CG2271

Real-Time Operating Systems

Lecture 7

Inter-Task Communication

colintan@nus.edu.sg

National University of Singapore

School of Computing

Learning Objectives

- **By the end of this lecture you will be able to:**
  - Understand what race conditions are, and why they are bad.
  - Understand the various ways to prevent race conditions.
  - Understand how to pass messages between tasks.

NUS National University of Singapore School of Computing

Introduction

- **In the "previous, previous" lecture we looked at how multiple tasks can run on a single CPU.**
- **In the previous lecture, we then looked at how a kernel chooses which task to run next.**
  - We basically assumed that tasks are independent!
- **In real-world applications, there are "dependencies" between tasks.**
  - Task B cannot proceed because it is waiting for Task A's result.
  - Task B and Task A update the same shared variable, which can result in errors.
  - …

Introduction

- **If both Task A and Task B are allowed to run freely, errors will occur.**
  - Task B proceeds before Task A completes, resulting in B using stale results.
  - Task A and B update a variable at the same time, causing one task to over-write the results of the other task.
  - Etc.
- **Some form of coordination is therefore required!**
- **This lecture comes from Modern Operating Systems.**
  - MOS uses the term "process" instead of task.
  - To be consistent with the book, we will therefore also say "process" instead of task.
    - ✓**Just remember that they mean the same thing!!**

Inter-Task Communications

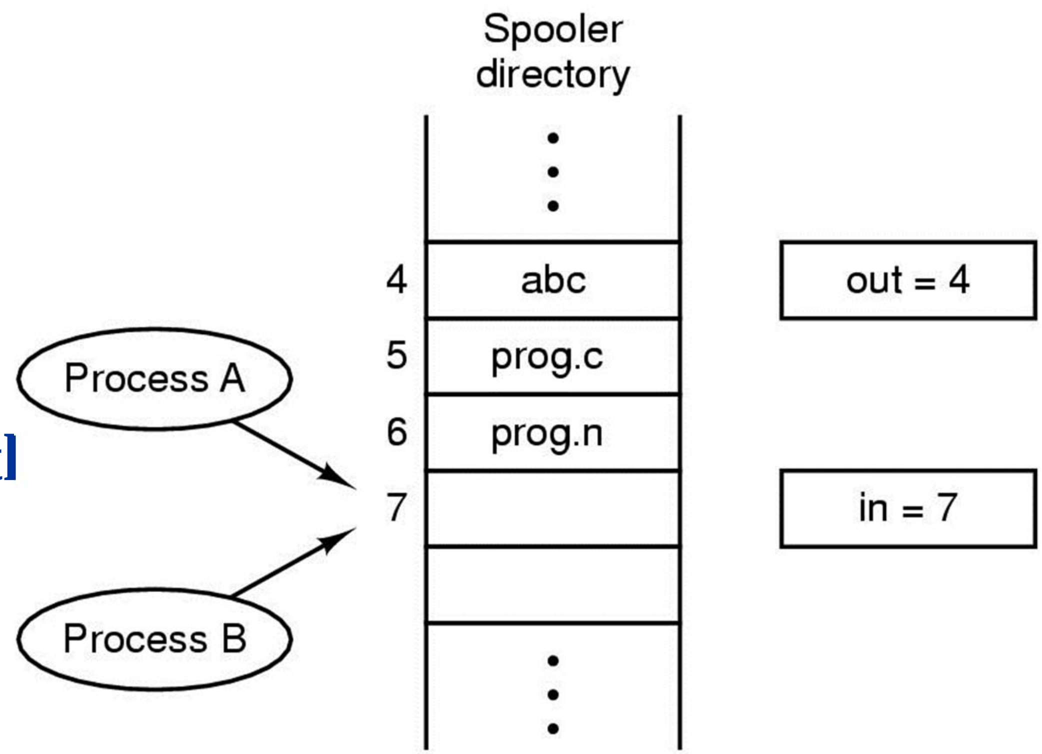
RACE CONDITIONS AND CRITICAL SECTIONS

Race Conditions

- **Race conditions occur when two or more processes attempt to access shared storage.**
  - This causes the final outcome to depend on who runs first.
  - “Shared storage” can mean:
    - ✓**Global variables.**
    - ✓**Memory locations.**
    - ✓**Hardware registers.**
      - *This refers to configuration registers rather than CPU registers.*
    - ✓**Files.**
  - To understand race conditions, we will consider the example of a queue in a print spooler.

Race Conditions

- **A process that wants to print enters the name of the file into a print directory.**
- **A print daemon (printd) periodically checks the directory, prints out the next file and removes its entry.**
- **Two variables keep track of tl queue:**
  - IN: Next available slot.
  - OUT: Next file for printing.



Race Conditions

- **The following can happen:**
  - Process A reads IN as 7 and stores it in a local variable *next*.
  - The OS pre-empts A and starts B.
  - B reads IN as 7 and stores it in a local variable *next*.
  - B inserts its file into slot 7 and updates IN to 8.
  - B is pre-empted and A restarts.
  - A still thinks slot 7 is available and inserts its file there, overwriting B's file, then updates IN to 8.
- **The daemon is unaware of the mistake, and B never gets a printout. ☹**

