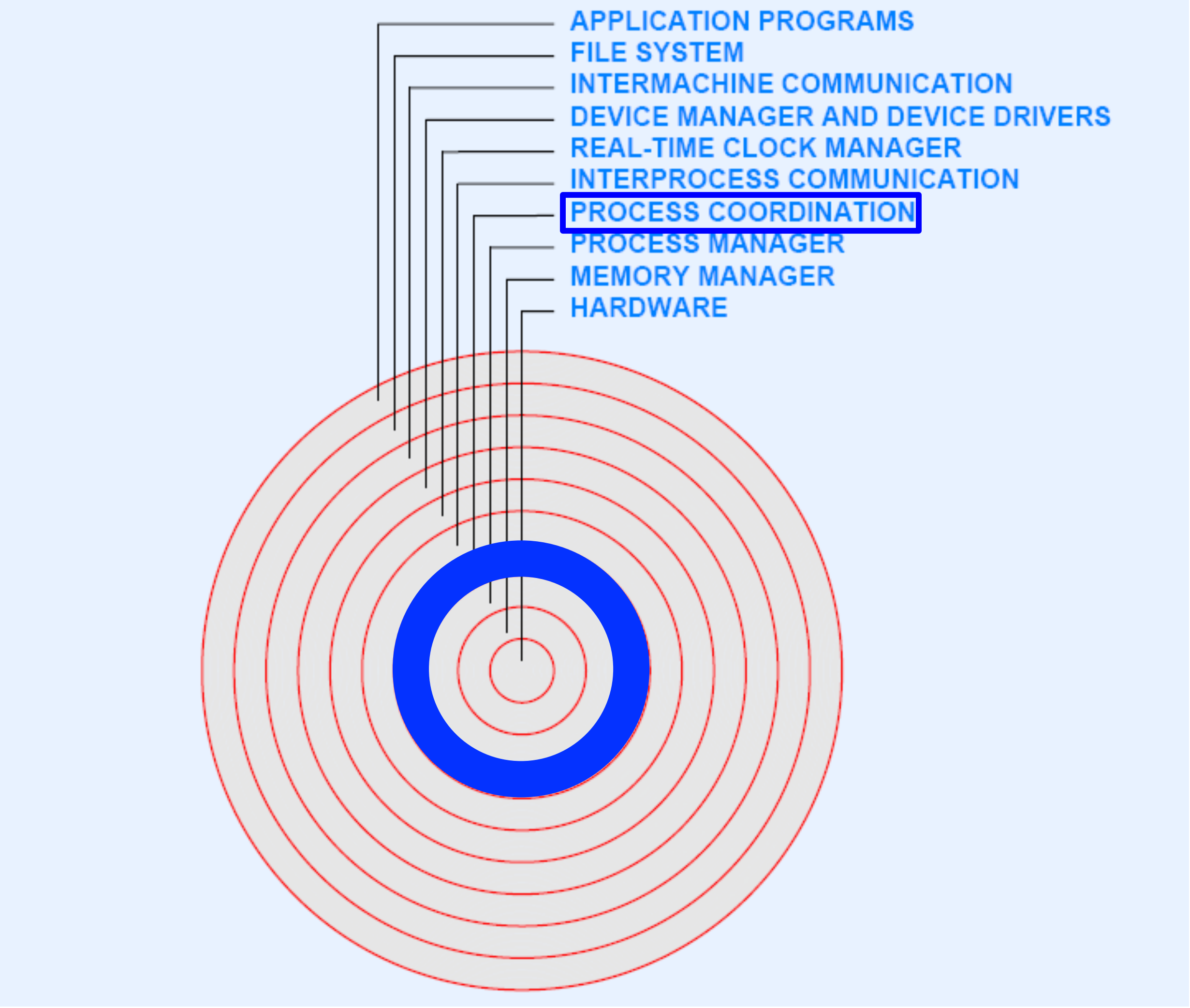
CSCI 8530
Advanced Operating Systems

Part 5

Process Coordination and Synchronization

Location of Process Coordination in the Hierarchy



Coordination of Processes

- Necessary in a concurrent system
- Avoids conflicts when accessing shared items
- Allows multiple processes to cooperate
- Can also be used when
  – Process waits for I/O
  – Process waits for another process
- Example of cooperation among processes: UNIX pipes

