CS370 Operating Systems

Colorado State University
Yashwant K Malaiya

Fall 2025 L10

Synchronization

Slides based on

- Text by Silberschatz, Galvin, Gagne
- Various sources

Process Synchronization: Outline

- Critical-section problem to ensure the consistency of shared data
- Software and hardware solutions of the critical-section problem
  - Peterson's solution
  - Atomic instructions
  - Mutex locks and semaphores
- Classical process-synchronization problems
  - Bounded buffer, Readers Writers, Dining Philosophers
- Another approach: Monitors

Process Synchronization

EW Dijkstra *Go To Statement Considered Harmful*

Colorado State University

Process Synchronization

Overview

- We synchronization is needed
- Critical section: access controlled to permit just one process
  - How the critical section be implemented
  - Mutex locks and semaphores
- Classic synchronization problems
- Will a solution cause a deadlock?

Too Much Milk Example

	Person A	Person B
12:30	Look in fridge. Out of milk.	
12:35	Leave for store.	Look in fridge. Out of milk.
12:40	Arrive at store.	Leave for store
12:45	Buy milk.	Arrive at store.
12:50	Arrive home, put milk away.	Buy milk
12:55		Arrive home, put milk away. Oh no!

Background

- Processes can execute concurrently
  - May be interrupted at any time, partially completing execution
- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
- **Illustration**: we wanted to provide a solution to the consumer-producer problem that fills ***all*** the buffers.
  - 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
  - decremented by the consumer after it consumes a buffer.

  Will it work without any problems?

Consumer-producer problem

Producer

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

Consumer

```
while (true) {
  while (counter == 0);
        /* do nothing */
   next_consumed = buffer[out];
   out = (out + 1) % BUFFER_SIZ
   counter--;
        /* consume the item in
        next consumed */
}
```

They run "concurrently" (or in parallel), and are subject to context switches at unpredictable times.

In, out: indices of empty and filled items in the buffer.

Race Condition

`counter++` **could be compiled as**

```
register1 = counter
register1 = register1 + 1
counter = register1
```

`counter--` **could be compiled as**

```
register2 = counter
register2 = register2 - 1
counter = register2
```

They run concurrently, and are subject to context switches at unpredictable times.

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}

Overwrites!

Critical Section Problem

We saw race condition between counter ++ and counter –

Solution to the "*race condition*" problem: critical section

- 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 follows.**

Race condition: when outcome depends on timing/order that is not predictable

Process Synchronization: Outline

- Critical-section problem to ensure the consistency of shared data
- Software and hardware solutions of the critical-section problem
  - Peterson's solution
  - Atomic instructions
  - Mutex locks and semaphores
- Classical process-synchronization problems
  - Bounded buffer, Readers Writers, Dining Philosophers
- Another approach: Monitors

General structure: Critical section

```
do {

    entry section
    critical section
    exit section

     remainder section

 } while (true);
```

Request permission to enter

Housekeeping to let other processes to enter

A process is prohibited from entering the critical section while another process is in it.
Multiple processes are trying to enter the critical section concurrently by executing the same code.

Solution to Critical-Section Problem

A good solution to the critical-section problem should have these attributes

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

Peterson's Solution

- Good algorithmic description of solving the problem
- Two process solution only
- Assume that the **`load`** and **`store`** machine-language instructions are atomic; that is, cannot be interrupted
- The two processes share two variables:
  - **`int turn;`**
  - **`Boolean flag[2]`**
  - The variable **`turn`** indicates whose turn it is to enter the critical section
  - The **`flag`** array is used to indicate if a process is ready to enter the critical section. **`flag[i] = `*`true`*** implies that process $\mathbf{P_i}$ is ready to enter!

Algorithm for Process P_i

For process Pi,
Pj runs the same code concurrently

Being nice!

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

- The variable **turn** indicates whose turn it is to enter the critical section
- The **flag** array is used to indicate if a process is ready to enter the critical section. **flag[i] = *true*** implies that process P_i is ready!
- Note: Entry section- Critical section-Exist section
- These algorithms assume 2 or more processes are trying to get in the critical section.

Colorado State University

Peterson’s Solution (Cont.)

Provable that the three CS requirement are met:

1. Mutual exclusion is preserved

   P_i enters CS only if:

   either **flag[j] = false** or **turn = i**

2. Progress requirement is satisfied

   If a process wants to enter, it only has to wait until the other finishes.