Critical Sections

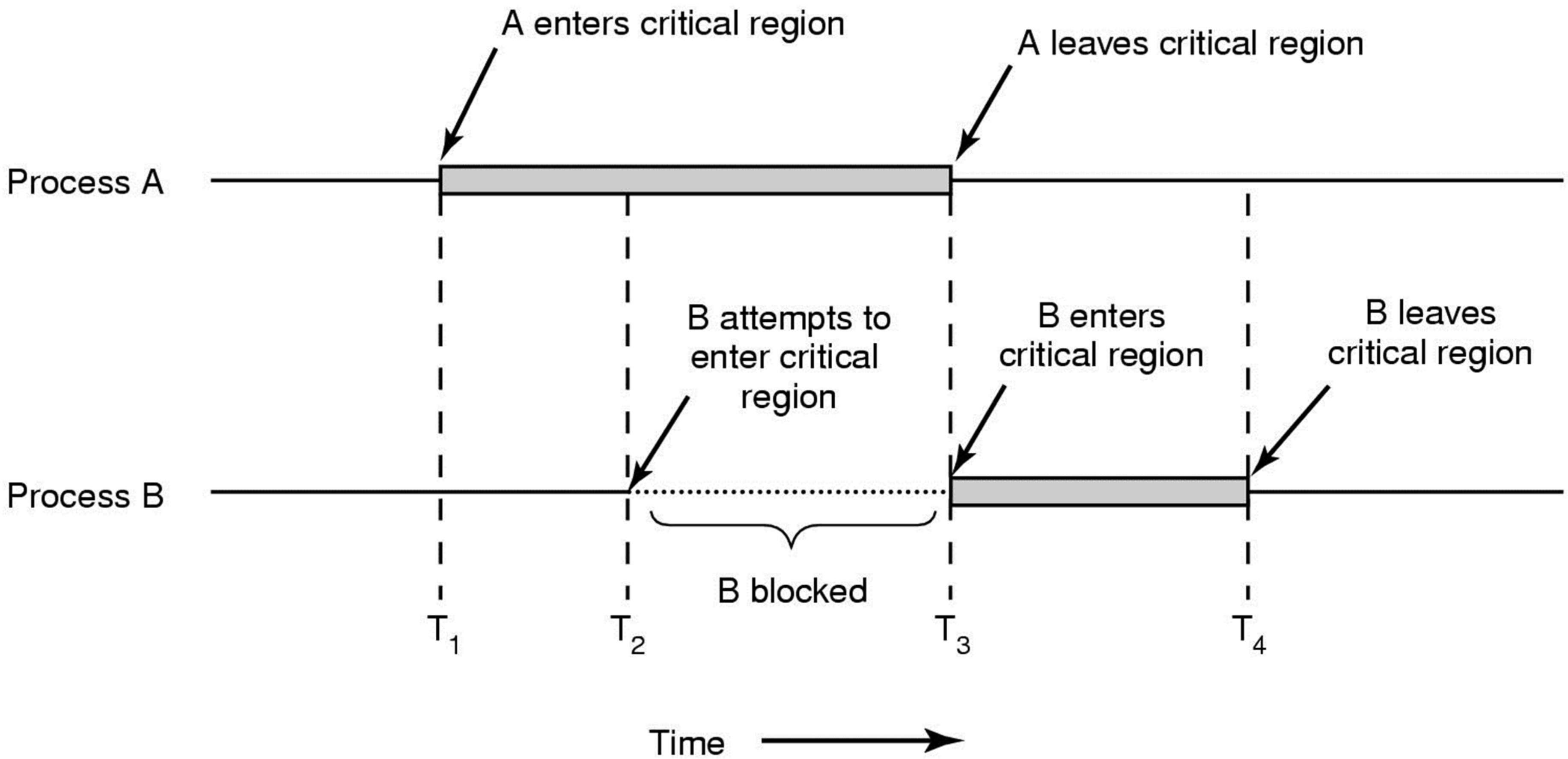
- **To prevent race conditions, we must prevent two processes from reading/writing shared resources at the same time.**
- **This is known as a "mutual exclusion", shortened to "mutex".**
- **Conceptually, a RUNNING process is always in one of two possible states:**
  - It is performing local computation. This does not involve global storage, hence to race condition is possible.
  - It is reading/updating global variables. This can lead to race conditions.
- **When a RUNNING process is in the second state, it is within its "critical section".**

Critical Sections

- **To prevent race conditions, 4 rules must be followed:**
  - No two processes can simultaneously be in their critical section.
  - No assumptions may be made about speeds or # of CPUs.
    - ✓ **Note: We can relax this assumption for *most* embedded systems since they have single CPUs.**
    - ✓ **May apply to systems using multicore microcontrollers.**
  - No process outside of its critical section can block other processes.
  - No process should wait forever to enter its critical section.

Critical Sections

- **In an ideal state, this is how mutual exclusion works:**



Inter-Task Communications

IMPLEMENTING MUTUAL EXCLUSION

Implementing Mutual Exclusion

- **There are several ways of implementing mutexes, each with their own + and – points:**
  - Disabling interrupts.
  - Lock variables.
  - Strict alternation.
  - Perterson's Solution.
  - Test and set lock.
  - Sleep/Wakeup

Implementing Mutual Exclusion Disabling Interrupts

- **Disabling Interrupts.**
  - This works because:
    - ✓**Time-slicing depends on a timer interrupt. If this is disable, the scheduler is never activated to switch to another process.**
    - ✓**Similarly, processes that are blocked pending an event (e.g. arrival of data from the network), depend on an interrupt to tell the scheduler that the event has taken place.**
  - Therefore disabling interrupts will prevent other processes from starting up and entering their critical sections.

NUS National University of Singapore School of Computing

Implementing Mutual Exclusion Disabling Interrupts

- **Disabling Interrupts.**
  - There are several problems with this approach:
    - ✓**Carelessly disabling interrupts can cause the entire system to grind to a halt.**
    - ✓**This only works on single-processor, single core systems. Violates Rule 2.**

Implementing Mutual Exclusion
Lock Variables

- **Using Lock Variables.**
  - A single global variable “lock” is initially 1.
  - Process A reads this variable and sets it to 0, and enters its critical section.
  - Process B reads “lock” and sees it’s a 0. It doesn’t enter its critical section and waits until “lock” is 1.
  - Process A finishes and sets “lock” to 1, allowing B to enter.