Two Approaches to Process Coordination

- Use hardware mechanisms: Disabling interrupts
  - Stop all activities except for one process
  - Most useful for multiprocessor systems
  - May rely on *busy waiting*
- Use operating system mechanisms
  - Works well with a single processor
  - No unnecessary execution
- Synchronization mechanisms are needed that
  - Allow a subset of processes to contend for access to a resource
  - Provide a policy that guarantees fair access

Note: we will mention hardware quickly, and focus on OS operating system functions

Key Situations that Process Coordination Mechanisms Handle

- Producer/consumer interaction
  - Each access to the buffer must be done in a critical section.
  - When the buffer is full, producers cannot place any more items in it. They therefore go to sleep, to be awakened when a consumer removes an item.
  - Consumers go to sleep when the buffer is empty, to be awaked by a consumer placing an item in the buffer.
- Mutual exclusion
  - This use is illustrated by the **`mutex`** semaphore in the producer-consumer solution. Some systems even provide a semaphore type with count limited to 0 or 1 and call them a mutex (e.g. Windows).

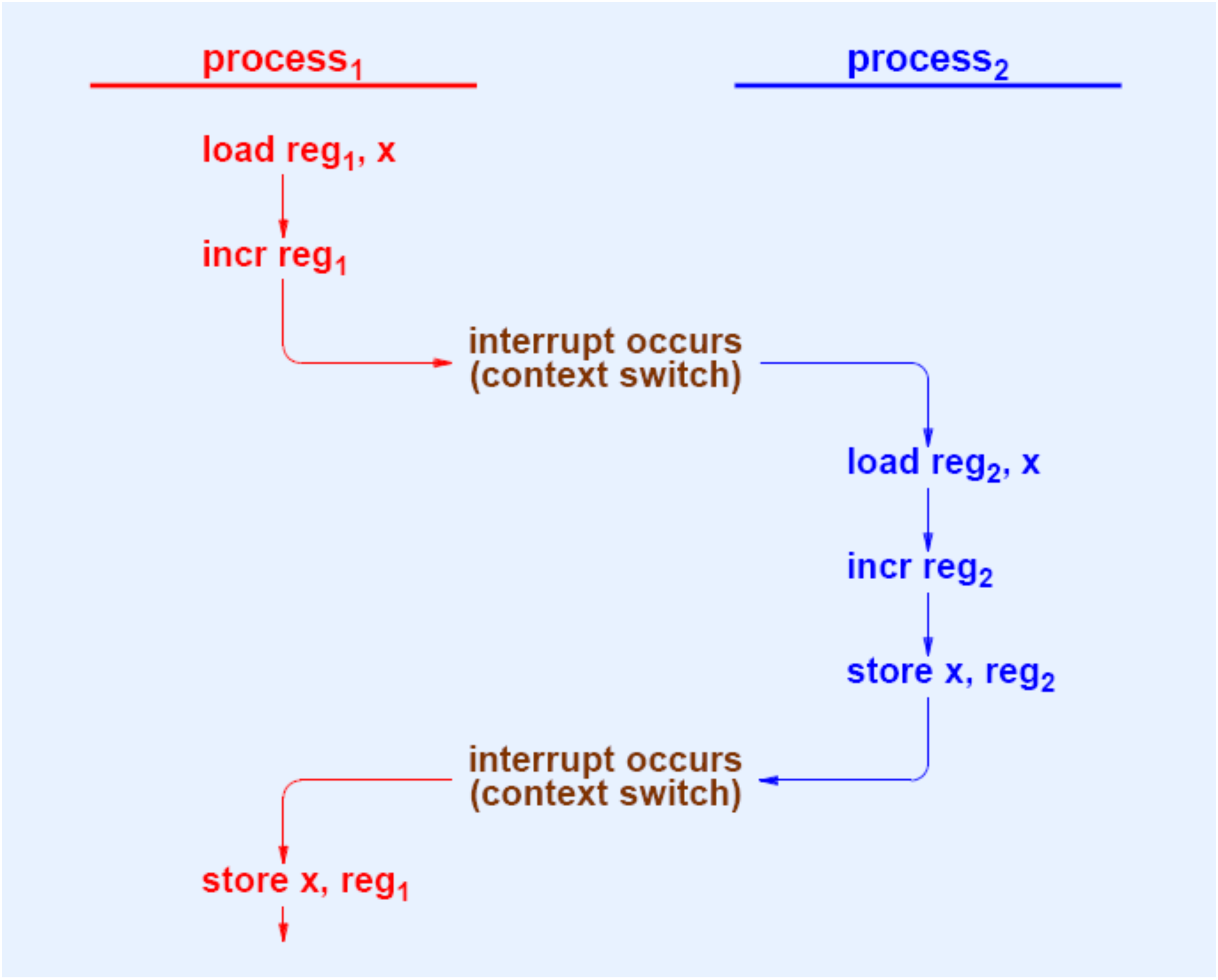
Producer-Consumer Synchronization

- Typical scenario
  - Shared circular buffer
  - Producing processes deposit items into buffer
  - Consuming processes extract items from buffer
- Must guarantee
  - A producer blocks when buffer full
  - A consumer blocks when buffer empty

Mutual Exclusion

- Concurrent processes access shared data
- Non-atomic operations can produce unexpected results
- Example: multiple steps used to increment variable *z*
  - Load variable *z* into register *i*
  - Increment register *i*
  - Store register *i* in variable *z*

Illustration of Two Processes Attempting to Increment a Shared Variable Concurrently



Single Producer/Consumer Using the Shared Variable