3. Bounded-waiting requirement is met.

   A process waits only one turn.

Detailed proof in the text.

Note: there exists a generalization of Peterson’s solution for more than 2 processes, but bounded waiting is not assured. May not work in multiple processor systems, turn may be modified by by both processors.

Synchronization: Hardware Support

- Most modern processors provide hardware support (*ISA*) for implementing the critical section code. FAQ
- All solutions below based on idea of **locking**
  - Protecting critical regions via locks
- Modern machines provide special atomic hardware instructions (binary machine instructions, not high-level like C)
    - **Atomic** = non-interruptible
  - test memory word and set value
  - swap contents of two memory words
  - others

Colorado State University

Solution 1: using test_and_set()

Lock TRUE: locked, Lock FALSE: not locked. Lock is a shared variable.
test_and_set(&lock) returns the lock value and then sets it to True .

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

```
do {
      while (test_and_set(&lock)) ; /* do nothing */

              /* critical section */
        …. . .
      lock = false;
              /* remainder section */
    …   ..
  } while (true);
```

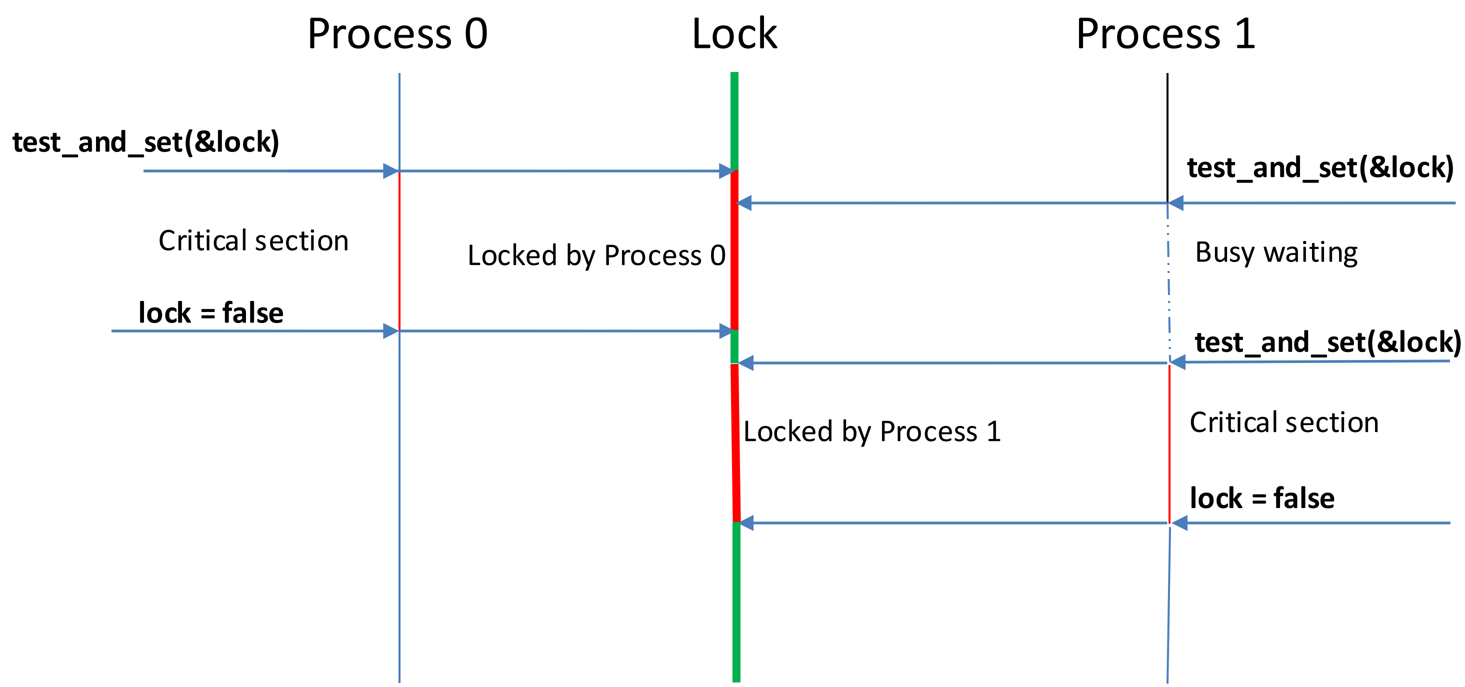
To break out:
Return value of TestAndSet should be FALSE

