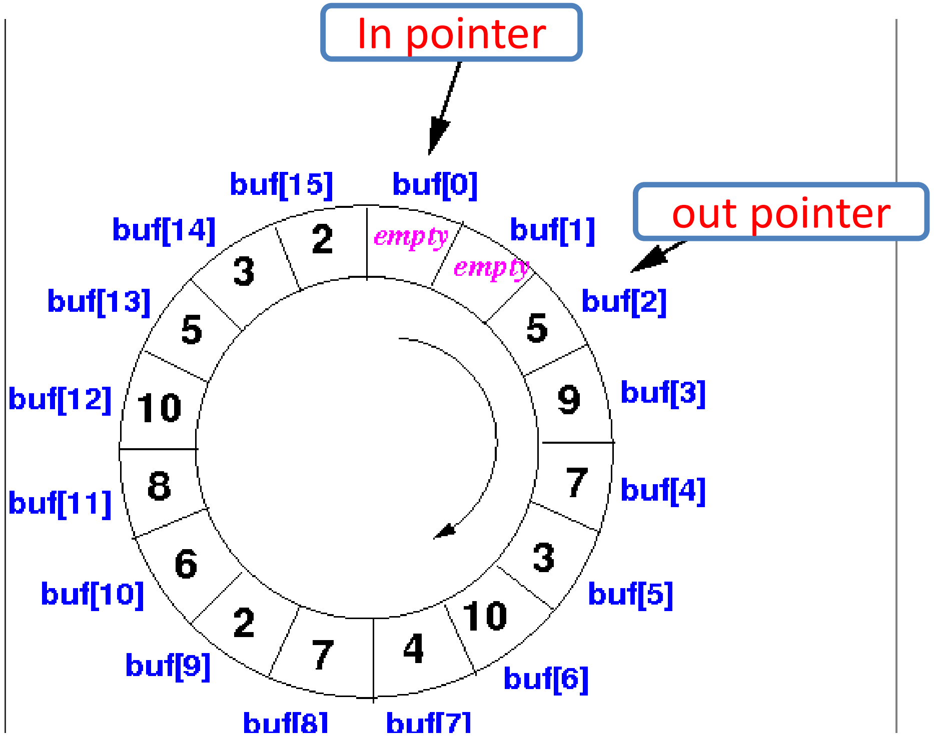
circular array

Array:



Pretend array is a circle:





One solution!

- A solution to consumer-producer problem that fills ***all*** the buffers.
  - We can have an integer **`counter`** that keeps track of the number of full buffers.
  - Initially, **`counter`** is set to 0.
  - It is incremented by the producer after it produces a new buffer
  - It is decremented by the consumer after it consumes a buffer.

Circular buffer & producer-consumer problem

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

Producer

```
item next_produced;
while (true) {
        /* produce an item in next produced */
        while (counter == BUFFER_SIZE)) ;
          /* do nothing */
        buffer[in] = next_produced;
        in = (in + 1) % BUFFER_SIZE;
        counter ++;
}
```

Consumer

```
item next_consumed;
while (true) {
        while (counter == 0);
            /* do nothing */

        next_consumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        counter --;
        /* consume the item in next
consumed */
}
```

Race condition

- **`counter++`** could be implemented as

```
register1 = counter                 MOV    AX, [100]
register1 = register1 + 1           ADD    AX, 1
counter = register1                 MOV   [100], AX
```

- **`counter--`** could be implemented as

```
register2 = counter                 MOV     BX, [100]
register2 = register2 - 1           ADD     BX, 1
counter = register2                 MOV     [100], BX
```

- Consider this execution interleaving with “count = 5” initially:

S0: producer execute `register1 = counter`	{register1 = 5}
S1: producer execute `register1 = register1 + 1`	{register1 = 6}
S2: consumer execute `register2 = counter`	{register2 = 5}
S3: consumer execute `register2 = register2 - 1`	{register2 = 4}
S4: producer execute `counter = register1`	{counter = 6 }
S5: consumer execute `counter = register2`	{counter = 4}

Another Race condition ⚠

- Invoking *echo()* procedure:

```
void echo()
{
  chin = getchar();
  chout = chin;
  putchar(chout);
}
```

Process P1	Process P2
•	•
chin = getchar();	•
•	chin = getchar();
chout = chin;	chout = chin;
putchar(chout);	•
•	putchar(chout);
•	•

- Same problem exists on:
  - Multiprogramming environment
  - Multiprocessing environment
  - Distributed processing environment

Problem is at the Lowest Level

- Most of the time, threads are working on separate data, so scheduling doesn't matter:

Thread A	Thread B
x = 1;	y = 2;

- However, what about (Initially, y = 12):

Thread A	Thread B
x = 1;	y = 2;
x = y+1;	y = y*2;

  – What are the possible values of x?
- Or, what are the possible values of x below?

Thread A	Thread B
x = 1;	x = 2;

  – X could be 1 or 2 (non-deterministic!)

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 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

Another Concurrent Program Example

- Two threads, A and B, compete with each other
  - One tries to increment a shared counter
  - The other tries to decrement the counter

Thread A	Thread B
i = 0;	i = 0;
while (i < 10)	while (i > -10)
i = i + 1;	i = i – 1;
printf(“A wins!”);	printf(“B wins!”);

- Assume that memory loads and stores are atomic, but incrementing and decrementing are *not* atomic
- Who wins? Could be either
- Is it guaranteed that someone wins? Why or why not?
- What if both threads have their own CPU running at same speed? Is it guaranteed that it goes on forever?

Other examples?

Have you ever seen other examples?

Motivating Example: "Too Much Milk"

- Great thing about OS's – analogy between problems in OS and problems in real life
  - Help you understand real life problems better
  - But, computers are much stupider than people
- Example: People need to coordinate:

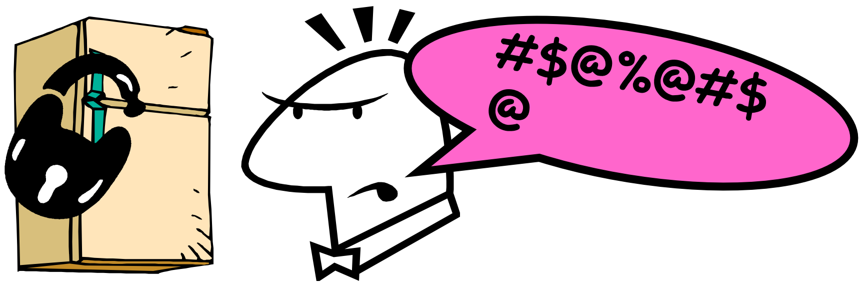
Time	Person A	Person B
3:00	Look in Fridge. Out of milk	
3:05	Leave for store	
3:10	Arrive at store	Look in Fridge. Out of milk
3:15	Buy milk	Leave for store
3:20	Arrive home, put milk away	Arrive at store
3:25		Buy milk
3:30		Arrive home, put milk away

Definitions

- Synchronization: using atomic operations to ensure cooperation between threads
  - For now, only loads and stores are atomic
  - We are going to show that its hard to build anything useful with only reads and writes
- Mutual Exclusion: ensuring that only one thread does a particular thing at a time
  - One thread *excludes* the other while doing its task
- Critical Section: piece of code that only one thread can execute at once. Only one thread at a time will get into this section of code
  - Critical section is the result of mutual exclusion
  - Critical section and mutual exclusion are two ways of describing the same thing

More Definitions

- Lock: prevents someone from doing something
  - Lock before entering critical section and before accessing shared data
  - Unlock when leaving, after accessing shared data
  - Wait if locked
    - Important idea: all synchronization involves waiting
- For example: fix the milk problem by putting a key on the refrigerator
  - Lock it and take key if you are going to go buy milk
  - Fixes too much: roommate angry if only wants OJ



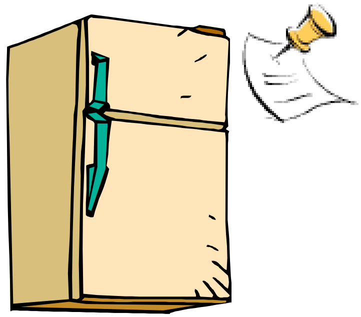
  - Of Course – We don't know how to make a lock yet

Too Much Milk: Correctness Properties

- Need to be careful about correctness of concurrent programs, since non-deterministic
  - Instead, think first, then code
  - Always write down behavior first
- What are the correctness properties for the “Too much milk” problem???
  - Never more than one person buys
  - Someone buys if needed
- Restrict ourselves to use only atomic load and store operations as building blocks

Too Much Milk: Solution #1

- Use a note to avoid buying too much milk:
  - Leave a note before buying (kind of "lock")
  - Remove note after buying (kind of "unlock")
  - Don't buy if note (wait)



- Suppose a computer tries this (remember, only memory read/write are atomic):

```
if (noMilk) {
   if (noNote) {
      leave Note;
      buy milk;
      remove note;
   }
}
```