Implementing Mutual Exclusion
Lock Variables

- **This approach obviously doesn't work!!**
  - Process A reads in "lock" and sees a "1". It gets pre-empted and Process B runs.
  - Process B reads in "lock", sees a "1", sets it to 0 and enters its critical section.
  - Before B leaves, A is re-started, and enters the critical section.
- **Now >1 process is in the critical section!**
- **PROBLEM: There's a race condition on "lock" itself!**

Implementing Mutual Exclusion
Strict Alternation

```
while (TRUE) {
    while (turn != 0)       /* loop */ ;
    critical_region( );
    turn = 1;
    noncritical_region( );
}
```

(a)

```
while (TRUE) {
    while (turn != 1)       /* loop */ ;
    critical_region( );
    turn = 0;
    noncritical_region( );
}
```

(b)

- **A "turn" variable keeps track of whose turn it is to enter the critical section.**
- **If it is currently "0", then A will enter and B will continuously test until it becomes "1".**
- **Once A finishes its critical section, it flips "turn" to 1 allowing B to enter the critical region. B flips it back to 0 when done.**

Implementing Mutual Exclusion Strict Alternation

- **Problems (assume “turn”=0):**
  - Since “turn” is 0, B just loops infinitely waiting for “turn” to be 1.
    - ✓**This is called “busy-waiting” and burns valuable CPU time for no reason.**
  - If “A” is spending a lot of time in its non-critical section, it will not reach the part of the loop where it flips “turn” to 1.
  - Result is that B cannot enter it’s critical section even though no one is there!
- **Violates Rule #3.**

Implementing Mutual Exclusion Peterson's Solution

```
#define FALSE  0
#define TRUE   1
#define N      2                      /* number of processes */

int turn;                             /* whose turn is it? */
int interested[N];                    /* all values initially 0 (FALSE) */

void enter_region(int process);       /* process is 0 or 1 */
{
      int other;                      /* number of the other process */

      other = 1 – process;            /* the opposite of process */
      interested[process] = TRUE;     /* show that you are interested */
      turn = process;                 /* set flag */
      while (turn == process && interested[other] == TRUE) /* null statement */ ;
}

void leave_region(int process)        /* process: who is leaving */
{
      interested[process] = FALSE;    /* indicate departure from critical region */
}
```

Implementing Mutual Exclusion Peterson's Solution

- **Suppose Process A (i.e. process 0) wants to enter it's critical section.**
  - It calls enter_region.
    - ✓ **"other" is set to 1, "turn" is set to 0.**
    - ✓ **interest[0] is set to 1.**
    - ✓ **Since "interest[other]=interest[1]=0", the while loop exits immediately and A enters the critical section.**
  - If Process B (process 1) wants to enter and calls "enter region":
    - ✓ **"other is set to 0, "turn" is set to 1, interest[1] is set to 1.**
    - ✓ **It checks "interest[other]=interest[0]=1", it loops until interest[0] is set to 0.**
  - When process A ends, it calls leave_region which sets interest[0] to 0 and allows Process B to exit the while loop.

Test and Set Lock

- **Many microprocessors have an instruction that looks like this:**
  - TSL reg, lock ; lock is a variable in memory
- **How this works:**
  - CPU locks the address and data buses, and reads “lock” from memory.
    - ✓**The locked address and data buses will block accesses from all other CPUs.**
  - The current value is written into register “reg”.
  - A “1” (or sometimes “0”) value is written to “lock”.
  - CPU unlocks the address and data buses.
- **The TSL is “atomic”.**
  - This means that NOTHING can interrupt execution of this instruction.
  - This is guaranteed in hardware.

Test and Set Lock

- **The TSL instruction is used as follows:**

```
enter_region:
    TSL REGISTER,LOCK          | copy lock to register and set lock to 1
    CMP REGISTER,#0            | was lock zero?
    JNE enter_region           | if it was non zero, lock was set, so loop
    RET | return to caller; critical region entered


leave_region:
    MOVE LOCK,#0               | store a 0 in lock
    RET | return to caller
```

Test and Set Lock

- **An alternative is the XCHG instruction, used on Intel machines.**
  - Swaps contents of “lock” and “reg” instead of just writing “1” to lock.

```
enter region:
        MOVE REGISTER, #1          ; Set REGISTER to 1
        XCHG REGISTER, LOCK        ; Exchange with Lock
        CMP REGISTER, #0           ; Was Lock 0?
        JNE enter_region           ; No, go back and try again.
        RET                        ; Yes, enter critical region.

leave_region:
        MOVE LOCK, #0              ; Clear the lock
        RET                        ; And exit
```

Deadlock

- **Busy-wait approaches like Peterson and TSL/XCHG have a problem called "deadlock".**
- **Consider two processes H and L, and a scheduler rule that says that H is always run when it is READY. Suppose L is currently in the critical region.**
  - H becomes ready, and L is pre-empted.
  - H tries to obtain a lock, but cannot because L is in the critical region.
  - H loops forever, and CPU control never gets handed to L.
  - As a result L never releases the lock.
- **Note: The book calls this a "priority inversion", which is incorrect.**

Sleep/Wake

- **One solution to this problem is through the use of "Sleep/Wake" functions.**
  - When a process finds that a lock has been set (i.e. another process in the critical section), it calls "sleep" and is put into the blocked state.
  - When the other process exits the critical section and clears the lock, it can call "wake" which moves the blocked process into the READY queue for eventual execution.
- **While this sounds like an ideal solution, it can create a problem called the "producer-consumer problem".**

