CSE 380
Computer Operating Systems

Instructor: Insup Lee

University of Pennsylvania
Fall 2003

Lecture 2.4: Interprocess Communication

Communicating Processes

- Many applications require processes to communicate and synchronize with each other
- Main problem: operations of different processes are interleaved in an unpredictable manner
- Same issues in multiple contexts
  - Multiple threads of same process accessing shared data
  - Kernel processes accessing shared data structures
  - User processes communicating via shared files
  - User processes communicating via shared objects in kernel space
  - High-level programming languages supporting parallelism
  - Database transactions

Example: Shared variable problem

- ❑ Two processes are each reading characters typed at their respective terminals
- ❑ Want to keep a running count of total number of characters typed on both terminals
- ❑ A shared variable V is introduced; each time a character is typed, a process uses the code:

  V := V + 1;

  to update the count.
- ❑ During testing it is observed that the count recorded in V is less than the actual number of characters typed. What happened?

Analysis of the problem

The programmer failed to realize that the assignment was not executed as a single indivisible action, but rather as an arbitrary shuffle of following sequences of instructions:

```
P1. MOVE V, r0                 Q1. MOVE V,
 r1
P2. INCR r0                Q2. INCR r1
P3. MOVE r0, V                 Q3. MOVE r1,
 V
```

The interleaving P1, Q1, P2, Q2, P3, Q3 increments V only by 1

Sample Question

```
interleave () {
  pthread_t th0, th1;
  int count=0;
  pthread_create(&th0,0,test,0);
  pthread_create(&th1,0,test,0);
  pthread_join(th0,0);
  pthread_join(th1,0);
  printf(count);
}
test () {
  for (int j=0; j<MAX; j++) count=count+1;
}
```

What's minimum/ maximum value output?

Sample Question

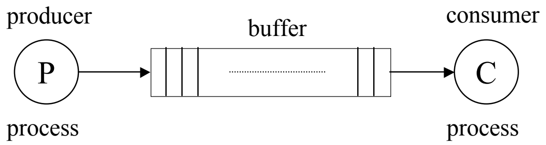
```
int count = 0; /* global var */
interleave () {
  pthread_t th0, th1;
  pthread_create(&th0,0,test,0);
  pthread_create(&th1,0,test,0);
  pthread_join(th0,0);
  pthread_join(th1,0);
  printf(count);
}
test () {
 for (int j=0; j<MAX; j++)
   count=count+1;
}
```

Maximum: 2 MAX, Minimum 2

For Minimum, consider the sequence:

- **Both threads read count as 0**
- **th0 increments count MAX-1 times**
- **th1 writes 1**
- **th0, in its last iteration, reads count=1**
- **th1 finishes all its iterations**
- **th0 writes 2 to count and ends**

The Producer/Consumer Problem



- from time to time, the producer places an item in the buffer
- the consumer removes an item from the buffer
- careful synchronization required
- the consumer must wait if the buffer empty
- the producer must wait if the buffer full
- typical solution would involve a shared variable count
- also known as the Bounded Buffer problem
- Example: in UNIX shell

eqn myfile.t | troff

Push and Pop example

```
struct stacknode {
   int data;
   struct stacknode *nextptr;
};

typedef struct stacknode STACKNODE;
typedef STACKNODE *STACKNODEPTR;

void push (STACKNODEPTR *topptr, int info)
{
   STACKNODEPTR newptr;
   newptr = malloc (sizeof (STACKNODE));
   newptr->date = info;          /* Push 1 */
   newptr->nextptr = *topptr;    /* Push 2 */
   *topptr = newptr;             /* Push 3 */
}
```

Pop

```
int pop (STACKNODEPTR *topptr)
{
   STACKNODEPTR tempptr;
   int popvalue;
   tempptr = *topptr;             /* Pop 1 */
   popvalue = (*topptr)->data;    /* Pop 2 */
   *topptr = (*topptr)->nextptr;  /* Pop 3 */
   free(tempptr);
   return popvalue;
}
```

Question: Is it possible to find an interleaved execution of Push 1, Push 2, …, Pop 3 such that the resulting data structure becomes inconsistent?