Too Much Milk: Solution #1

- Use a note to avoid buying too much milk:
  - Leave a note before buying (kind of "lock")
  - Remove note after buying (kind of "unlock")
  - Don't buy if note (wait)
- Suppose a computer tries this (remember, only memory read/write are atomic):

Thread A

```
if (noMilk) {


   if (noNote) {
     leave Note;
  buy Milk;
  remove Note;
   }
}
```

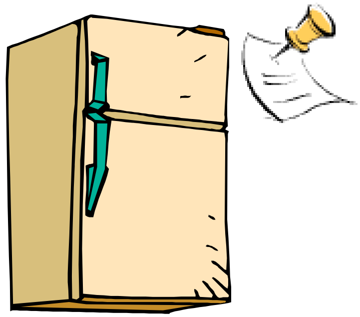
Thread B

```
if (noMilk) {
   if (noNote) {






       leave Note;
      buy Milk;
      remove Note;
   }
}
```

Too Much Milk: Solution #1

- Use a note to avoid buying too much milk:
  - Leave a note before buying (kind of “lock”)
  - Remove note after buying (kind of “unlock”)
  - Don’t buy if note (wait)
- Suppose a computer tries this (remember, only memory read/write are atomic):

```
if (noMilk) {
   if (noNote) {
      leave Note;
      buy milk;
      remove note;
   }
}
```



- Result?
  - Still too much milk but only occasionally!
  - Thread can get context switched after checking milk and note but before buying milk!
- Solution makes problem worse since fails intermittently
  - Makes it really hard to debug…
  - Must work despite what the dispatcher does!

Too Much Milk: Solution #1½

- Clearly the Note is not quite blocking enough
  - Let's try to fix this by placing note first
- Another try at previous solution:


```
leave Note;
if (noMilk) {
   if (noNote) {
      buy milk;
   }
}
remove Note;
```

- What happens here?
  - Well, with human, probably nothing bad
  - With computer: no one ever buys milk

Too Much Milk Solution #2

- How about labeled notes?
  - Now we can leave note before checking
- Algorithm looks like this:

Thread A

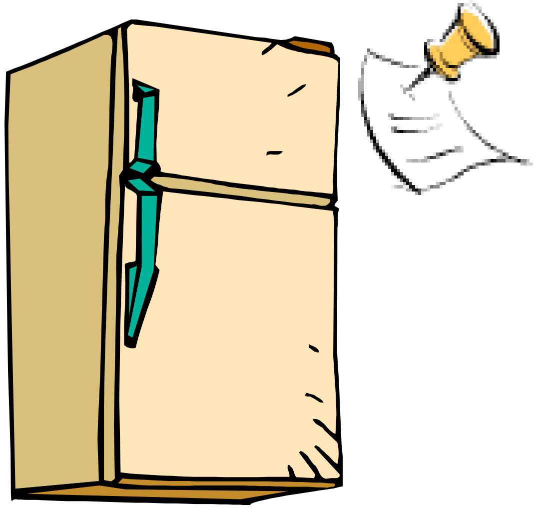
```
leave note A;
if (noNote B) {
   if (noMilk) {
      buy Milk;
   }
}
remove note A;
```

Thread B

```
leave note B;
if (noNoteA) {
   if (noMilk) {
      buy Milk;
   }
}
remove note B;
```

- Does this work?
- Possible for neither thread to buy milk
  - Context switches at exactly the wrong times can lead each to think that the other is going to buy
- Really insidious:
  - Extremely unlikely this would happen, but will at worse possible time
  - Probably something like this in UNIX

Too Much Milk Solution #2: problem!



- *I'm* not getting milk, *You're* getting milk
- This kind of lockup is called "starvation!"

Too Much Milk Solution #3

- Here is a possible two-note solution:

Thread A

```
leave note A;
while (note B) {\\X
   do nothing;
}
if (noMilk) {
   buy milk;
}
remove note A;
```

Thread B

```
leave note B;
if (noNote A) {\\Y
   if (noMilk) {
      buy milk;
   }
}
remove note B;
```

- Does this work? Yes. Both can guarantee that:
  - It is safe to buy, or
  - Other will buy, ok to quit
- At X:
  - If no note B, safe for A to buy,
  - Otherwise wait to find out what will happen
- At Y:
  - If no note A, safe for B to buy
  - Otherwise, A is either buying or waiting for B to quit

Case 1

- "`leave note A`" happens before "`if (noNote A)`"

```
leave note A;
while (note B) {\\X
    do nothing;
};

if (noMilk) {
    buy milk;}
}
remove note A;
```

happened before

```
leave note B;
if (noNote A) {\\Y
    if (noMilk) {
        buy milk;
    }
}
remove note B;
```

Case 1

- "`leave note A`" happens before "`if (noNote A)`"

```
leave note A;
while (note B) {\\X
    do nothing;
};

if (noMilk) {
    buy milk;}
}
remove note A;
```

happened before

```
leave note B;
if (noNote A) {\\Y
    if (noMilk) {
        buy milk;
    }
}
remove note B;
```

Case 1

- “`leave note A`” happens before “`if (noNote A)`”

```
leave note A;
while (note B) {\\X
    do nothing;
};
```

happened before

```
leave note B;
if (noNote A) {\\Y
    if (noMilk) {
        buy milk;
    }
}
remove note B;
```

Wait for note B to be remove

```
if (noMilk) {
    buy milk;}
}
remove note A;
```

Case 2

- “`if (noNote A)`” happens before “`leave note A`”

```
leave note A;
while (note B) {\\X
    do nothing;
};

if (noMilk) {
    buy milk;}
}
remove note A;
```

happened before

```
leave note B;
if (noNote A) {\\Y
    if (noMilk) {
        buy milk;
    }
}
remove note B;
```

Case 2

- "`if (noNote A)`" happens before "`leave note A`"

```
leave note A;
while (note B) {\\X
    do nothing;
};

if (noMilk) {
    buy milk;}
}
remove note A;
```

happened before

```
leave note B;
if (noNote A) {\\Y
    if (noMilk) {
        buy milk;
    }
}
remove note B;
```

Case 2

- “`if (noNote A)`” happens before “`leave note A`”

```
leave note A;
while (note B) {\\X
    do nothing;
};
```

Wait for note B to be remove

```
if (noMilk) {
    buy milk;}
}
remove note A;
```

happened before

```
leave note B;
if (noNote A) {\\Y
    if (noMilk) {
        buy milk;
    }
}
remove note B;
```

Solution #3 discussion

- Our solution protects a single "Critical-Section" piece of code for each thread:

```
if (noMilk) {
   buy milk;
}
```

- Solution #3 works, but it's really unsatisfactory
  - Really complex – even for this simple an example
    - Hard to convince yourself that this really works
  - A's code is different from B's – what if lots of threads?
    - Code would have to be slightly different for each thread
  - While A is waiting, it is consuming CPU time
    - This is called "busy-waiting"
- There's a better way
  - Have hardware provide higher-level primitives than atomic load & store
  - Build even higher-level programming abstractions on this hardware support

Too Much Milk: Solution #4

- Suppose we have some sort of implementation of a lock
  - `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 milk problem is easy:

```
milklock.Acquire();
if (nomilk)
   buy milk;
milklock.Release();
```

- Once again, section of code between `Acquire()` and `Release()` called a "Critical Section"
- Of course, you can make this even simpler: suppose you are out of ice cream instead of milk
  - Skip the test since you always need more ice cream ;-)

Where are we going with synchronization?

Programs	Shared Programs
Higher-level API	Locks Semaphores Monitors
Hardware	Load/Store Disable Ints Test&Set Compare&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

Critical Section Problem

Definition

- Race condition
  - Several processes access and manipulate the same data concurrently
  - Outcomes of the execution depends on the order in which the access take place