Inter-Task Communications

THE PRODUCER/CONSUMER PROBLEM

NUS National University of Singapore School of Computing

The Producer/Consumer problem

```
#define N 100                               /* number of slots in the buffer */
int count = 0;                              /* number of items in the buffer */

void producer(void)
{
      int item;

      while (TRUE) {                        /* repeat forever */
            item = produce_item( );         /* generate next item */
            if (count == N) sleep( );       /* if buffer is full, go to sleep */
            insert_item(item);              /* put item in buffer */
            count = count + 1;              /* increment count of items in buffer */
            if (count == 1) wakeup(consumer);  /* was buffer empty? */
      }
}


void consumer(void)
{
      int item;

      while (TRUE) {                        /* repeat forever */
            if (count == 0) sleep( );       /* if buffer is empty, got to sleep */
            item = remove_item( );          /* take item out of buffer */
            count = count – 1;              /* decrement count of items in buffer */
            if (count == N – 1) wakeup(producer);  /* was buffer full? */
            consume_item(item);             /* print item */
      }
}
```

The Producer/Consumer problem

- **Producer and consumer share a fixed-size buffer.**
  - A global variable "count" keeps track of the number of items.
    - ✓ **If count==N (FULL), producer sleeps, if count==0 (EMPTY) consumer sleeps.**
  - After reading from the buffer, if count==N-1:
    - ✓ **Consumer reasons that the buffer was earlier full and wakes the producer.**
  - After writing to the buffer, if count==1
    - ✓ **Producer reasons that the buffer was earlier empty and wakes the consumer.**

The Producer/Consumer problem

- **Deadlock occurs when:**
  - Consumer checks “count” and finds it is 0.
  - Consumer gets pre-empted and producer starts up.
  - Producer adds an item, increments count to “1”, then sends a WAKE to the consumer.
    - ✓**Since consumer is not technically sleeping yet, the WAKE is lost.**
  - Consumer starts up, and since count is 0, goes to SLEEP.
  - Producer starts up, fills buffer until it is full and SLEEPs.
- **Since consumer is also SLEEPing, no one wakes the producer. Deadlock.**

Inter-Task Communications

SEMAPHORES

Semaphores

- **A semaphore is a special lock variable that counts the number of wake-ups saved for future use.**
  - A value of "0" indicates that no wake-ups have been saved.
- **Two ATOMIC operations on semaphores:**
  - DOWN, PEND or P:
    - ✓**If the semaphore has a value of >0, it is decremented and the DOWN operation returns.**
    - ✓**If the semaphore is 0, the DOWN operation blocks.**
  - UP, POST or V:
    - ✓**If there are any processes blocking on a DOWN, one is selected and woken up.**
    - ✓**Otherwise UP increments the semaphore and returns.**

Using Semaphores in the Producer/Consumer Problem

```
#define N 100                          /* number of slots in the buffer */
typedef int semaphore;                 /* semaphores are a special kind of int */
semaphore mutex = 1;                   /* controls access to critical region */
semaphore empty = N;                   /* counts empty buffer slots */
semaphore full = 0;                    /* counts full buffer slots */

void producer(void)
{
     int item;

     while (TRUE) {                    /* TRUE is the constant 1 */
          item = produce_item( );      /* generate something to put in buffer */
          down(&empty);                /* decrement empty count */
          down(&mutex);                /* enter critical region */
          insert_item(item);           /* put new item in buffer */
          up(&mutex);                  /* leave critical region */
          up(&full);                   /* increment count of full slots */
     }
}


void consumer(void)
{
     int item;

     while (TRUE) {                    /* infinite loop */
          down(&full);                 /* decrement full count */
          down(&mutex);                /* enter critical region */
          item = remove_item( );       /* take item from buffer */
          up(&mutex);                  /* leave critical region */
          up(&empty);                  /* increment count of empty slots */
          consume_item(item);          /* do something with the item */
     }
}
```

- EMPTY - # of empty slots.
- FULL - # of full slots.
- MUTEX – Prevents simultaneous access to the buffer.

Mutual Exclusion with Semaphores

- **When a semaphore's counting ability is not needed, we can use a simplified version called a "mutex".**
  - 1 = Unlocked.
  - 0 = Locked.
- **Two processes can then attempt do DOWN the semaphore.**
  - Only one will succeed. The other will block.
  - When the successful process exits the critical section, it does an UP, waking the other process up.

Mutual Exclusion with Semaphores

Process A

```
sema=1
…
non_critical_section()
DOWN(sema)
critical_section()
UP(sema)
…
```

Process B

```
non_critical_section()
DOWN(sema)
critical_section()
UP(sema)
…
```

Mutual Exclusion with TSL/XCHG

- **We can also implement mutexes with TSL or XCHG.**
  - 0 = Unlocked, 1 = Locked.

```
mutex_lock:
      TSL REGISTER,MUTEX           | copy mutex to register and set mutex to 1
      CMP REGISTER,#0              | was mutex zero?
      JZE ok                       | if it was zero, mutex was unlocked, so return
      CALL thread_yield            | mutex is busy; schedule another thread
      JMP mutex_lock               | try again later
ok:   RET | return to caller; critical region entered


mutex_unlock:
      MOVE MUTEX,#0                | store a 0 in mutex
      RET | return to caller
```