Issues in Concurrent Programming

- ❑ Operations on shared data structures typically correspond to a sequence of instructions
- ❑ When two processes/threads are executing concurrently, the result depends on the precise interleaving of the two instruction streams (this is called **race condition**)
- ❑ Race conditions could cause bugs which are hard to reproduce
- ❑ Besides race condition, the second issue is **synchronization** (one process is waiting for the results computed by another)
  - Can we avoid busy waiting?

Overview of Solutions

Idealized Problems
  - Producer-Consumer
  - Dining Philosophers
  - Readers-Writers

High-level Synchronization Primitives
  - Monitors (Hoare, Brinch-Hansen)
  - Synchronized method in Java

OS-level support (mutual exclusion and synchronization)
  - Special variables: Semaphores, Mutexes
  - Message passing primitives (send and receive)

Low-level (for mutual exclusion)
  - Interrupt disabling
  - Using read/write instructions
  - Using powerful instructions (Test-and-set, Compare-and Swap…)

Mutual Exclusion Problem

❑ Motivation: Race conditions can lead to undesirable effects

❑ Solution:

- Identify a block of instructions involving shared memory access that should be executed without interference from others
- This block of code is called **critical region/section** (e.g., the entire assignment **"V:=V+1"** in our first example)
- Ensure that processes execute respective critical sections in a mutually exclusive manner

❑ Mutual exclusion is required in multiple contexts where simultaneous access to shared data needs to enforce integrity constraints (e.g., airline reservation system)

Requirements for solutions to Mutual Exclusion Problem

1. **Safety:** No two processes should be simultaneously in their critical regions
2. **Generality:** No assumptions should be made about speeds or numbers of CPUs (i.e., it should work in the worst case scenario)
3. **Absence of deadlocks:** Should not reach a state where each process is waiting for the other, and nobody gets to enter
4. **Bounded liveness (or fairness):** If a process wants to enter a critical section then it should eventually get a chance

---

Low-level solution: Disable interrupts

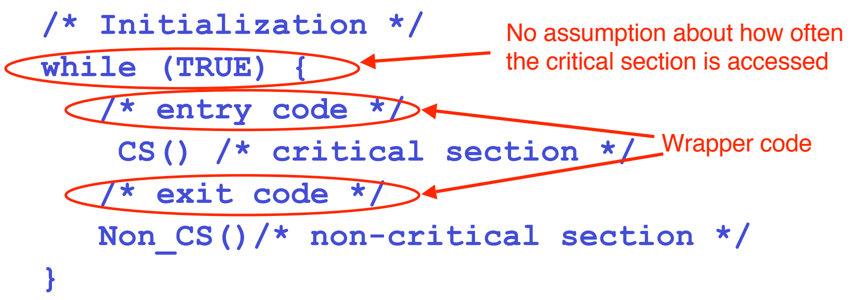
```
process A                    process B
     . . .                        . . .
     disable interrupts           disable interrupts
       CS                           CS
     enable interrupts            enable interrupts
```

- Prevents context-switch during execution of CS
- Recall maskable interrupts in Pentium architecture
- This is sometimes necessary (to prevent further interrupts during interrupt handling)
- Not a good solution for user programs (too powerful and not flexible)

Shared Variable Solutions
General Skeleton

Two processes with shared variables

```
Assumption: Reads and Writes are atomic
Each process P0 and P1 executes
```



```
  /* Initialization */
  while (TRUE) {
     /* entry code */
       CS() /* critical section */
     /* exit code */
      Non_CS()/* non-critical section */
  }
```