- How to remove Race Condition?
  - Serial execution

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);
```

Requirements to solutions

- **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

Preemption definition

- Preemption
  - The act of temporarily interrupting a task being carried out by a computer system, without requiring its cooperation, and with the intention of resuming the task at a later time [wiki]

Handling critical-section by OS

- Two approaches, depend on type of OS kernels
  - **Preemptive**
    - Allows preemption of process when running in kernel mode
    - Difficult to design in SMP architectures (why?)
  - **Non-preemptive**
    - Runs until exits kernel mode, blocks, or voluntarily yields CPU
      - Essentially free of race conditions in kernel mode (why?)
- Which one
  - is responsive?
  - is suitable for real-time programming?

1) Peterson's solution

- A classis SW solution

- No guarantees in correct working of the method
  - Correctness depends on computer architecture
  - Atomic instructions are needed (which & where?)

- Good algorithm!

- Shared variables
  - `int turn; /* whose turn is */`
  - `Boolean flag[2] /* who enters the critical-section */`

Peterson algorithm for P_i

(Pi, Pj)=(P0, P1)

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

How requirements are satisfied?

- Mutual exclusion (?)
- Progress (?)
- Bounded waiting (?)

the algorithm uses two variables, *flag* and *turn*.

- A *flag[n]* value of *true* indicates that the process *n* wants to enter the critical section.
- *turn* resolves simultaneously conflicts
  - entrance to the critical section is granted for process P0 if P1 does not want to enter its critical section or if P1 has given priority to P0 by setting *turn* to 0.

Proof

- **Mutual Exclusion**
  - assume both P0 and P1 are in their CS
  - then flag[0] = flag[1] = =true
  - the test for entry cannot have been true for both processes at the same time (because turn favors one);
  - therefore one process must have entered its CS first；the other process will be bocked.
- **Progress**
  - **Case I: (Stuck)**
  - P1 is not interested in entering its CS
  - then flag[1] = false; hence the while loop is false for P0 and it can go
  - **Case II: (Deadlock)**
  - P1 is also blocked at the while loop
  - impossible, because turn = 0 or 1;hence the while loop is false for some process and it can go

2) Hardware solution

- Some hardwares support implementing the critical section code!

- All solutions are based on idea of **locking**
  - Protecting critical regions via locks

- Uniprocessors – could disable interrupts
  - Currently running code would execute without preemption
  - Generally too inefficient on multiprocessor systems
    - Operating systems using this not broadly scalable

- Multiprocessors – provide special atomic hardware instructions
    - **Atomic** = non-interruptible
  - either
    - test memory word and set value
    - swap contents of two memory words

Hardware solution for critical section

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

How requirements are satisfied?

- Mutual exclusion (?)
- Progress (?)
- Bounded waiting (?)

test_and_set instruction

Definition:

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

1. Executed atomically

2. Returns the original value of passed parameter

3. Set the new value of passed parameter to “TRUE”.

Hardware solution using *test_and_set()*

- Shared Boolean variable lock, initialized to FALSE

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

    } while (true);
```

compare_and_swap instruction

Definition:

```
   int compare_and_swap(int *value, int expected, int
new_value) {
        int temp = *value;

        if (*value == expected)
           *value = new_value;
    return temp;     /* old value */
   }
```

1. Executed atomically

2. Returns the original value of passed parameter “value”

3. Set the variable “value” the value of the passed parameter “new_value” but only if “value” ==“expected”. That is, the swap takes place only under this condition.

Hardware solution using *compare_and_swap()*

- Shared integer "lock" initialized to 0;

```
do {
    while (compare_and_swap(&lock, 0, 1) != 0)
       ; /* do nothing */

      /* critical section */

     lock = 0;

     /* remainder section */

      } while (true);
```

How requirements are satisfied?

Mutual exclusion (?)

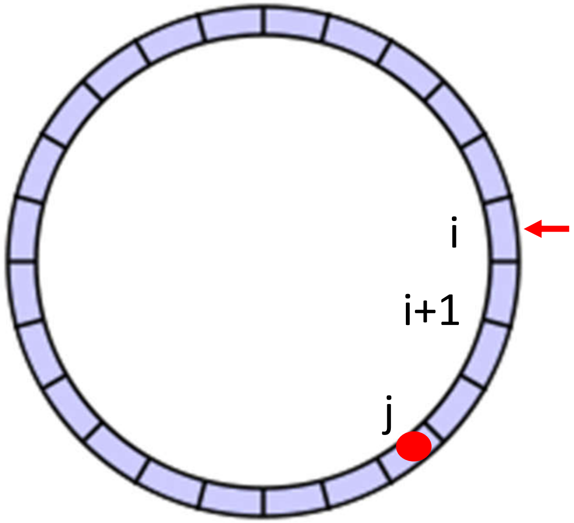
Progress (?)

Bounded waiting (?)

Bounded-waiting mutual exclusion with test_and_set

```
do {
   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;
  /* remainder section */
} while (true);
```

- Key indicates who can enter CS
- Use waiting array indicates who is waiting to enter CS

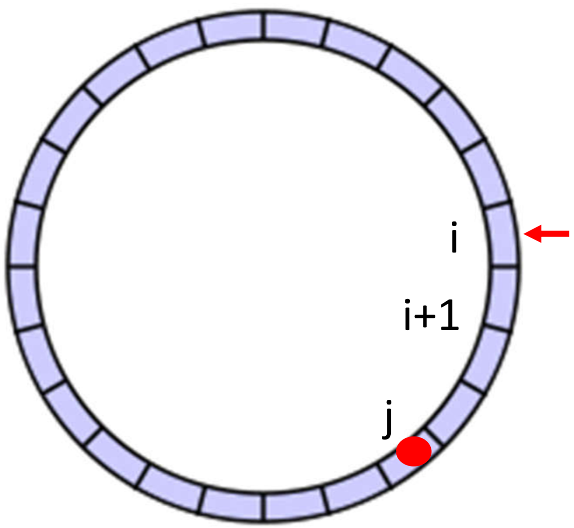


Waiting array

Case :

- P_i can enter it's critical section only if either
  - Waiting[i] == false or key == false

- When Pi leaves it's critical section, it scans the array waiting in the cyclic order
  - i+1, i+2, ……,n-1,0,……, i-1
  - find the first process in waiting array



Waiting array

3) OS solution!: Mutex locks

- Previous solutions are complicated and generally inaccessible to application programmers
- OS designers build software tools to solve critical section problem
- Simplest is *mutex* lock *(mutual exclusions)*
- 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
- But this solution requires **busy waiting**
  - This lock therefore called a **spinlock**

acquire() and release()

- ```
  acquire() {
     while (!available)
         ; /* busy wait */
     available = false;;
  }
  ```

- ```
  release() {
     available = true;
  }
  ```

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

How requirements are satisfied?

Mutual exclusion (?)

Progress (?)

Bounded waiting (?)

Naïve use of Interrupt Enable/Disable

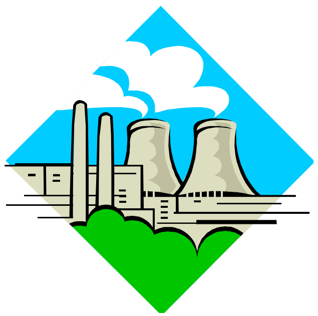
- How can we build multi-instruction atomic operations?
  - Recall: dispatcher gets control in two ways.
    - Internal: Thread does something to relinquish the CPU
    - External: Interrupts cause dispatcher to take CPU
  - On a uniprocessor, can avoid context-switching by:
    - Avoiding internal events (although virtual memory tricky)
    - Preventing external events by disabling interrupts
- Consequently, naïve Implementation of locks:

```
LockAcquire { disable Ints; }
LockRelease { enable Ints; }
```

- Problems with this approach:
  - Can't let user do this! Consider following:

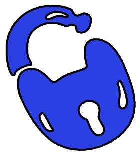
```
LockAcquire();
While(TRUE) {;}
```

  - Real-Time system—no guarantees on timing!
    - Critical Sections might be arbitrarily long



Better Implementation of Locks by Disabling Interrupts

- Key idea: maintain a lock variable and impose mutual exclusion only during operations on that variable



```
int value = FREE;

Acquire() {
   disable interrupts;
   if (value == BUSY) {
      put thread on wait queue;
      Go to sleep();
      // Enable interrupts?
   } else {
      value = BUSY;
   }
   enable interrupts;
}
```

```
Release() {
   disable interrupts;
   if (anyone on wait queue) {
      take thread off wait queue
      Place on ready queue;
   } else {
      value = FREE;
   }
   enable interrupts;
}
```

What is the main problem of all mentioned methods?

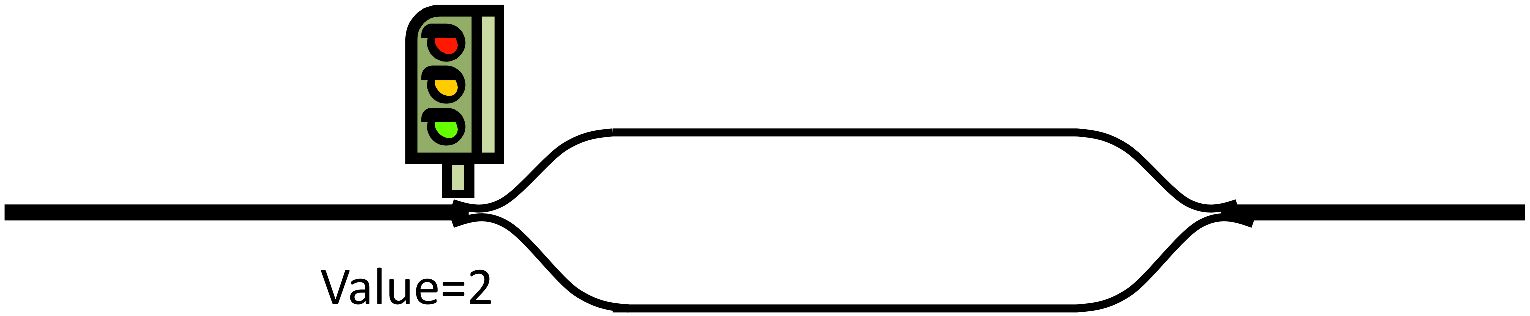
Busy waiting!