NUS National University of Singapore
School of Computing

Problems with Semaphores

Deadlock

- **Our producer/consumer solution has a problem:**
  - If we swapped the semaphores for empty/full with the mutex semaphore, we have a potential deadlock:

```
#define N 100                           /* number of slots in the buffer */
typedef int semaphore;                  /* semaphores are a special kind of int */
semaphore mutex = 1;                    /* controls access to critical region */
semaphore empty = N;                    /* counts empty buffer slots */
semaphore full = 0;                     /* counts full buffer slots */

void producer(void)
{
     int item;

     while (TRUE) {                     /* TRUE is the constant 1 */
          item = produce_item( );       /* generate something to put in buffer */
          down(&mutex);                 /* decrement empty count */
          down(&empty);                 /* enter critical region */
          insert_item(item);            /* put new item in buffer */
          up(&mutex);                   /* leave critical region */
          up(&full);                    /* increment count of full slots */
     }
}

void consumer(void)
{
     int item;

     while (TRUE) {                     /* infinite loop */
          down(&mutex);                 /* decrement full count */
          down(&full);                  /* enter critical region */
          item = remove_item( );        /* take item from buffer */
          up(&mutex);                   /* leave critical region */
          up(&empty);                   /* increment count of empty slots */
          consume_item(item);           /* do something with the item */
     }
}
```

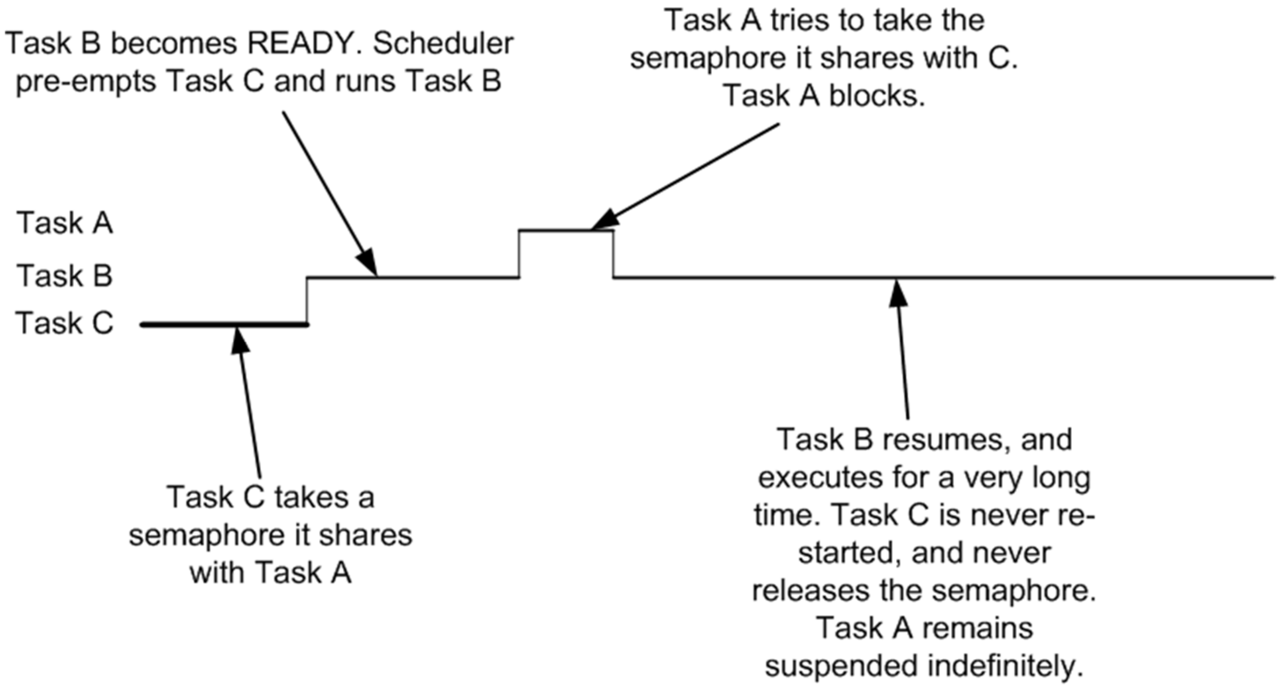
NUS National University of Singapore School of Computing

Problems with Semaphores Deadlock

- **This can happen:**
  - Producer successfully DOWNs the mutex.
  - Producer DOWNs “empty”. However the queue is full so this blocks.
  - Consumer DOWNs mutex and blocks.
    - ✓ **Consumer now never reaches the UP for “empty” and therefore cannot unblock the producer.**
    - ✓ **The producer in turn never reaches the UP for mutex and cannot unblock the consumer.**
    - ✓ **Deadlock!**

Problems with Semaphores Priority Inversion

- **In the diagram on the following page, priority(Task C) < priority(Task B) < priority(Task A).**



- **Task B effectively blocks out Task A, although Task A has higher priority!**

Inter-Task Communications

MONITORS

NUS National University of Singapore School of Computing

Monitors

- **A monitor is similar to a class or abstract-data type in C++ or JAVA:**
  - Collection of procedures, variables and data structures grouped together in a package.
    - ✓**Access to variables and data possible only through methods defined in the monitor.**
  - However, only one process can be active in a monitor at any point in time.
    - ✓**I.e. if any other process tries to call a method within the monitor, it will block until the other process has exited the monitor.**

Monitors

- **Implementation:**
  - When a process calls a monitor method, the method first checks to see if any other process is already using it.
  - If so, the calling process blocks until the other process has exited the monitor.
    - ✓**This can be achieved using mutexes or binary semaphores.**
    - ✓**The mutex/semaphore operations are inserted by the compiler itself rather than by the user, reducing the likelihood of errors.**

Monitors and Condition Variables

- **Monitors achieve mutual exclusion, but we also need other mechanisms for coordination.**
  - E.g. in our producer/consumer problem, mutual exclusion alone is not enough to prevent the producer from proceeding when the buffer is full.
- **We introduce “condition variables”.**
  - One process WAITs on a condition variable and blocks, until..
  - Another process SIGNALs on the same condition variable, unblocking the WAITing process.