If two TestAndSet() are attempted *simultaneously*, they will be executed *sequentially* in some arbitrary order

Colorado State University

test_and_set(&lock)

Shared variable lock is initially FALSE




```
while (test_and_set(&lock)) ; /* do nothing */

                /* critical section */
             …..
         lock = false;
                /* remainder section */
```


Colorado State University

Solution 2: Swap: Hardware implementation

Another way of sensing/setting the lock (next slide).

Background: Remember this C code?

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

Using Swap (concurrently executed by both)

```
do {
    key = TRUE;
    while (key == TRUE) {
        Swap(&lock, &key)
        }

    critical section

    lock = FALSE;

    remainder section
} while (TRUE);
```

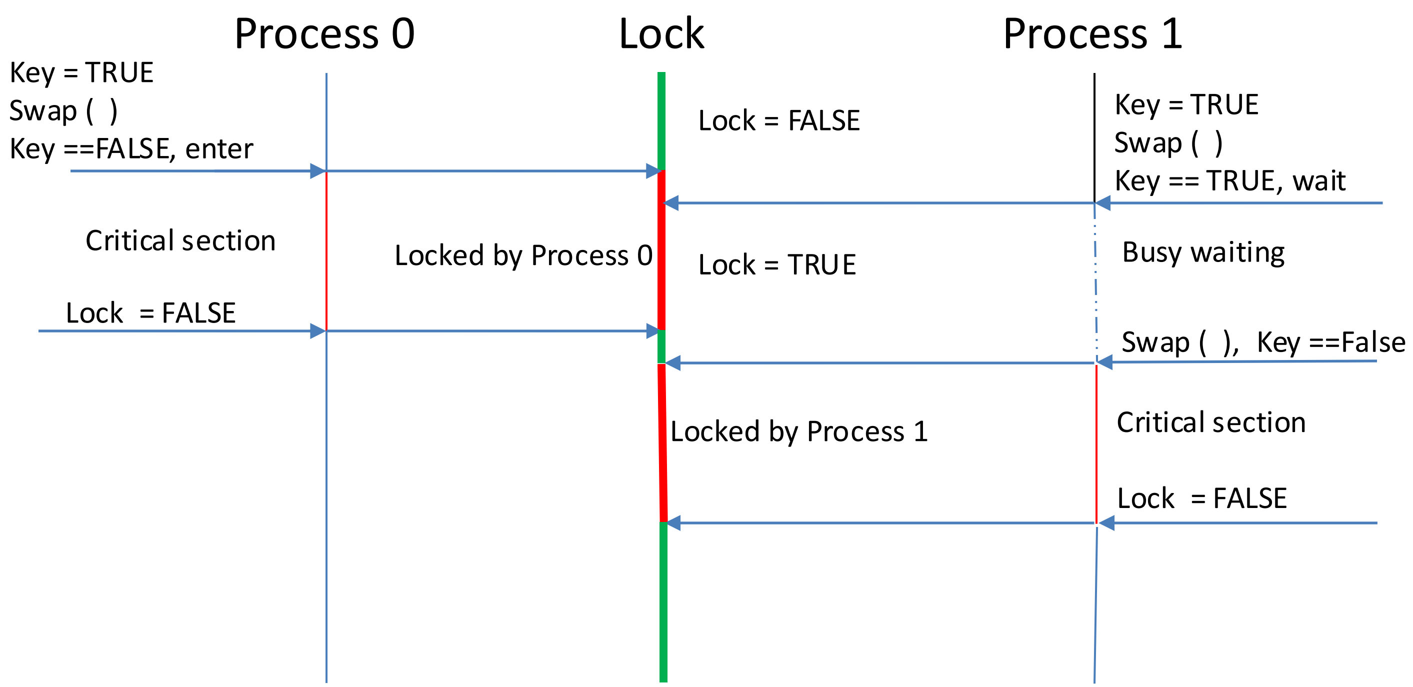
Lock is a SHARED variable.
Key is a variable local to the process.