4) 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 P and V – can't read or write value, except to set it initially
  - Operations must be atomic
    - Two P's together can't decrement value below zero
    - Similarly, thread going to sleep in P won't miss wakeup from V – even if they both happen at same time
- Semaphore from railway analogy
  - Here is a semaphore initialized to 2 for resource control:



Semaphore(cont.)

- Synchronization tool that provides more sophisticated ways (than Mutex locks) for process to synchronize their activities
  - Semaphore ***S*** – integer variable
  - Can only be accessed via two indivisible (atomic) operations

  `P() (wait())` and **`V() (signal())`**

S. P()

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

S. V()

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

The implementation of semaphore

```
Class Semaphore {
    int sem;
     WaitQueue q;
}
```

```
Semaphore::P() {
    sem--;
    if (sem < 0) {
        Add this thread t to q;
          block(p);
      }
}
```

```
Semaphore::V() {
    sem++;
    if (sem<=0) {
        Remove a thread t from q;
         wakeup(t);
    }
}
```

Note that in this implementation, semaphore values may be negative, whereas semaphore values are never negative under the classical definition of semaphores with busy waiting.

Types of semaphore

- Types
  - Binary semaphore (same as mutex lock)
  - Counting semaphore (suitable for managing number of resources)
- Can solve various synchronization problems
- Example:
  - Consider P_1 and P_2 that require S_1 to happen before S_2

Create a semaphore "**synch**" initialized to zero

```
P1:
    S1;
    signal(synch);
```

```
P2:
    wait(synch);
    S2;
```

Two Uses of Semaphores: Mutual Exclusion

Mutual Exclusion (initial value = 1)

- Also called “Binary Semaphore”.
- Can be used for mutual exclusion:

```
semaphore.P();
    // Critical section goes here
semaphore.V();
```

```
mutex = new Semaphore(1);
```

```
//Thread A:
mutex.P();
 // Critical section
mutex.V();
```

```
//Thread B:
mutex.P();
 // Critical section
mutex.V();
```

Two Uses of Semaphores: synchronization

Scheduling Constraints (initial value = 0)

- Allow thread 1 to wait for a signal from thread 2
  - thread A schedules thread B when a given event occurs

```
condition = new Semaphore(0);
```

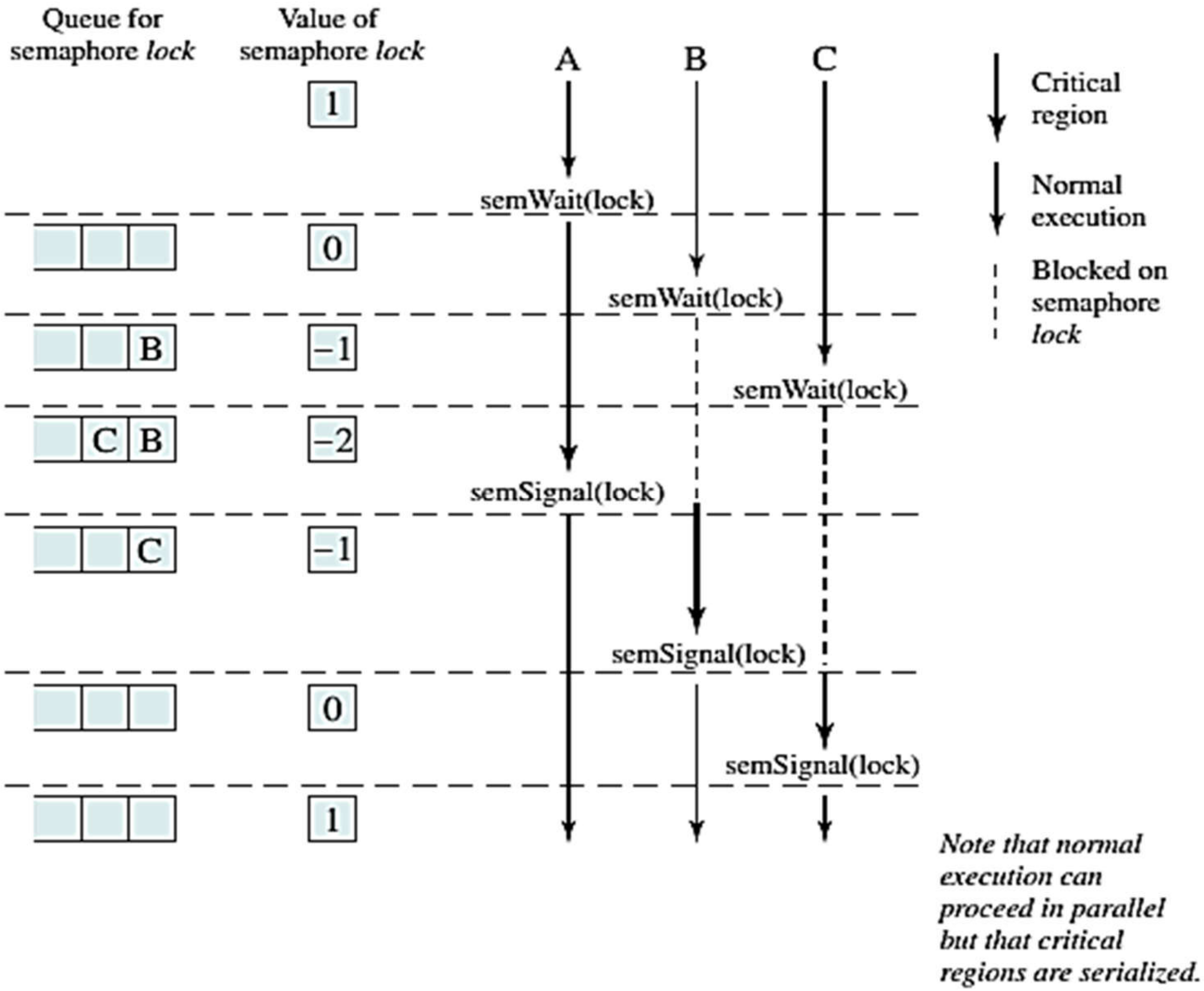
线程A

```
… M …
condition->P();
… N …
```

线程B

```
… X …
condition->V();
… Y …
```

Accessing shared data by Semaphore



Semaphore points

- Must guarantee that no two processes can execute the **`wait()`** and **`signal()`** on the same semaphore at the same time (why?)
  - wait() and signal() must be atomic!
  - wait() and signal() generate a Critical Section Problem!
  - How to solve?
    - Uniprocessors
      - Disabling interrupts
    - SMP (Multiprocessors)
      - Disabling interrupts (bad performance effect)
      - Other methods: compare_and_swap() and spinlock (is it good to have busy waiting?)

Two implementations of semaphores

```
semWait(s)
{
   while (compare_and_swap(s.flag, 0 , 1) == 1)
      /* do nothing */;
   s.count--;
   if (s.count < 0) {
      /* place this process in s.queue*/;
      /* block this process (must also set
s.flag to 0) */;
   }
   s.flag = 0;
}

semSignal(s)
{
   while (compare_and_swap(s.flag, 0 , 1) == 1)
      /* do nothing */;
   s.count++;
   if (s.count<= 0) {
      /* remove a process P from s.queue */;
      /* place process P on ready list */;
   }
   s.flag = 0;
}
```

(a) Compare and Swap Instruction

```
semWait(s)
{
   inhibit interrupts;
   s.count--;
   if (s.count < 0) {
      /* place this process in s.queue */;
      /* block this process and allow inter-
rupts*/;
   }
   else
      allow interrupts;
}

semSignal(s)
{
   inhibit interrupts;
   s.count++;
   if (s.count<= 0) {
      /* remove a process P from s.queue */;
      /* place process P on ready list */;
   }
   allow interrupts;
}
```

(b) Interrupts

Problems with semaphores

- Be careful in the usage
  - Deadlock, Starvation, Priority inversion

P_0

```
wait(S);
wait(Q);
...
signal(S);
signal(Q);
```

P_1

```
wait(Q);
wait(S);
...
signal(Q);
signal(S);
```

- Starvation
  - LIFO queue

- **Priority Inversion** – Scheduling problem when lower-priority process holds a lock needed by higher-priority process
  - Example: L (R) < M < H (R)

➢ Solved via **priority-inheritance protocol**

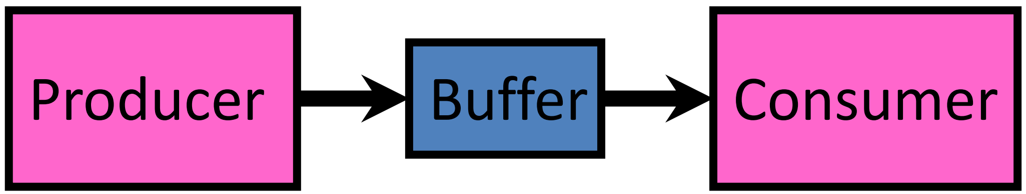
Classic synchronization problems

- The bounded-buffer problem
- The readers-writers problem
- The dining-philosophers problem

How can semaphore solve these problems?