Monitors and Condition Variables

- **Implementing the Producer/Consumer problem with semaphores and condition variables:**
  - When the buffer is full (count==N), producer will WAIT on a full condition.
  - When buffer is empty (count==0), consumer will WAIT on empty.

```
monitor ProducerConsumer
    condition full, empty;
    integer count;
    procedure insert(item: integer);
    begin
        if count = N then wait(full);
        insert_item(item);
        count := count + 1;
        if count = 1 then signal(empty)
    end;
    function remove: integer;
    begin
        if count = 0 then wait(empty);
        remove = remove_item;
        count := count − 1;
        if count = N − 1 then signal(full)
    end;
    count := 0;
end monitor;

procedure producer;
begin
    while true do
    begin
        item = produce_item;
        ProducerConsumer.insert(item)
    end
end;
procedure consumer;
begin
    while true do
    begin
        item = ProducerConsumer.remove;
        consume_item(item)
    end
end;
```

Monitors and Condition Variables

- **When a process encounters a WAIT, it is blocked and another process is allowed to enter the monitor.**
- **Problem:**
  - When there's a SIGNAL, the sleeping process is woken up.
  - We will potentially now have two processes in the monitor at the same time:
    - ✓ **The process doing the SIGNAL (the signaler).**
    - ✓ **The process that just woke up because of the SIGNAL (the signaled).**

Monitors and Condition Variables

- **We have 3 ways to resolve this:**
  - We require that the signaler exits immediately after calling SIGNAL.
  - We suspend the signaler immediately and resume the signaled process.
  - We suspend the signaled process until the signaler exits, and resume the signaled process only after that.

Monitors and Condition Variables

- **A condition variable is different from a semaphore.**
  - Semaphore:
    - ✓**If Process A UPs a semaphore with no pending DOWN, the UP is saved.**
    - ✓**The next DOWN operation will not block because it will match immediately with a preceding UP.**
  - Condition variable:
    - ✓**If Process A SIGNALs a condition variable with no pending WAIT, the SIGNAL is simply lost.**
    - ✓**This is similar to the SLEEP/WAKE problem earlier on.**

Monitors and Condition Variables

- **This code looks suspiciously like our original producer/consumer problem on Page 23!**
  - Same issues too:
    - ✓**Page 23:**

      Consumer sees count==0, and intends to SLEEP but gets pre-empted.

      Producer sends a WAKE but the WAKE is lost.

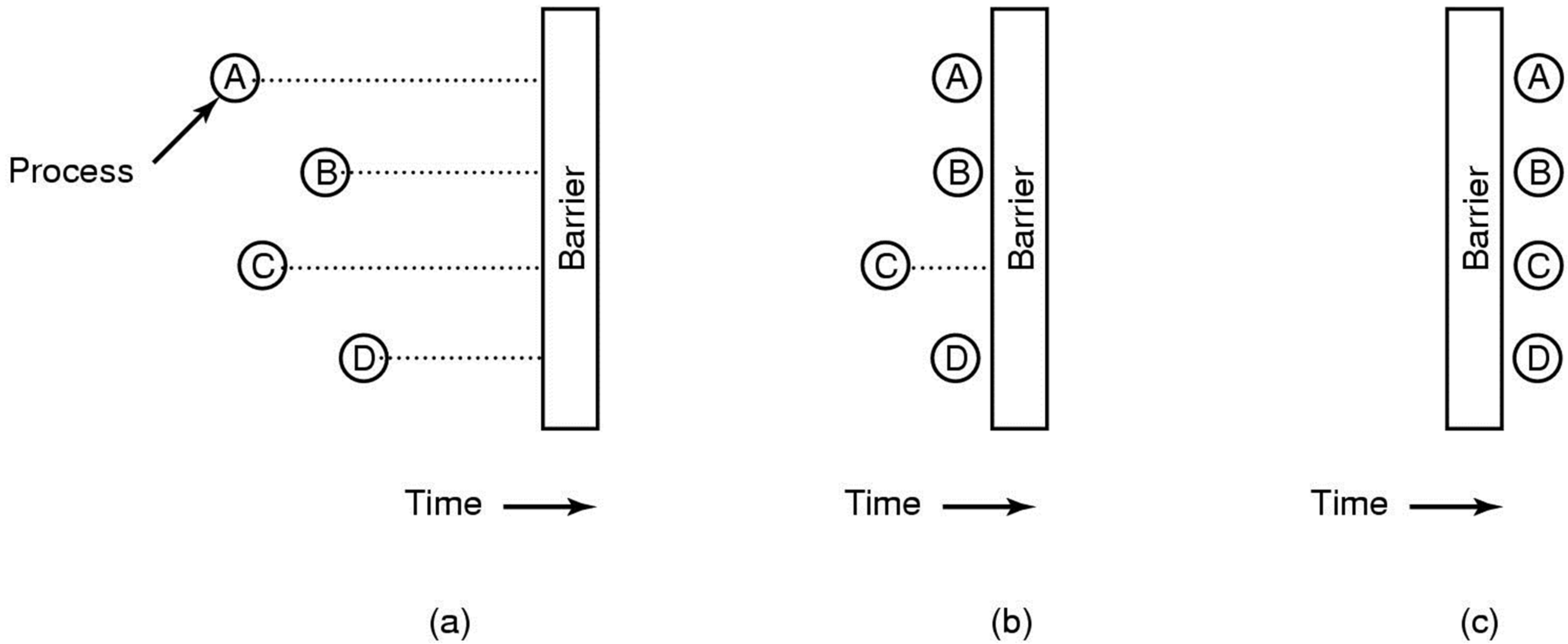
      In this case, if the consumer gets pre-empted before a WAIT, the corresponding SIGNAL from the producer is also lost!
    - ✓**However we see that in this case, the mutual exclusion from the monitor prevents the SIGNAL from being lost! (WHY?)**