No assumption about how often the critical section is accessed

Wrapper code

Using mutual exclusion

```
int count=0, turn=0; /* global vars */
bool flag[1]=false;  /* global array */
interleave () {
  pthread_t th0, th1;
  pthread_create(&th0,0,test,0);
  pthread_create(&th1,0,test,1);
  pthread_join(th0,0);
  pthread_join(th1,0);
  printf(count); /* count is guaranteed to be 2 MAX */
}
test (int i) {
  for (int j=0; j<MAX; j++) {
      flag[i]=true; turn=i; /* Entry code of Peterson */
      repeat until (flag[1-i]==false | turn!=i);
      count=count+1; /* critical section */
      flag[i]=false; /* exit code of Peterson soln */
      }
}
```

Proof of Mutual Exclusion

- To prove: P0 and P1 can never be simultaneously in CS
- Observation: Once P0 sets flag[0], it stays true until P0 leaves the critical section (same for P1)
- Proof by contradiction. Suppose at time t both P0 and P1 are in CS
- Let t0/t1 be the times of the most recent executions of the assignments turn = 0 / turn =1 by P0 / P1, resp.
- Suppose t0 < t1
- During the period t0 to t, flag[0] equals TRUE
- During the period from t1 to t, turn equals to 1
- Hence, during the period t1 to t, P1 is blocked from entering its CS; a contradiction.

Also satisfies bounded liveness (why?)

1st Attempt for Mutual exclusion

```
Shared variable: turn :{0,1}
turn==i means process Pi is allowed to enter
Initial value of turn doesn’t matter
Solution for process P0: (P1 is symmetric)
while (TRUE) {
  while (turn != 0);  /* busy waiting */
  CS();
  turn = 1; /* be fair to other */
  Non_CS();
 }
```

Ensures mutual exclusion, but requires strict alternation
A process cannot enter its CS twice in succession
even if the other process does not need to enter CS

2nd Attempt

```
Shared variable: flag[i] : boolean, initially FALSE
Solution for process P0: (P1 is symmetric)
while (TRUE) {
  while (flag[1]); /* wait if P1 is trying */
  flag[0] = TRUE;  /* declare your entry */
  CS();
  flag[0] = FALSE; /* unblock P1 */
  Non_CS();
}
```

Mutual Exclusion is violated:

P0 tests flag[1] and finds it False
P1 tests flag[0] and finds it False
Both proceed, set their flags to True and enter CS

3rd Attempt

```
Shared variable: flag[i] : boolean, initially FALSE
Solution for process P0: (P1 is symmetric)
while (TRUE) {
  flag[0] = TRUE;  /* declare entry first */
  while (flag[1]);  /* wait if P1 is also trying */
  CS();
  flag[0] = FALSE; /* unblock P1 */
  Non_CS();
}
```

Leads to deadlock:

P0 sets flag[0] to TRUE
P1 sets flag[1] to TRUE
Both enter their while loops and keep waiting

Peterson's Solution

```
Shared variables: flag[i] :boolean; turn :{0,1}
Solution for process P0: (P1 is symmetric)
flag[0] = FALSE;
while (TRUE) {
  flag[0] = TRUE;   /* declare interest */
  turn = 0; /* takes care of race condition */
  repeat until (  /* busy wait */
     flag[1] == FALSE
    | turn != 0);
  CS();
  flag[0] = FALSE; /* unblock P1 */
  Non_CS();
}
```

P1 is not contending

P1 is contending, but
turn = 1 executed before turn = 0

Hardware Supported Solution

- ❑ Challenge so far was designing a solution assuming instruction-set supported only load and store
- ❑ If reading and writing can be done in one instruction, designing solutions is much easier
- ❑ A popular instruction: test-and-set

  TSL X, L X: register, L : memory loc (bit)

  L's content are loaded into X, and L is set to 1
- ❑ Test-and-set seems simple, but can be used to implement complex synchronization schemes
- ❑ Similarly powerful instructions:
  - SWAP (L1, L2) : atomically swaps the two locations
  - Compare and swap (Pentium)
  - Load-linked/Store conditional (MIPS)

Hardware Instructions

- MIPS -- Load-Linked/Store Conditional (LL/SC)
- Pentium -- Compare and Exchange, Exchange, Fetch and Add
- SPARC -- Load Store Unsigned Bit (LDSTUB) in v9
- PowerPC -- Load Word and Reserve (lwarx) and Store Word Conitional (stwcx)