Lock == false when no process is in critical section.

Cannot enter critical section UNLESS lock == FALSE *by other process or initially*

If two Swap() are executed simultaneously, they will be executed sequentially in some arbitrary order

Swap()



Note: I created this to visualize the mechanism. It is not in the book. - Yashwant

Bounded-waiting Mutual Exclusion with test_and_set

For process i:

```
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);
```

Shared Data structures initialized to FALSE

- `boolean waiting[n];` Pr n wants to enter
- `boolean lock;`

The entry section for process i :

- First process to execute TestAndSet will find key == false ; ENTER critical section,
- EVERYONE else must wait

The exit section for process i:

Attempts to finding a suitable waiting process j (while loop) and enable it,

or if there is no suitable process, make lock FALSE.

Bounded-waiting Mutual Exclusion with test_and_set

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 i exits the CS, it either sets lock to false, or waiting[i] to false (allowing j to get in) , 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

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 (boolean **mu**tual **ex**clusion)
- 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()

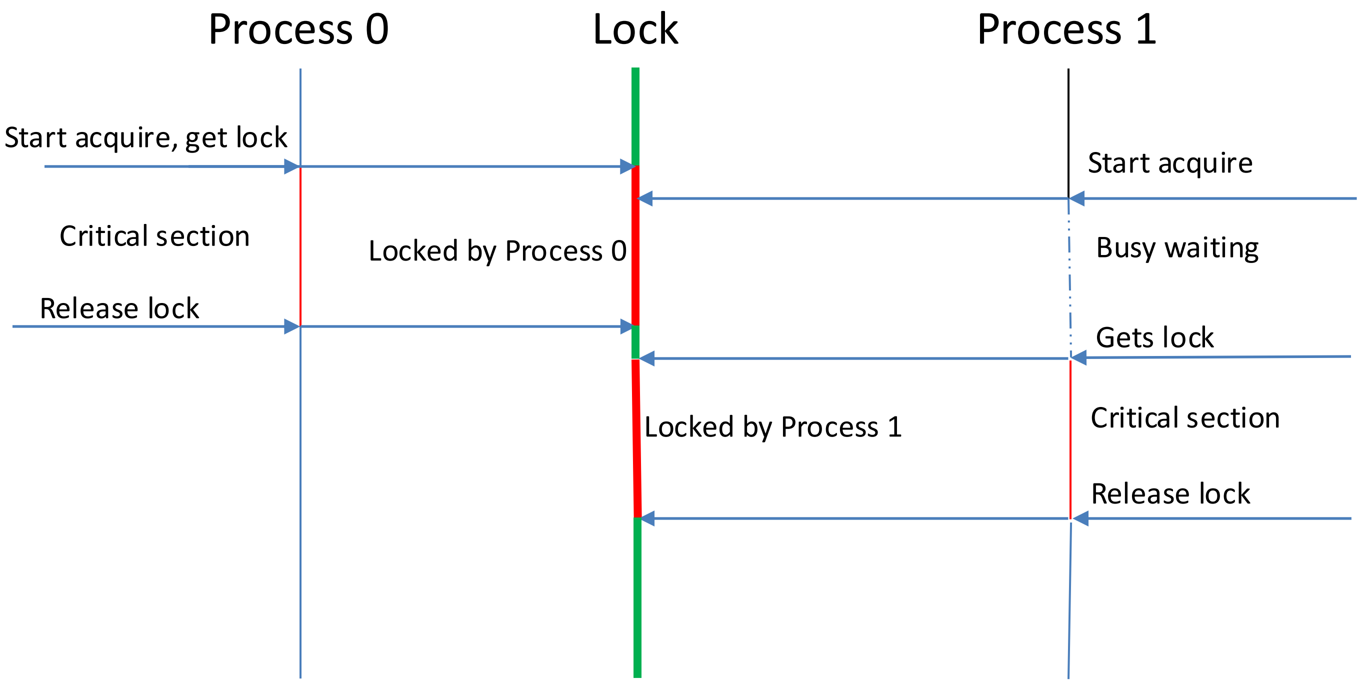
`acquire() {`
`    while (!available)`
`        ; /* busy wait */`
	`release() {`
	`    available = true;`
	`}`

- **Usage**

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


Colorado State University

acquire() and release()



Colorado State University

How are locks supported by hardware?