Inter-Task Communications

BARRIERS

Barriers

- **A "barrier" is a special form of synchronization mechanism that works with groups of processes rather than single processes.**



Barriers

- **The idea of a barrier is that all processes must reach the barrier (signifying the end of one phase of computation) before any of them are allowed to proceed.**
    - Process D reaches the end of the current phase and calls a BARRIER primitive in the OS. It gets blocked.
    - Similarly processes A and B reach the end of the current phase, calls the same BARRIER primitive and is blocked.
    - Finally process C reaches the end of its computation, calls the BARRIER primitive, causing all processes to be unblocked at the same time.

Barriers

- **Example barrier application.**
  - Computing fluid motion across a surface represented by a 1,000,000 x 1,000,000 matrix.
    - ✓**The complete matrix for iteration *N* must be available before computing for iteration *N+1*.**
    - ✓**1,000 individual processes each computing a part of the entire matrix.**
    - ✓**We must wait for all 1,000 processes to complete finding the entire matrix before we can start on the next iteration.**
  - A barrier would be very useful in achieving this.

Inter-Task Communications

COMMUNICATION MECHANISMS

Other Communication Mechanisms

- **The mechanisms we've looked at are largely for coordination.**
  - Locks, semaphores, monitors, etc.
  - Any communication takes place through global variables.
- **We will now look at several mechanisms that combine coordination with communication.**
  - Can help eliminate global variables and reduce errors.
- **Mechanisms we will look at:**
  - Queues
  - Mailboxes
  - Pipes

Inter-Task Communications

QUEUES

Queues

- **Queues have four operations:**
  - q = initq(ptr, size): Initializes the queue of size “size” at memory location “ptr”. Returns the queue identifier “q”.
  - enq(q, data): Adds a new item “data” to the tail of the queue “q”.
    - ✓**Returns an error message when the queue is full.**

Queues

- data = deq(q, timeout): Reads and removes an item from the head of the queue and returns it.
  - ✓**Blocks if the queue is empty, up to a specific time specified by “timeout”.**
  - ✓**If timeout is 0, deq blocks forever until there is a message in the queue.**
- destroy(q): Destroys a queue. A queue, once destroyed, cannot be used unless you call initq again.

Queues

- **The RTOS guarantees the correctness of the queue operations.**
  - If a task gets pre-empted in the middle of an enq or deq operation, the RTOS ensures that the operation is completed correct.
  - E.g. if the second task calls enq to the same queue, the RTOS can block it until the first task completes enq properly.

Example

- **Consider a case with two tasks Task1 and Task2.**
  - Both are high priority, with urgent things to attend to.
- **If they encounter an error, they need to report this error across a network.**

Example

- **Writing to a network is typically time consuming:**
  - Hundreds of instructions to transfer data to buffers, wait for acknowledgement etc.
  - Latencies in waiting to access the network medium.
- **Makes sense to partition this into another lower priority task.**

Example

- **Task1 and Task2 must therefore share data with this error reporting task *ErrorsTask*.**
- **For our example we will assume that the queue is properly initialized.**

Queue Example

- **Task Codes**

```
void Task1(void)
{
  ……
  if(!!problem arises)
      vLogError(ERROR_TYPE_X);
  ……
}

Void Task2(void)
{
  ……
  if(!!problem arises)
      vLogError(ERROR_TYPE_Y);
  ……
}
```

Queue Example

```
void vLogError(int iErrorType)
{
  enq(q, iErrorType);
}
static int cErrors;
void ErrorsTask(int iErrorType)
{
  int iErrorType = deq(q, 0);
  ++cErrors;
  !! send cErrors and iErrorType across network.
}
```

Inter-Task Communications

MAILBOXES

Mailboxes

- **A mailbox is like a queue, except:**
  - The number of mailboxes within a system is typically fixed.
    - ✓**This number is initialized when your application first starts.**
    - ✓**Not possible to add more mailboxes.**
    - ✓**Unlike queues, which you can create as and when you need them.**

NUS National University of Singapore
School of Computing

Mailboxes

- A mailbox message is also often prioritized.
  - ✓**In a queue, the data is read in the order that it is written.**
  - ✓**Higher priority messages in a mailbox are read ahead of lower priority messages, regardless of the order of writing.**

Mailbox Operations

- **Several operations are defined on mailboxes.**
  - initmbox(num): Initializes “num” mailboxes in the system. Can only be done once.
  - sendmsg(mbID, message, priority): Uses mailbox “mbID” to send a message at the given priority.
  - rcvmsg(mbID): Gets the highest priority message from mbID. Blocks if there are no messages.
  - chkmsg(mbID): Similar to rcvmsg, but returns NULL if the mailbox is empty instead of blocking.

Inter-Task Communications

PIPES

Pipes

- **A pipe is like a queue:**
  - Not prioritized. Messages are read in the order that they are written.
  - Can be created and deleted as and when required.
- **However a pipe is generally byte-oriented.**
  - A queue, for example, reads/writes an entire integer, which may consist of between 2 and 8 bytes.
  - A pipe reads/writes in units of 1 byte.
- **This can give more flexible message passing.**

Pipes

- **Pipe Operations:**
  - Standard C *fread* and *fwrite* functions.
- **Pipes are usually ready-supported by desktop operating systems.**
  - Using pipes allows your programs to be ported to desktop OSes that are adapted for RTOS use.
    - ✓**E.g. BlueCat, a Linux derivative.**

Inter-Task Communications

SELECTING MECHANISMS

Choosing Between Message Passing Methods

- **Each method has its advantages and drawbacks:**
  - Semaphores:

    GOOD: Fast, lightweight.