Producer-Consumer with a Bounded Buffer

- Problem Definition
  - Producer puts things into a shared buffer
  - Consumer takes them out
  - Need synchronization to coordinate producer/consumer
- Don't want producer and consumer to have to work in lockstep, so put a fixed-size buffer between them
  - Need to synchronize access to this buffer
  - Producer needs to wait if buffer is full
  - Consumer needs to wait if buffer is empty



Correctness constraints for solution

- Correctness Constraints:
  - Consumer must wait for producer to fill buffers, if none full (scheduling constraint)
  - Producer must wait for consumer to empty buffers, if all full (scheduling constraint)
  - Only one thread can manipulate buffer queue at a time (mutual exclusion)
- Remember why we need mutual exclusion
  - Because computers are stupid
- General rule of thumb:
  Use a separate semaphore for each constraint
  - `Semaphore fullBuffer; // consumer's constraint`
  - `Semaphore emptyBuffer;// producer's constraint`
  - `Semaphore mutex;       // mutual exclusion`

Full Solution to Bounded Buffer

```
Semaphore mutex = 1;
Semaphore fullBuffer = 0;   //empty
Semaphore emptyBuffer = bufSize;
```

```
Producer(item){
    emptyBuffer.P();
    mutex.P();
    Add item to the buffer;
    mutex.V();
    fullBuffer.V();
}
```

```
Consumer(item) {
    fullBuffers.P();
    mutex.P();
    Remove item from buffer;
    mutex.V();
    emptyBuffers.V();
}
```

Does it matter about the order of P()、V();

Discussion about Solution

Decrease # of empty buffer

Increase # of occupied buffer

- Why asymmetry?
  - Producer does: `emptyBuffer.P(), fullBuffer.V()`
  - Consumer does: `fullBuffer.P(), emptyBuffer.V()`

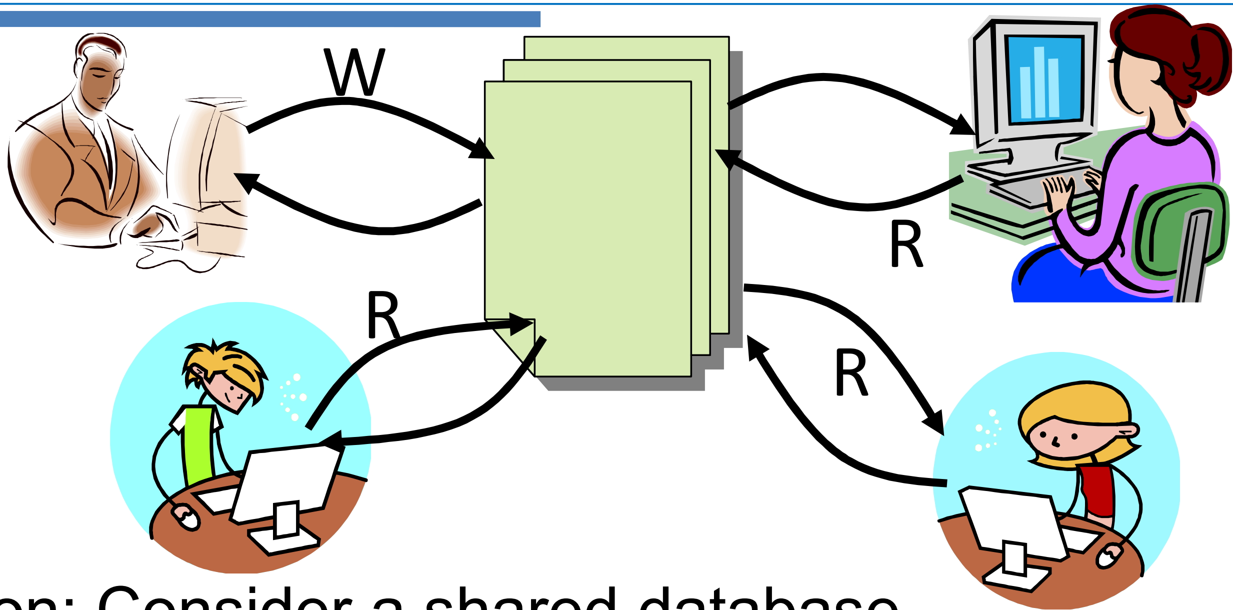
Decrease # of occupied buffer

Increase # of empty buffer

- Is order of P's important?
- Is order of V's important?
  - No, except that it might affect scheduling efficiency
- What if we have 2 producers or 2 consumers?
  - Do we need to change anything?

```
Producer(item) {
   mutex.P();
   emptyBuffer.P();
   Enqueue(item);
   mutex.V();
   fullBuffer.V();
}
Consumer() {
   fullBuffer.P();
   mutex.P();
   item = Dequeue();
   mutex.V();
   emptyBuffer.V();
   return item;
}
```

Readers/Writers Problem

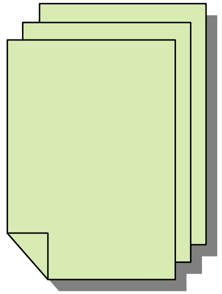


- Motivation: Consider a shared database
  - Two classes of users:
    - Readers – never modify database
    - Writers – read and modify database
  - Is using a single lock on the whole database sufficient?
    - Like to have many readers at the same time
    - Only one writer at a time

Basic Readers/Writers Solution

- Correctness Constraints:
  - Readers can access database when no writers
  - Writers can access database when no readers or writers
  - Only one thread manipulates state variables at a time
- Basic structure of a solution:
  - **`Reader()`**

```
Wait until no writers
Access data base
Check out – wake up a waiting writer
```



  - **`Writer()`**

```
Wait until no active readers or writers
Access database
Check out – wake up waiting readers or writer
```

  - Semaphore and variable:

```
semaphore rw_mutex = 1;   //read and write semaphore
semaphore mutex = 1;      // mutual exclusion for read_count
int read_count = 0;       // currently reading
```

Solution 1: using semaphore

Writer

Reader

```
write;
```

```
read;
```

Solution 1: using semaphore(cont.)

Writer

```
P(rw_mutex);
  write;
V(rw_mutex);
```

Reader

```
P(rw_mutex);

read;

V(rw_mutex);
```

Solution 1: using semaphore(cont.)

Writer

```
P(rw_mutex);

  write;

V(rw_mutex);
```

Reader

```
  if (read_count == 0)
     P(rw_mutex);
    ++read_count;


read;




    V(rw_mutex);
```

Solution 1: using semaphore(cont.)

Writer

```
P(rw_mutex);

  write;

V(rw_mutex);
```

Reader

```
  if (read_count == 0)
    P(rw_mutex);
   ++read_count;


read;


  --read_count;
  if (read_count == 0)
   V(rw_mutex);
```

Solution 1: using semaphore(cont.)

Writer

```
P(rw_mutex);

  write;

V(rw_Mutex);
```

Reader

```
P(mutex);
  if (read_count == 0)
     P(rw_mutex);
     ++read_count;
V(mutex);

read;


  --read_count;
  if (read_count == 0)
     V(rw_utex);
```

Solution 1: using semaphore(cont.)

Writer

```
P(rw_mutex);

  write;

V(rw_mutex);
```

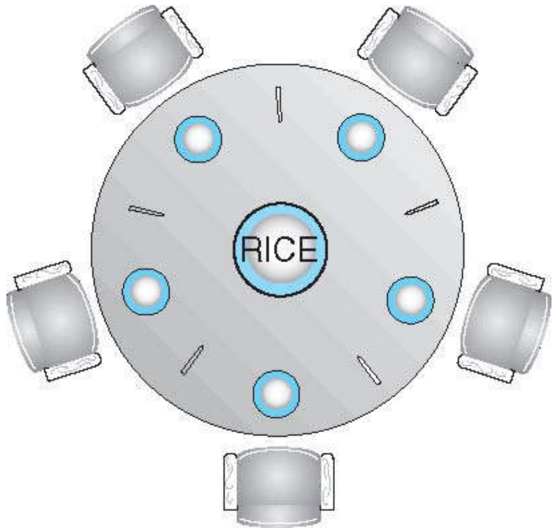
Reader

```
P(mutex);
  if (read_count == 0)
     P(rw_mutex);
    ++read_count;
V(mutex);

read;

P(mutex);
  --read_count;
  if (read_count == 0)
    V(rw_mutex);
V(mutex)
```