- Atomic read-modify-write
- Atomic instructions in x86
  - LOCK instruction prefix, which applies to an instruction does a read-modify-write on memory (INC, XCHG, CMPXCHG etc)
  - Ex: lock cmpxchg <dest>, <source>
- In RISK processors? Instruction-pairs
  - LL (Load Linked Word), SC (Store Conditional Word) instructions in MIPS
  - LDREX, STREX in ARM
  - Creates an atomic sequence

Semaphores by Dijkstra

- 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
  - **`wait()`** and **`signal()`**
    - Originally called **`P()`** and **`V()`** **`based on Dutch words`**
- Definition of the **`wait() operation`**

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

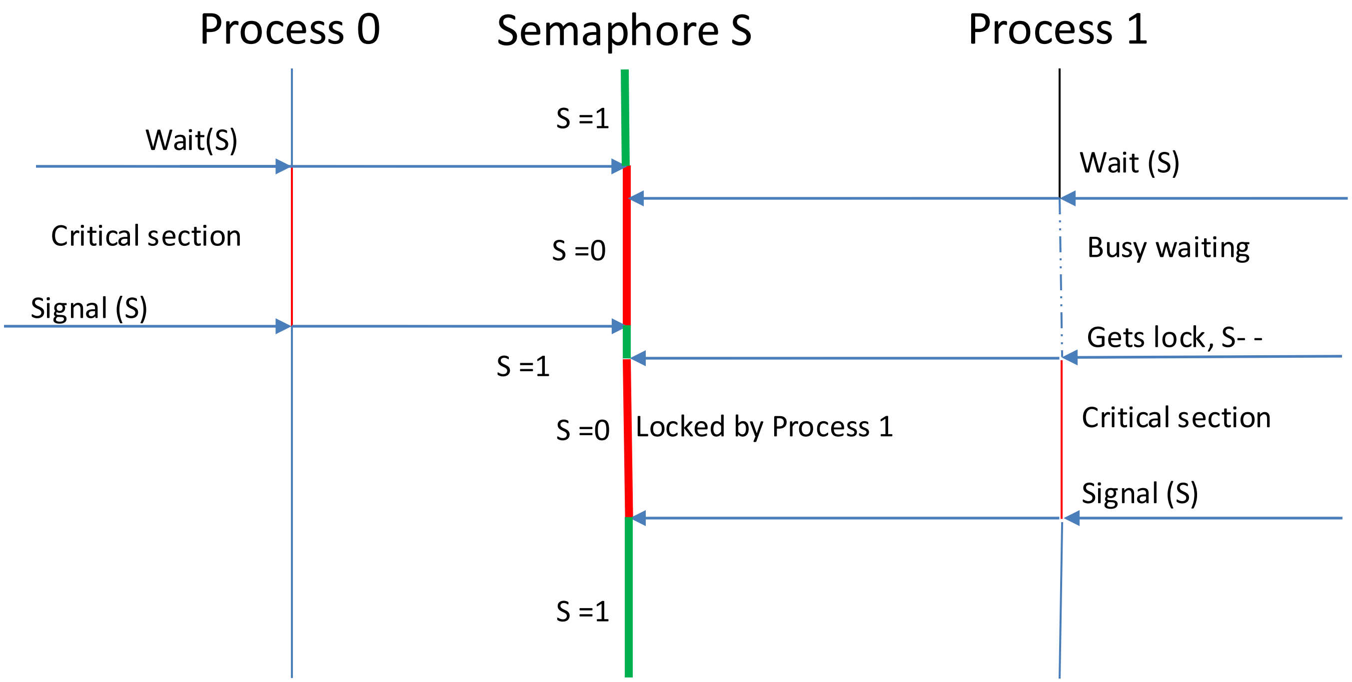
Waits until another process makes S=1

- Definition of the **`signal() operation`**

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

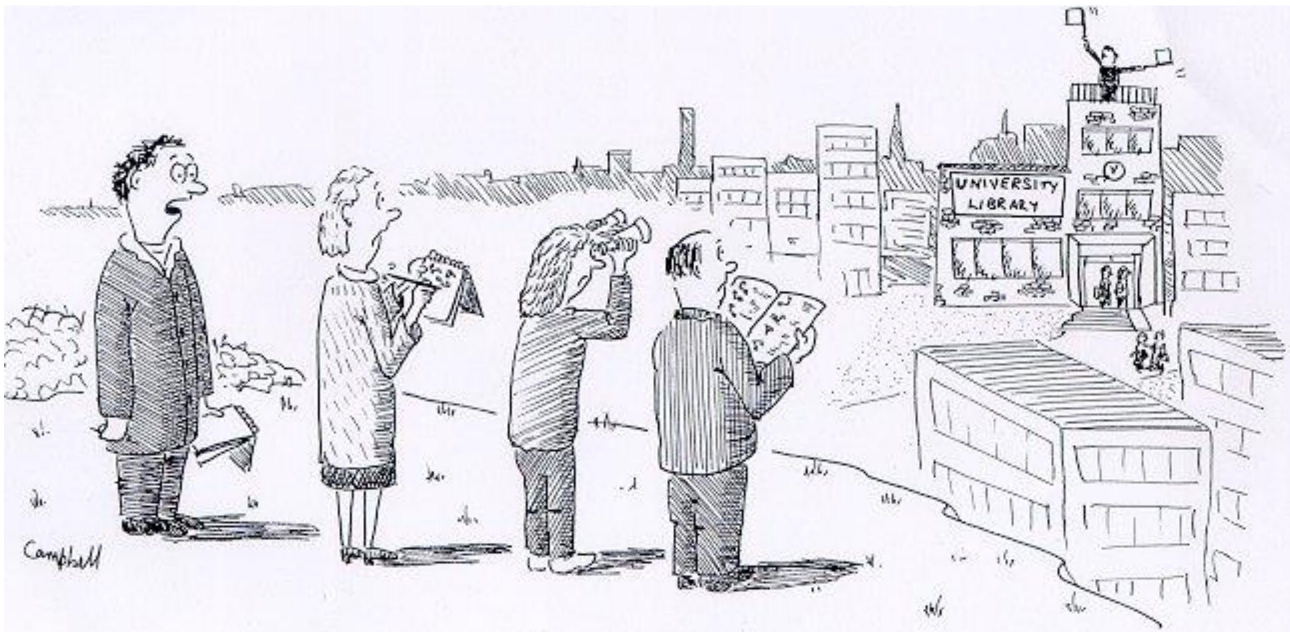
Binary semaphore: When s is 0 or 1, it is a mutex lock

Wait(S) and Signal (S)



Colorado State University

Semaphores



I was hoping the distance learning service might use more up-to-date technology

Semaphore Usage

- **Counting semaphore** – integer value can range over an unrestricted domain
- **Binary semaphore** – integer value can range only between 0 and 1
  - Practically same as a **mutex lock**
- Can solve various synchronization problems
- Ex: Consider P_1 and P_2 that requires event S_1 to happen before S_2
  Create a semaphore "**synch**" initialized to 0 i.e not available

P1:	P2:
S_1;	wait(synch);
signal(synch);	S_2;

- Can implement a counting semaphore ***S*** as a binary semaphore

The counting semaphore

- **Controls access to a finite set of resources**
- Initialized to the number of resources
- Usage:
  - Wait (S): to use a resource
  - Signal (S): to release a resource
- When all resources are being used: S == 0
  - Block until S > 0 to use the resource

Applicable to different types of synchronization problems.

0: no waiting threads (or processes)

Positive: no waiting threads, a wait operation would not put the invoking thread in queue.

Negative: number of threads waiting

Semaphore Implementation

- Must guarantee that no two processes can execute the `wait()` and `signal()` on the same semaphore at the same time
- Thus, the implementation becomes the critical section problem where the `wait` and `signal` code are placed in the critical section
  - Could now have **busy waiting** in critical section implementation
    - But implementation code is short
    - Little busy waiting if critical section rarely occupied
- Note that some applications may spend lots of time in critical sections and therefore this is not a good solution
- Alternative: block and wakeup (next slide)

Semaphore Implementation with no Busy waiting

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

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

Implementation with no Busy waiting (Cont.)

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

If value < 0
abs(value) is the number of waiting processes

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

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

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

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

  - P0 executes wait(s), P1 executes wait(Q)
    - P0 must wait till P1 executes signal(Q)
    - P1 must wait till P0 executes signal(S) Deadlock!