Locks

- lock (x) performs
-
  ```
  lock: [ if x = false then x := true
                       else go to lock ]
  ```
- unlock (x) performs
-
  ```
        [x := false ]
  ```
- E.g.,
-
  ```
  var x : boolean
  parbegin
      P1:  ... lock(x); CS_1; unlock(x) …
      …
      Pn:  ... lock(x); CS_n; unlock(x) …
  parend
  ```

Properties

- Starvation is possible.
- Busy waiting.
- Different locks may be used for different shared resources.
- Proper use not enforced. E.g., forget to lock.

How to implement locks

- Requires an atomic (uninterruptable at the memory level) operations like test-and-set or swap.
-
```
atomic function TestAndSet
             (var x: boolean): boolean;
   begin
     TestAndSet := x;
     x := true;
   end
```
-
```
procedure Lock (var x : boolean);
     while TestAndSet(x) do skip od;
```
-
```
procedure Unlock (var x: boolean);
     x := false;
```
- (1) If not supported by hardware, TestAndSet can be implemented by disabling and unabling interrupts.
  (2) Lock can also be implemented using atomic swap(x,y).

Solution using TSL

```
Shared variable: lock :{0,1}
lock==1 means some process is in CS
Initially lock is 0
Code for process P0 as well as P1:
while (TRUE) {
  try: TSL X, lock  /* test-and-set lock */
  if (X!=0) goto try; /*retry if lock set*/
  CS();
  lock = 0; /* reset the lock */
  Non_CS();
 }
```

CSE 380
Computer Operating Systems

Instructor: Insup Lee

University of Pennsylvania
Fall 2003

Lecture 2.5: Process Synchronization Primitives

Avoiding Waiting

- Solutions seen so far teach us:
  - How to ensure exclusive access
  - How to avoid deadlocks
- But, in all cases, if P0 is in CS, and P1 needs CS, then P1 is busy waiting, checking repeatedly for some condition to hold. Waste of system resources!
- Suppose we have following system calls for synchronization
  - Sleep: puts the calling thread/process in a blocked/waiting state
  - Wakeup(arg): puts the argument thread/process in ready state

---

Sleep/wakeup Solution to Producer-Consumer Problem

- bounded buffer (of size N)
- producer writes to it, consumer reads from it
- Solution using sleep/wakeup synchronization

```
int count = 0          /* number of items in buffer */
```

Producer code:

```
 while (TRUE) {
   /* produce */
   if (count == N) sleep;
  /* add to buffer */
   count = count + 1;
   if (count == 1)
    wakeup(Consumer);
}
```

Consumer code:

```
 while (TRUE) {
   if (count==0) sleep;
  /* remove from buffer */
   count = count -1;
   if (count == N-1)
        wakeup(Producer);
   /* consume */
}
```

Problematic Scenario

- Count is initially 0
- Consumer reads the count
- Producer produces the item, inserts it, and increments count to 1
- Producer executes wakeup, but there is no waiting consumer (at this point)
- Consumer continues its execution and goes to sleep
- Consumer stays blocked forever unnecessarily
- Main problem: wakeup was lost

Solution: Semaphores keeping counts

Dijkstra's Semaphores

- A semaphore **s** has a non-negative integer value
- It supports two operations
- **up(s)** or V(s) : simply increments the value of s
- **down(s)** or P(s) : decrements the value of s if s is positive, else makes the calling process wait
- When s is 0, down(s) does not cause busy waiting, rather puts the process in sleep state
- Internally, there is a queue of sleeping processes
- When s is 0, up(s) also wakes up one sleeping process (if there are any)
- up and down calls are executed as atomic actions

Mutual Exclusion using Semaphores

Shared variable: a single semaphore s == 1
Solution for any process

```
while (TRUE) {
  down(s);  /* wait for s to be 1 */
  CS();
  up(s); /* unblock a waiting process */
  Non_CS();
}
```