Reader first

Dining Philosophers Problem

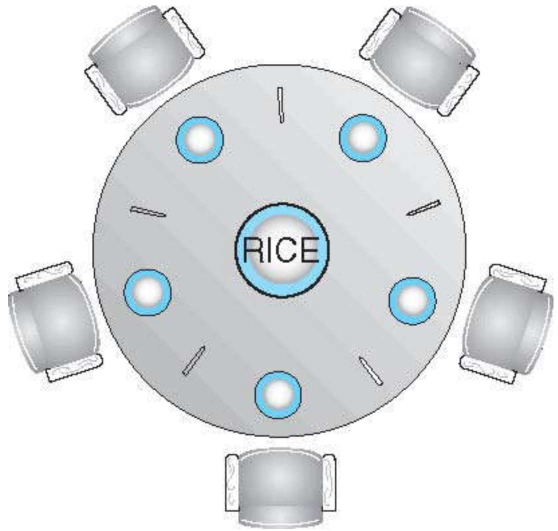
- *Five* philosophers seated around a circular table
  - ➢ There is one chopstick between each philosopher
  - ➢ A philosopher must pick up its two nearest chopsticks in order to eat
  - ➢ A philosopher must pick up first one chopstick, then the second one, not both at once



- Devise an algorithm for allocating these limited resources (chopsticks) among several processes (philosophers) in a manner that is
  - – deadlock-free, and
  - – starvation-free

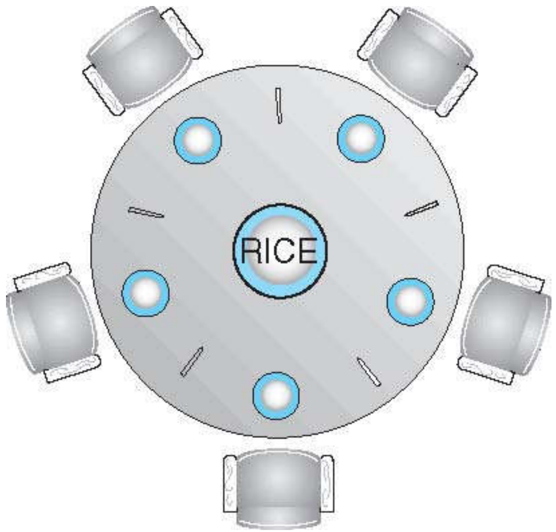
The dining-philosophers problem

- Five philosophers are in a thinking - eating cycle.
- When a philosopher gets hungry, he sits down, picks up two nearest chopsticks, and eats.
- A philosopher can eat only if he has both chopsticks.
- After eating, he puts down both chopsticks and thinks.
- This cycle continues.



The dining-philosophers problem(cont.)

- Chopsticks are shared items (by two philosophers) and must be protected.
- Each chopstick has a semaphore with initial value 1.
- A philosopher calls wait() before picks up a chopstick and calls signal() to release it.



Solution 1:

Thinking and eating alternatively

```
#define N 5                                  // number of philosopher
semaphore chopstick[N];                      // semaphore initialize 1
 void   philosopher(int   i)                 // philosopher id: 0 - 4
      while(TRUE)
      {
        think( );                            // thinking
        P(chopstick[i]);                     // get left chopstick
        P(chopstick[(i + 1) % N]);           //get right chopstick
        eat( );                              // eating....
        V(chopstick[i]);                     // put left chopstick
        V(chopstick[(i + 1) % N ]);          // put right chopstick
      }
```

Any problem?

starvation

Solution 2:

```
#define   N   5                  // number of philosopher
semaphore chopstick [5];    // semaphore initialize 1
semaphore   mutex;               //  mutex,  initial value 1
```

Solution 2:

```
#define   N   5                    // number of philosopher
semaphore chopstick [5];           // semaphore initialize 1
semaphore   mutex;                  //  mutex,  initial value 1
void   philosopher(int   i)        // philosopher Id:  0  -  4
     while(TRUE){
          think( );                                    // thinking



          eat( );                                   // eating ….



     }
```

Solution 2:

```
#define   N   5                    // number of philosopher
semaphore chopstick [5];           // semaphore initialize 1
semaphore   mutex;                  //  mutex, initial value 1
void   philosopher(int   i)        // philosopher Id: 0 - 4
     while(TRUE){
          think( );                              // thinking
          P(mutex);                             // enter CS

          eat( );                              // eating …


          V(mutex);                            // leaving CS



     }
```

Solution 2:

```
#define   N   5                    // number of philosopher
semaphore chopstick [5];           // semaphore initialize 1
semaphore   mutex;                 //  mutex, initial value 1
void   philosopher(int   i)        // philosopher Id: 0 - 4
    while(TRUE){
        think( );                          // thinking
        P(mutex);                          // enter CS

        P(chopstick[i]);                   // get left chopstick
        P(chopstick[(i + 1) % N]);         //get right chopstick

        eat( );                            // eating ….


        V(mutex);                          // leaving CS

    }
```

Solution 2:

```
#define   N   5                  // number of philosopher
semaphore chopstick [5];         // semaphore initialize 1
semaphore   mutex;               // mutex, initial value 1
void   philosopher(int   i)      // philosopher Id: 0 - 4
    while(TRUE){
        think( );                          // thinking
        P(mutex);                          // enter CS

        P(chopstick[i]);                   // get left chopstick
        P(chopstick[(i + 1) % N]);         //get right chopstick

        eat( );                            // eating ….

        V(chopstick[i]);           // put left chopstick
        V(chopstick[(i + 1) % N ]); // put right chopstick
        V(mutex);                          // leaving CS
    }
```

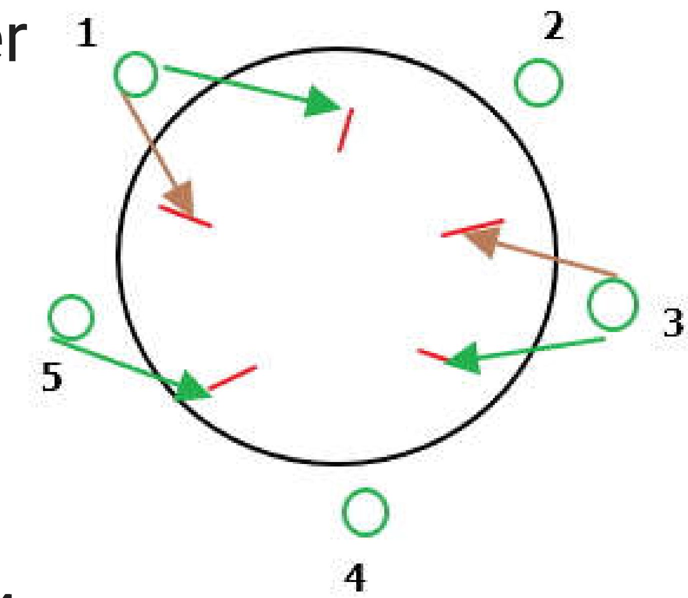
Any problem? It's correct! But just one philosopher can eat!

Dining Philosophers Problem

- Some deadlock-free solutions:
  - allow at most 4 philosophers at the same table when there are 5 resources
  - odd philosophers pick first left then right, while even philosophers pick first right then left
  - allow a philosopher to pick up chopsticks only if both are free. This requires protection of critical sections to test if both chopsticks are free before grabbing them.
    - ✓ we'll see this solution next using monitors
- A deadlock-free solution is not necessarily starvation-free
  - – for now, we'll focus on breaking deadlock

Solution 3：

- **asymmetric solution**
- breaking the circular wait
- **an** odd philosopher pick up first her left chopstick and then picks up her right chopstick ，
- **an** even philosopher picks up her right chopstick and then her left chopstick
- when 1,3,5 got left chopstick , 2,4 cannot get their right chopstick
- Now, 1,3 got their right chopstick too and got executed. After that 2,4 can participate.



Solution 3:

```
#define    N    5                    // number of philosopher
semaphore chopstick [5];             // semaphore initialize 1
semaphore    mutex;                  // mutex, initial value 1
void    philosopher(int    i)        // philosopher Id: 0 - 4
    while(TRUE){
        think( );                              // thinking

       if (i%2 == 0) {

           P(chopstick[i]);                    // get left chopstick
           P(chopstick[(i + 1) % N]);          //get right chopstick
       else{
            P(chopstick[i+1]%N);           // get right chopstick
            P(chopstick[i % N]);           //get left chopstick
        }

        eat( );                                // eating ….

        V(chopstick[i]);                       // put left chopstick
        V(chopstick[(i + 1) % N ]);            // put right chopstick
  }
```

Other problems with semaphore

- Problems with bad usage

```
signal(mutex);
    ...
    critical section
    ...
wait(mutex);
```

```
wait(mutex);
    ...
    critical section
    ...
wait(mutex);
```

```
    ...
    critical section
    ...
wait(mutex);
```

```
wait(mutex);
    ...
    critical section
    ...
```

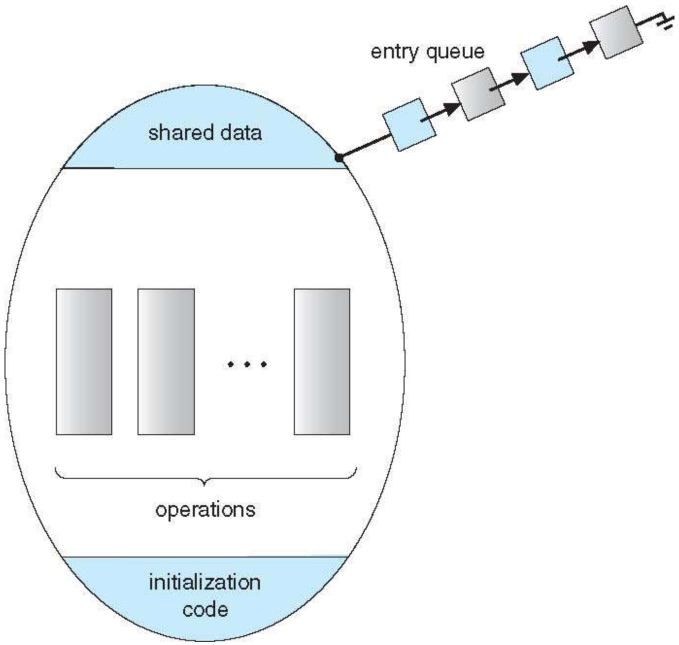
- Deadlock and starvation are possible.