    BAD: Can be difficult to use correctly.

    For example, you may find yourself having to set up multiple semaphores: One to coordinate between processes, another to protect data updates.

    A semaphore that is taken too late or released too early can be worse than not having a semaphore at all.

Choosing Between Message Passing Methods

- Queues:
  - **GOOD: Simple to use.**
    - *Initialize, enqueue data, dequeue data.*
    - *All task coordination is achieved through the queue.*
    - *RTOS guarantees queue data integrity.*
  - **BAD: No message priorities, heavier than semaphores.**
    - *Need to manage queue "head" and "tail" pointers.*
    - *Need to guarantee correctness of enq and deq operations.*

Choosing Between Message Passing Methods

- Mailboxes:

  GOOD: Intuitively appealing.

  Each task has its own mailbox to pass it messages.

  GOOD: Prioritized messaging.

  Tasks can be read by priority instead of FIFO.

  BAD: Even heavier than queues.

  Extra code to ensure that maxilbox messages are sorted by priority.

  Can be very expensive for large mailboxes.

NUS National University of Singapore
School of Computing

Choosing Between Message Passing Methods

- Pipes
  - ✓**GOOD: Byte level mechanisms. Very flexible for passing data.**
  - ✓**GOOD: Supported by almost all desktop OSes**

    Allows you to port your software to such OSes that have been adapted to RTOS.
  - ✓**BAD: Byte-level operations are often more error prone.**

Choosing Between Message Passing Methods

- Pipes
  - ✓**BAD: Byte-level operations are slow.**

    CPUs transfer in fixed numbers of bytes called the "word size".

    Byte level operations would mean chopping up and joining words.
  - ✓**BAD: No priorities.**

School of Computing

Choosing Between Message Passing Methods

- **The RTOS manuals will list the sizes and expected execution times for each mechanism.**
- **It is critical that you choose the best mechanism for your application.**
  - Do you need maintainability, priority, execution speed or small footprint?
- **RTOSs sometimes allow you to remove mechanisms that you don't use.**
  - Might think about sticking with just one mechanism.

Inter-Task Communications

POTENTIAL PITFALLS

Potential Pitfalls

- **While queues, mailboxes and pipes simplify data sharing, they also make it easy to insert bugs into your system**
- **For readability, the pitfalls listed talk about queues. Unless otherwise stated, they apply to mailboxes and pipes as well.**

Potential Pitfalls

1. **Reading/Writing to the wrong queue.**
   - RTOSs don't have the ability to restrict a queue to a particular task.
   - If one task writes temperature data to the error messages queue, then the error handling task will get confused.
2. **Wrong Queue Semantics**
   - Task A writes integers to a queue.
   - Task B reads the queue and interprets the data as a pointer to an integer, instead of the integer itself.

Potential Pitfalls

```
OSEVENT *pq;
// Prototype for the enq and deq functions
void enq(OSEVENT *q, void *data);
void *deq(OSEVENT *q);

void taskA(void)
{
    int datum;
    … some processing on datum …
    // Enqueue an integer “datum”, casting it as
    // void * to fit the enq prototype.
    enq(pq, (void *) datum);
}
```

Potential Pitfalls

```
void taskB(void)
{
    int *dataptr;

    dataptr = (int *) deq(pq);
    *dataptr = !! Result of some computation.
}
```

- This code is a disaster because taskB is going to (almost) randomly overwrite memory that is used by other tasks.
    - Remember that there is no memory protection in most RTOS!

Potential Pitfalls

3. Running out of space.
   - This is a disaster because the data being passed is not usually optional.
4. Inadvertently creating unprotected shared variables.
   - This happens when you pass pointers instead of data.
   - If you pass data, the RTOS ensures that the enq and deq operations place and remove the data correctly from the queue.
   - If you pass pointers, the RTOS ensures that the enq and deq operations place and remove the pointers correctly from the queue.
     - BUT: The data the pointers point to are themselves not protected!

Potential Pitfalls

```
OSEVENT *pq;
// Prototype for the enq and deq functions
void enq(OSEVENT *q, void *data);
void *deq(OSEVENT *q);

void vReadTemperaturesTask(void)
{
    static int iTemperatures[2];
    while(1)
    {
      iTemperatures[0] = !! Read in temp1 from hw
      iTemperatures[1] = !! Read in temp2 from hw

      // Add address of iTemperatures to queue
      enq(pq, (void *) iTemperatures);
    }
```

Potential Pitfalls

```
void vMainTask(void)
{
    int *pTemp;

    while(1)
    {
      pTemp = (int *) deq(pq);
      if(pTemp[0] != pTemp[1])
            !! Set of howling alarm
    }
}
```

Potential Pitfalls

i. *vMainTask* will get the address of *iTemperatures* from the queue, assigning it to *pTemp*.

ii. It then reads *pTemp[0]* (which is *iTemperatures[0]* since *pTemp* points to the same *iTemperatures* array).

iii. *vMainTask* gets pre-empted, and *vReadTemperaturesTask* updates *iTemperatures[0]* and *iTemperatures[1]*.

Potential Pitfalls

iv. *vMainTask* resumes, and reads in *pTemp[1]*.

v. The comparison will fail, causing the alarm to be triggered errorneously.

Summary

- **In this lecture we looked at:**
  - What race conditions and critical sections are.
  - Mechanisms to prevent race conditions.
  - Mechanisms that provide both coordination and communication functions.