- ❑ No busy waiting
- ❑ Works for an arbitrary number of processes, i ranges over 0..n

Potential Implementation

```
 typedef struct {
      int value;
      *pid_t wait_list; /* list of processes
       } semaphore;


down( semaphore S){
    if (S.value >0) S.value--;
    else { add this process to S.wait_list;
            sleep;
          }

 up( semaphore S){
     if (S.wait_list==null) S.value++;
     else {  remove a process P from S.wait_list;
             wakeup(P);
          }
```

To ensure atomicity of up and down, they are included in a critical section, maybe by disabling interrupts

The Producer-Consumer Problem

- bounded buffer (of size n)
- one set of processes (producers) write to it
- one set of processes (consumers) read from it

```
semaphore:    full = 0     /* number of full slots */
              empty = n    /* number of empty slots */
              mutex = 1    /* binary semaphore for CS */
```

Producer code:

```
while (TRUE) {
   /* produce */
   down (empty)
   down (mutex)
  /* add to buffer */
   up (mutex)
   up (full)
}
```

Consumer code:

```
while (TRUE) {
   down (full)
   down (mutex)
  /* remove from buffer */
   up (mutex)
   up (empty)
   /* consume */
}
```

Mutual exclusion
For accessing buffer

The Producer-Consumer Problem

```
semaphore:    full = 0     /* number of full slots */
              empty = n    /* number of empty slots */
              mutex = 1    /* binary semaphore for CS */
```

Producer code:

```
while (TRUE) {
   /* produce */
   down (empty)
   down (mutex)
  /* add to buffer */
   up (mutex)
   up (full)
}
```

Consumer code:

```
while (TRUE) {
   down (full)
   down (mutex)
  /* remove from buffer */
   up (mutex)
   up (empty)
   /* consume */
}
```

What happens if we switch the order of down(empty) and down(mutex) ?
What happens if we switch the order of up(mutex) and up(full) ?

POSIX Semaphore System Calls

- *int sem_init(sem_t *sp, unsigned int count, int type):* Initialize semaphore pointed to by sp to count. *type* can assign several different types of behaviors to a semaphore
- *int sem_destroy(sem_t *sp);* destroys any state related to the semaphore pointed to by sp.
- *int sem_wait(sem_t *sp);* blocks the calling thread until the semaphore count pointed to by sp is greater than zero, and then it atomically decrements the count.
- *int sem_trywait(sem_t *sp);* atomically decrements the semaphore count pointed to by sp, if the count is greater than zero; otherwise, it returns an error.
- *int sem_post(sem_t *sp);* atomically increments the semaphore count pointed to by sp. If there are any threads blocked on the semaphore, one will be unblocked.

Roadmap

Idealized Problems
- Producer-Consumer
- Dining Philosophers
- Readers-Writers

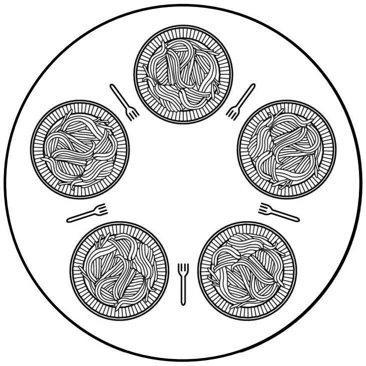
High-level Synchronization Primitives
- Monitors (Hoare, Brinch-Hansen)
- Synchronized method in Java

OS-level support (mutual exclusion and synchronization)
- Special variables: Semaphores, Mutexes
- Message passing primitives (send and receive)

Low-level (for mutual exclusion)
- Interrupt disabling
- Using read/write instructions
- Using powerful instructions (Test-and-set, Compare-and Swap…)

Dining Philosophers

- ❑ Philosophers eat/think
- ❑ Eating needs 2 forks
- ❑ Pick one fork at a time
- ❑ How to prevent deadlock



The Dining Philosopher Problem

- Five philosopher spend their lives thinking + eating.
- One simple solution is to represent each chopstick by a semaphore.
- Down (i.e., P) before picking it up & up (i.e., V) after using it.

```
❑ var chopstick: array[0..4] of semaphores=1
  philosopher i
❑   repeat
      down( chopstick[i] );
      down( chopstick[i+1 mod 5] );
         ...
         eat
         ...
      up( chopstick[i] );
      up( chopstick[i+1 mod 5] );
         ...
         think
         ...
    forever
```

- Is deadlock possible?