5) Monitor

- A high-level abstraction that provides a convenient and effective mechanism for process synchronization

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

```
monitor monitor-name
{
      // shared variable
     declarations

      procedure P1  (…)  {  ….  }

      procedure Pn  (…)  {……}

    Initialization_Code  (…)  {  …  }

}
```



Monitors and Condition Variables

- Example:

```
monitor sharedcounter {
    int counter;
    function add() { counter++;}
    function sub() { counter--;}
    init() { counter=0; }
}
```

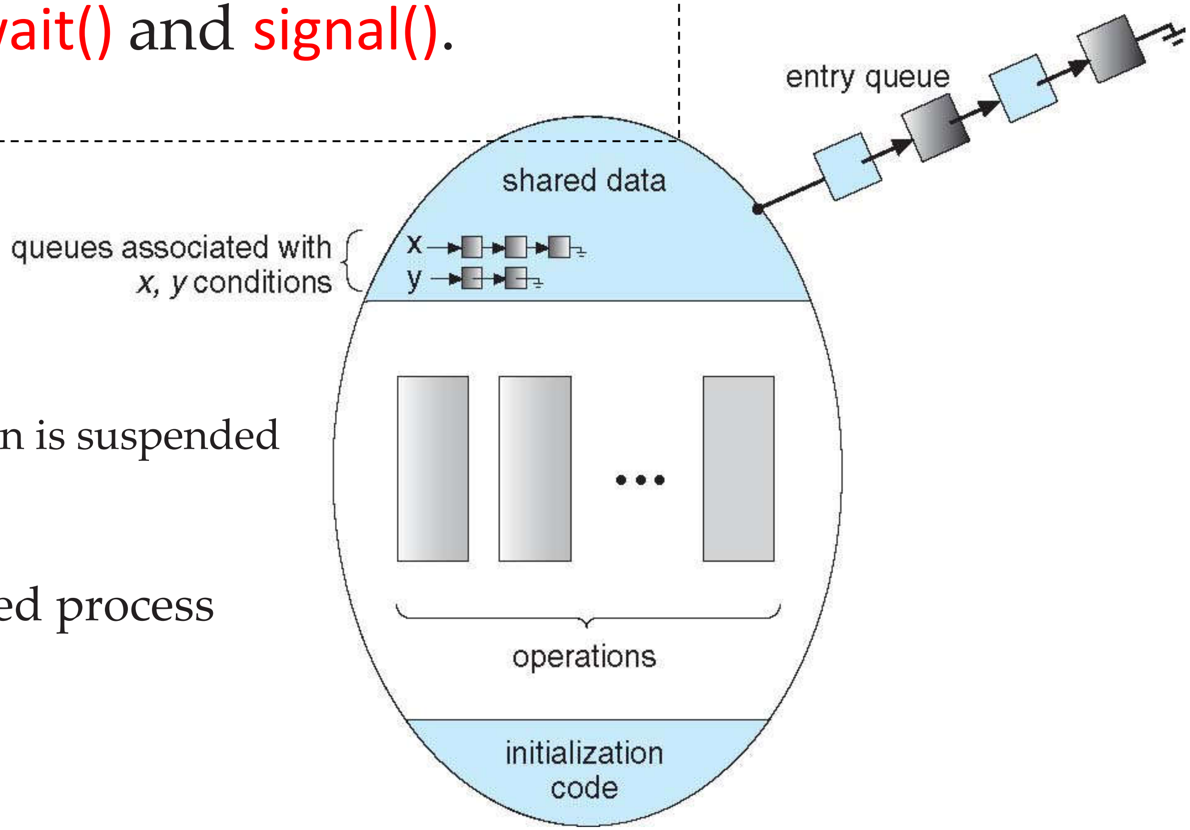
- If two processes want to access this *sharedcounter* monitor, then access is mutually exclusive and only one process at a time can modify the value of counter
  - if a write process calls sharedcounter.add(), then it has exclusive access to modifying counter until it leaves add(). No other process, e.g. a read process, can come in and call sharedcounter.sub() to decrement counter while the write process is still in the monitor

Monitors and Condition Variables

- In the previous *sharedcounter* example, a writer process may be interacting with a reader process via a bounded buffer
  - like the solution to the bounded buffer producer/consumer problem, the writer should signal blocked reader processes when there are no longer zero elements in the buffer
  - monitors alone don't provide this signalling synchronization capability
- In general, there may be times when one process wishes to signal another process based on a condition, much like semaphores.
  - Thus, monitors alone are insufficient.
  - Augment monitors with *condition variables*.

Monitor (with *condition variables*)

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



condition x, y;

// the process invoking this operation is suspended
x.wait();

// resumes exactly one suspended process
x.signal();

Monitors and Condition Variables

Semantics concerning what happens just after x.signal() is called by a process P in order to wake up a process Q waiting on this CV x

- Hansen(singal-and-wait)
  - Applied in OS and Java

```
l.acquire()
…
x.wait()
```

T1 waiting

T2 enter monitor

```
l.acquire()
…
x.signal()
…
l.release()
```

T2 leave monitor

T1 resume

```
…
l.release()
```

- Hoare(signal-and-continue)
  - text book

```
l.acquire()
…
x.wait()
```

T1 waiting

T2 enter monitor

```
l.acquire()
…
x.signal()
```

T2 waiting

T1 resume

```
…
l.release()
```

T1 leave monitor

T2 resume

```
…
l.release()
```

Monitor-based Solution to Dining Philosophers

- Key insight: pick up 2 chopsticks only if both are free
  - this avoids deadlock
  - a philosopher moves to his/her eating state only if both neighbors are not in their eating states
    - thus, need to define a state for each philosopher
  - if one of my neighbors is eating, and I'm hungry, ask them to signal() me when they're done
    - thus, states of each philosopher are: thinking, hungry, eating
    - thus, need condition variables to signal() waiting hungry philosopher(s)
  - Also, need to Pickup() and Putdown() chopsticks

Monitor-based Solution to Dining Philosophers

- Some basic pseudo-code for monitor

```
Monitor
  DiningPhilosophers {
    // THINKING; HUNGRY, EATING
    status state[5];
    condition self[5];
    Pickup(int i);
    Putdown(int i);
  }
```

- Each philosopher *i* runs pseudo-code:

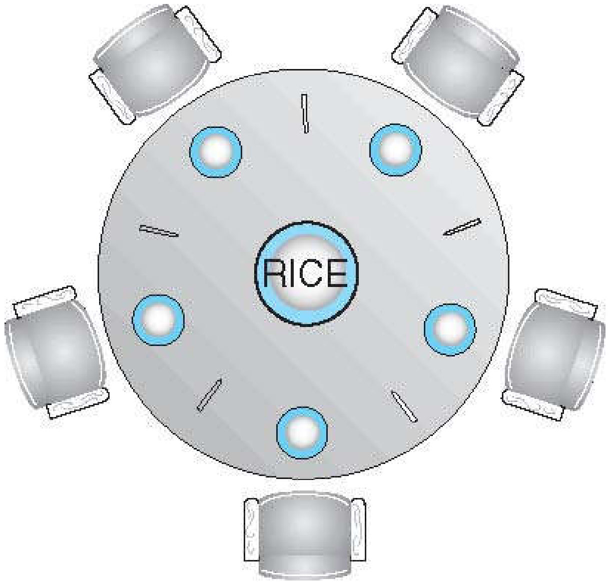
```
DP.Pickup(i);
...
DP.Putdown(i);
```

The dining-philosophers problem

```
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);
    }
```

```
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;
}

}
```

The dining-philosophers problem




```
procedure philosopher(i)
 {
   while TRUE do
    {
      THINKING;
      DiningPhilosophers.pickup(i);
      EATING;
      DiningPhilosophers.putdown(i);

    }
 }
```

Any problem?