Priority Inversion

- **Priority Inversion** – Scheduling problem when lower-priority process P_L holds a lock needed by higher-priority process P_H.
  - The low priority task may be preempted by a medium priority task P_M which does not use the lock, causing P_H to wait because of P_M.

Mars pathfinder
Mission problem 1997

- Solved via **priority-inheritance protocol**
  - Process accessing resource needed by higher priority process Inherits higher priority till it finishes resource use
  - Once done, process reverts to lower priority

Classical Problems of Synchronization

- Classical problems used to test newly-proposed synchronization schemes
  - Bounded-Buffer Problem
  - Readers and Writers Problem
  - Dining-Philosophers Problem
- Monitors: higher level handling of synchronization

Bounded-Buffer Problem

- *n* buffers, each can hold one item
- Binary semaphore (mutex)
  - Provides mutual exclusion for accesses to buffer pool
  - Initialized to 1
- Counting semaphores
  - empty: Number of empty slots available
    - Initialized to n
  - full: Number of filled slots available n
    - Initialized to 0

3 semaphores needed,
1 binary, 2 counting

Bounded-Buffer : Note

- Producer and consumer must be ready before they attempt to enter critical section
- Producer readiness?
  - When a slot is available to add produced item
    - wait(empty)
      - empty is initialized to n
- Consumer readiness?
  - When a producer has added new item to the buffer
    - wait(full)
      - full initialized to 0

empty: Number of empty slots available
wait(empty) wait until at least 1 empty

full: Number of filled slots available
wait(full) wait until at least 1 full

Bounded Buffer Problem (Cont.)

The structure of the producer process

empty: initialized to n
full: initialized to 0

```
do {
      ...
       /* produce an item in next_produced */
      ...
   wait(empty);              wait till slot available
   wait(mutex);          Allow producer OR consumer to (re)enter critical section
      ...
      /* add next produced to the buffer */
      ...
   signal(mutex); Allow producer OR consumer to (re)enter critical section
   signal(full);             signal consumer that a slot is available
 } while (true);
```

Colorado State University

Bounded Buffer Problem (Cont.)

The structure of the consumer process

empty: initialized to n
full: initialized to 0

```
Do {
    wait(full);  wait till slot available for consumption
    wait(mutex);  Only producer OR consumer can be in critical section
        . . .
    /* remove an item from buffer to next_consumed */
        . . .
    signal(mutex);  Allow producer OR consumer to (re)enter critical section
    signal(empty);  signal producer that a slot is available to add
        . . .
    /* consume the item in next consumed */
        . . .
 } while (true);
```

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. No readers permitted when writer is accessing the data.
- Several variations of how readers and writers are considered – all involve some form of priorities

Readers-Writers Problem

- Shared Data
  - Data set
  - Semaphore **`rw_mutex`** initialized to 1 (mutual exclusion for writer)
  - Semaphore **`mutex`** initialized to 1 (mutual exclusion for read_count)
  - Integer **`read_count`** initialized to 0 (how many readers?)

Readers-Writers Problem (Cont.)

- The structure of a writer process

```
do {
     wait(rw_mutex);
          ...
    /* writing is performed */
          ...
     signal(rw_mutex);
} while (true);
```

Colorado State University

Readers-Writers Problem (Cont.)

- The structure of a reader process

```
do {
     wait(mutex);
        read_count++;
        if (read_count == 1)
              wait(rw_mutex);
     signal(mutex);
          ...
        /* reading is performed */
          ...
     wait(mutex);
        read count--;
        if (read_count == 0)
             signal(rw_mutex);
     signal(mutex);
} while (true);
```

First reader needs to wait for the writer to finish.
If other readers are already reading, a new reader
Process just goes in.

mutex for mutual exclusion to read_count

Cannot read if writer is writing

When:
writer in critical section
and if n readers waiting
1 is queued on rw_mutex
(n-1) queued on mutex

When the last reader leaves, a writer can go in.

Readers-Writers Problem Variations

- ***First*** variation – no reader kept waiting unless writer has already obtained permission to use shared object
- ***Second*** variation – once writer is ready, it performs the write ASAP, i.e. if a writer is waiting, no new readers may start.
- Both may have starvation leading to even more variations
- Problem is solved on some systems by kernel providing reader-writer locks