Number of possible states

- 5 philosophers
- Local state (LC) for each philosoper
  - thinking, waiting, eating
- Glabal state = (LC 1, LC 2, …, LC5)
  - E.g., (thinking, waiting, waiting, eating, thinking)
  - E.g., (waiting, eating, waiting, eating, waiting)
- So, the number of global states are 3 ** 5 = 243
- Actually, it is a lot more than this since waiting can be
  - Waiting for the first fork
  - Waiting for the second fork

Number of possible behaviors

- Sequence of states
- Initial state: (thinking,thinking,thinking,thinking,thinking)
- The number of possible behaviors = 5 x 5 x 5 x …
- Deadlock state: (waiting,waiting,waiting,waiting, waiting)
- Given the state transition model of your implementation, show that it is not possible to reach the deadlock state from the initial state.

The Readers and Writers Problem

Shared data to be accessed in two modes: reading and writing.

- Any number of processes permitted to read at one time
- writes must exclude all other operations.

Intuitively:

```
  Reader:                        | Writer:
   when(no_writers==0) {         |  when(no_readers==0
     no_readers=no_readers+1     |    and no_writers==0) {
                                 |     no_writers = 1
                                 |
     <read>                      |     <write>
                                 |
     no_readers=no_readers-1     |     no_writers = 0
     .                           |     .
     .                           |     .
```

A Solution to the R/W problem

```
Semaphores:
   mutex = 1 /*mutual excl. for updating readcount */
   wrt = 1   /* mutual excl. for writer */
   int  readcount = 0
Reader:    down(mutex)
           readcount = readcount + 1
           if readcount == 1 then down(wrt)
           up(mutex)
            <read>
           down(mutex)
           readcount = readcount - 1
           if readcount == 0 then up(wrt)
           up(mutex)

Writer: down(wrt);  <write>  up(wrt)
```

Notes: wrt also used by first/last reader that enters/exits critical section. Solution gives priority to readers in that writers can be starved by a stream of readers.

Readers and Writers Problem

- Goal: Design critical section access so that it has
  - Either a single writer
  - Or one or more readers (a reader should not block another reader)
- First step: Let’s use a semaphore, wrt, that protects the critical section
  - Initially wrt is 1
  - wrt should be zero whenever a reader or writer is inside it
- Code for writer:

```
down(wrt); write(); up(wrt);
```

- How to design a reader?
  - Only the first reader should test the semaphore (i.e., execute down(wrt)

Readers and Writers Problem

- More on Reader’s code
  - To find out if you the first one, maintain a counter, readcount, that keeps the number of readers
- First attempt for reader code:

```
readcount++;
if (readcount==1)  down(wrt);
read();
readcount--;
```

- What are the problems with above code?

Readers and Writers Problem

- Corrected reader code:

```
down(mutex); /* mutex: semaphore protecting updates to readcount
readcount++;
if (readcount==1) down(wrt);
up(mutex);
read();
down(mutex);
readcount--;
if (readcount==0) up(wrt);
up(mutex);
```

- What happens if a new reader shows up if a writer is waiting while one or more readers are reading?