- `count++` could be implemented as

```
register1 = count
register1 = register1 + 1
count = register1
```

- `count--` could be implemented as

```
register2 = count
register2 = register2 - 1
count = register2
```

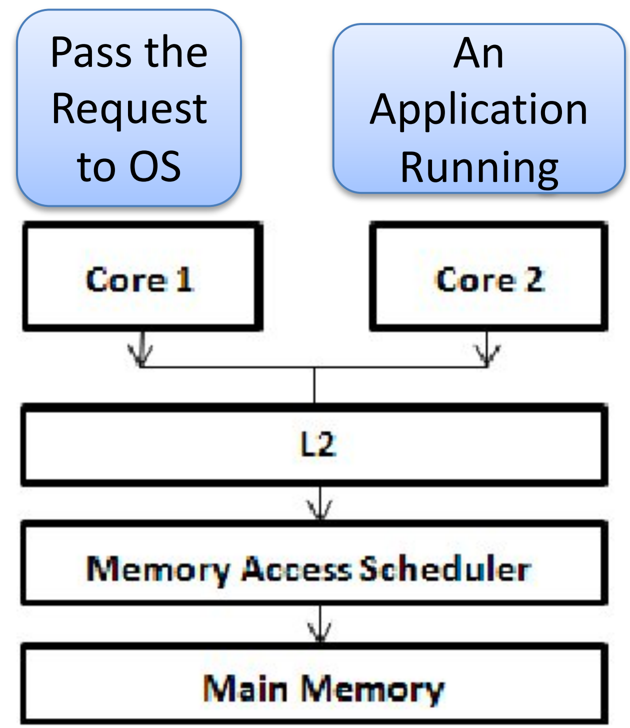
- Consider this execution interleaving with "count = 5" initially:

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

- The variable **count** is shared between the producer and consumer, and access to it is not synchronized.

To Prevent Problems

- Ensure that only one process accesses a shared item at any time
  – Not efficient for multicores
- Trick: once a process obtains access, make all other processes wait
- Three solutions
  1. Spin lock hardware instructions
  2. Disabling all interrupts
  3. Semaphores (implemented in software): *wait* and *signal*



Multicore architecture with memory access scheduler

Handling Mutual Exclusion with Spin Locks

- Used in multiprocessors; does not work for single processor
  - Defines a set of *K* spin locks (*K* might be less than 1024) → Each is a single bit
- A special instruction: *test-and-set*
- Atomic hardware operation tests a memory location and changes it
  - Sets the contents of a given address to one and returns the previous value.
- Also called *spin lock* because it involves *busy waiting*

Examples of a Spin Lock (x86)

- Instruction performs atomic compare and exchange

  [lock] cmpxchg reg, reg/mem

- Spin loop: repeat the following
  - Place a "unlocked" value (e.g, 0) in register *eax*
  - Place an "locked" value (e.g., 1) in register *ebx*
  - Place the address of a memory location in register *ecx* (the lock)
  - Execute the *cmpxchg* instruction
  - Register *eax* will contain the value of the lock before the compare and exchange occurred
  - Continue the spin loop as long as *eax* contains the "locked" value
- To release, assign the "unlocked" value to the lock