No deadlock

semaphores in C language

Semaphores in C

- the POSIX system in Linux presents its own built-in semaphore library. To use it, we have to :
  1. Include semaphore.h
  2. Compile the code by linking with -lpthread -lrt

To declare a semaphores

```
sem_t sem;
```

To lock a semaphore or wait we can use the sem_wait function:

```
int sem_wait(sem_t *sem);
```

To release or signal a semaphore, we use the **sem_post** function:

```
int sem_post(sem_t *sem);
```

A semaphore is initialised by using **sem_init**(for processes or threads) or **sem_open** (for IPC).

```
sem_init(sem_t *sem, int pshared, unsigned int value);
```

To destroy a semaphore

```
sem_destoy(sem_t *mutex);
```

EX:1

```c
// C program to demonstrate working of Semaphores
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>

sem_t mutex;

void* thread(void* arg)
{
    //wait
    sem_wait(&mutex);
    printf("\nEntered..\n");

    //critical section
    sleep(4);

    //signal
    printf("\nJust Exiting...\n");
    sem_post(&mutex);
}


int main()
{
    sem_init(&mutex, 0, 1);
    pthread_t t1,t2;
    pthread_create(&t1,NULL,thread,NULL);
    sleep(2);
    pthread_create(&t2,NULL,thread,NULL);
    pthread_join(t1,NULL);
    pthread_join(t2,NULL);
    sem_destroy(&mutex);
    return 0;
}
```

gcc a.c -lpthread -lrt

Ex 2:

```c
#include <pthread.h>
#include <semaphore.h>
#include <stdio.h>
#include <stdlib.h>

#define NITER 1000000

int cnt = 0;
sem_t mutex;

void * Count(void * a)
{
    int i, tmp;
    for(i = 0; i < NITER; i++)
    {
        sem_wait(&mutex);
        tmp = cnt;      /* copy the global cnt locally */
        tmp = tmp+1;    /* increment the local copy */
        cnt = tmp;      /* store the local value into the
        sem_post(&mutex);
    }
}

int main(int argc, char * argv[])
{
    pthread_t tid1, tid2;
    sem_init(&mutex, 0,1);    // mutex
```

```c
int main(int argc, char * argv[])
{
    pthread_t tid1, tid2;
    sem_init(&mutex, 0,1);    // mutex

    if(pthread_create(&tid1, NULL, Count, NULL))
    {
      printf("\n ERROR creating thread 1");
      exit(1);
    }

    if(pthread_create(&tid2, NULL, Count, NULL))
    {
      printf("\n ERROR creating thread 2");
      exit(1);
    }

    if(pthread_join(tid1, NULL))    /* wait for th
    {
      printf("\n ERROR joining thread");
      exit(1);
    }

    if(pthread_join(tid2, NULL))        /* wait fo
    {
      printf("\n ERROR joining thread");
      exit(1);
    }
```

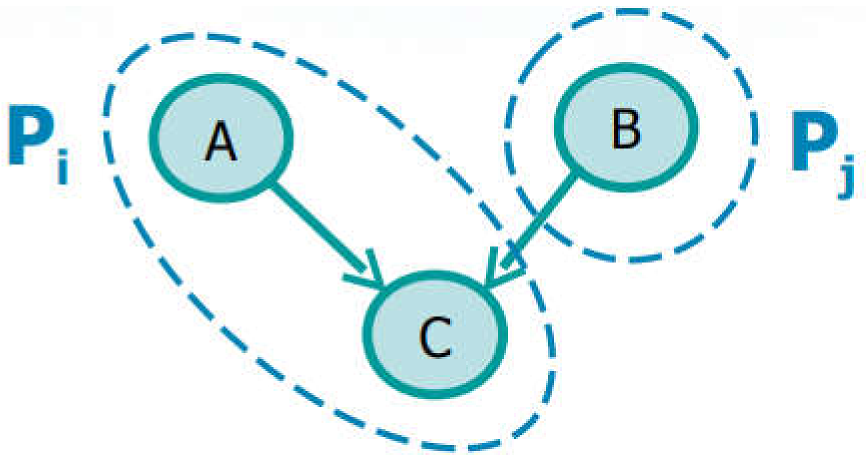
Points to monitor

- Monitors can be implemented by semaphores
- OSes support
  - Monitor, semaphore, spinlock, mutex
  - Examples
    - Solaris
    - Windows
    - Linux
    - Pthreads
- Alternative approaches
  - Transactional Memory
  - OpenMP
  - Functional Programming Languages

```
void update(int value)
{
        #pragma omp critical
        {
          count += value
        }
}
```

Semaphore use: Exercise 1

- Obtain the following precedence graph



init (S, 0);

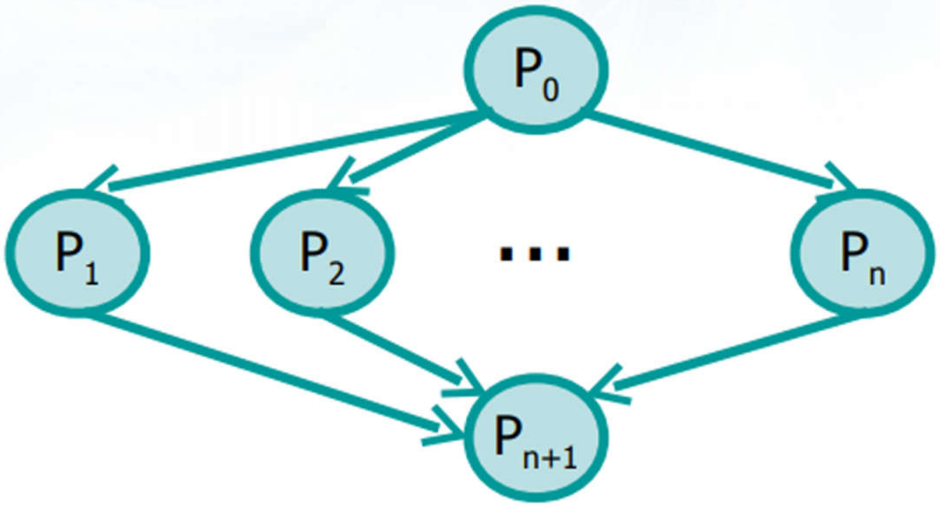
Process i

A
wait (S);
C

Process j

B
signal (S);

Semaphore use: Exercise 2




```
init (S1, 0);
init (S2, 0);
```

Process 0

```
...
i=1
while (i<=n) {
  signal (S1);
  i++;
}
...
```

Process i

```
wait (S1);
...
signal (S2);
...
```

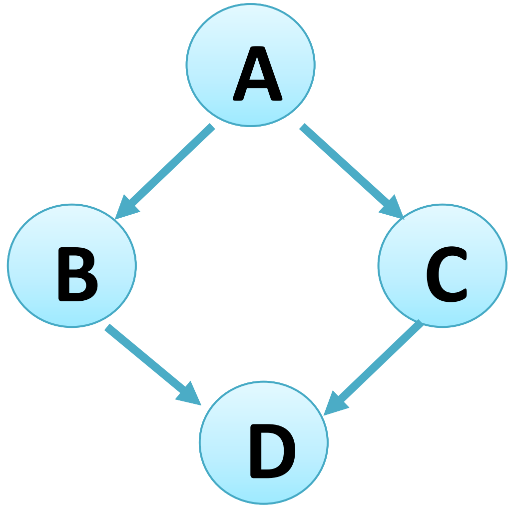
Process n+1

```
...
i=1;
while (i<=n) {
  wait (S2);
  i++;
}
...
```

Semaphore use: Exercise 3

- Realize the represented precedence graph using semaphores

```
init (S1, 0);
init (S2, 0);
```



B

```
...
wait (S1);
B
signal (S2);
...
```

A

```
A
signal (S1);
signal (S1);
...
```

C

```
...
wait (S1);
B
signal (S2);
...
```

D

```
...
wait (S2);
wait (S2);
D
```

Summary (1/2)

- Important concept: Atomic Operations
  - An operation that runs to completion or not at all
  - These are the primitives on which to construct various synchronization primitives
- Talked about hardware atomicity primitives:
  - Disabling of Interrupts, test&set, swap, compare&swap, load-locked & store-conditional
- Showed several constructions of Locks
  - Must be very careful not to waste/tie up machine resources
    - Shouldn't disable interrupts for long
    - Shouldn't spin wait for long
  - Key idea: Separate lock variable, use hardware mechanisms to protect modifications of that variable

Summary (2/2)

- Semaphores: Like integers with restricted interface
  - Two operations:
    - `P()`: Wait if zero; decrement when becomes non-zero
    - `V()`: Increment and wake a sleeping task (if exists)
    - Can initialize value to any non-negative value
  - Use separate semaphore for each constraint
- Monitors: A lock plus one or more condition variables
  - Always acquire lock before accessing shared data
  - Use condition variables to wait inside critical section
    - Three Operations: `Wait()`, `Signal()`

Questions?