Monitors

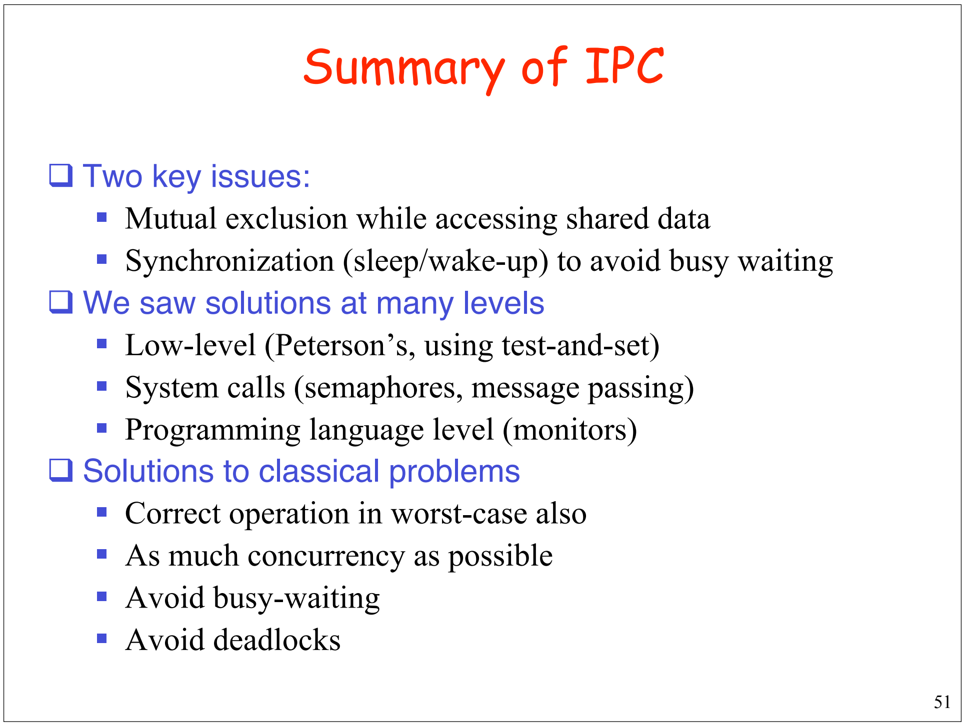
- Semaphores are powerful, but low-level, and can lead to many programming errors
- Elegant, high-level, programming-language-based solution is monitors (proposed by Hoare and by Brinch Hansen)
- A monitor is a shared data object together with a set of operations to manipulate it.
- To enforce mutual exclusion, at most one process may execute a method for the monitor object at any given time.
- All uses of shared variables should be encapsulated by monitors.
- Data type “condition” for synchronization (can be waited or signaled within a monitor procedure)
- Two operations on “condition” variables:
  - wait: Forces the caller to be delayed, releases the exclusive access.
  - signal: One of the waiting processes is resumed.
- “synchronized” methods in Java are similar

Traffic Synchronization

- ❑ Suppose there is a two-way traffic with a one-way tunnel at some point
- ❑ Goal: a northbound car should wait if the tunnel has cars in the other direction
- ❑ Monitor-based solution:
  - Tunnel is a monitor with two variables, nb and sb, keeping track of cars in the two direction
  - Southbound car checks nb, and if nb is 0, increments sb and proceeds (northbound car is symmetric)
  - Methods of a monitor are executed exclusively
  - To avoid busy waiting, use a condition variable, busy
  - A southbound car, if nb is positive, executes wait on busy, and the last northbound car will wake all of the waiting cars

Monitor-based Solution

```
monitor tunnel {
   int nb=0, sb=0;
   condition busy;
 public:
    northboundArrive() {
       if (sb>0) busy.wait;
       nb = nb +1;
       };
    northboundDepart() {
       nb = nb -1;
       if (nb==0)
               while (busy.queue) busy.signal;
       };
```



Summary of IPC

- Two key issues:
  - Mutual exclusion while accessing shared data
  - Synchronization (sleep/wake-up) to avoid busy waiting
- We saw solutions at many levels
  - Low-level (Peterson’s, using test-and-set)
  - System calls (semaphores, message passing)
  - Programming language level (monitors)
- Solutions to classical problems
  - Correct operation in worst-case also
  - As much concurrency as possible
  - Avoid busy-waiting
  - Avoid deadlocks