Example Spin Lock Code for x86 (part 1)

```
/* mutex.S - mutex_lock, mutex_unlock */

        .text
        .globl mutex_lock
        .globl mutex_unlock

/*------------------------------------------------------------------------
 * mutex_lock(uint32 *lock) -- Acquire a lock
 *------------------------------------------------------------------------
 */
mutex_lock:

        /* Save registers that will be modified */

        pushl %eax
        pushl %ebx
        pushl %ecx
```

Example Spin Lock Code for x86 (part 2)

```
spinloop:
        movl $0, %eax              /* Place the "unlocked" value in eax    */
        movl $1, %ebx              /* Place the "locked" value in ebx      */
        movl 16(%esp), %ecx        /* Place the address of the lock in ecx */

        lock cmpxchg %ebx, (%ecx)  /* Atomic compare-and-exchange:         */
                                   /* Compare ebx with memory (%ecx)       */
                                   /* if equal                             */
                                   /* load %ebx in memory (%ecx)           */
                                   /* else                                 */
                                   /* load %ebx in register (%eax)         */

        /* If eax is 1, the mutex was locked, so continue the spin loop    */

        cmp $1, %eax
        je spinloop

        /* We hold the lock now, so pop the saved registers and return     */
        popl %ecx
        popl %ebx
        popl %eax
        ret
```

cmpxchg can't work with an immediate operand

Example Spin Lock Code for x86 (part 3)

```
/*------------------------------------------------------------------------
 * mutex_unlock (uint32 *lock) - release a lock
 *------------------------------------------------------------------------
 */
mutex_unlock:

        /* Save register eax */
        pushl %eax

        /* Load the address of lock onto eax */
        movl 8(%esp), %eax

        /* Store the "unlocked" value in the lock, thereby unlocking it */
        movl $0, (%eax)

        /* Restore the saved register and return */
        popl %eax
        ret
```

Handling Mutual Exclusion with Semaphores

- Operating system guarantees only one process can access the shared item at a given time
  - Ex: Two executing processes each need to insert items into a shared linked list.
- Semaphore allocated for item to be protected
- Known as a *mutex* , or binary semaphore
- Applications must be programmed to use the mutex before accessing shared item
- Implementation avoids busy waiting

Definition of Critical Section

- Each piece of shared data must be protected from concurrent access
  – Create a single semaphore with an initial count of 1
- Programmer inserts mutex operations
  – Call *wait* before access to shared item
  – Call *signal* after access to shared item
- Protected code known as *critical section*
- Mutex operations can be placed in each function that accesses the shared item

```
/* ex6.c - additem */
```

Semaphores and Mutual Exclusion

```
/* ex6.c - additem */
#include <xinu.h>

sid32       mutex                         /* Assume initialized with semcreate */
Int32       shared[100];                  /* An array shared by many processes */
int32       n = 0;                        /* Count of items in the array */

/*------------------------------------------------------------------------
additem - Obtain exclusive use of array shared and add an item to it
*------------------------------------------------------------------------ */
void  additem(
            int32       item       /* Item to add to shared array */
            )
{
            wait(mutex);
            shared[n++] = item;
            signal(mutex);
}
```

- *mutex* will be assigned the ID of a semaphore before any calls to *additem* occur.

mutex = semcreate(1);

- calls *wait* on semaphore *mutex* before accessing the array.
- calls *signal* on the semaphore when access is complete.

At what level of granularity should mutual exclusion be applied in an operating system?

Coarse-grained synchronization: no matter how much native support for concurrency the hardware provides, only one thread at a time can execute

- Fault-tolerance: interrupts anytime while still holds a lock
- Speedup: do not need to consider # of physical processors supported by the machine

Low-level Mutual Exclusion

- Mutual exclusion needed
  - By application processes
  - Inside operating system
- Mutual exclusion can be guaranteed provided no context switching occurs because of shared data
- Context changed by
  - Interrupts
  - Calls to *resched*
- Low-level mutual exclusion:
  - Mask interrupts (hardware): control interrupts
  - Avoid rescheduling

Interrupt Mask

- Hardware mechanism that controls interrupt recognition
- Internal machine register; may be part of processor status word
- On some hardware, zero value means interrupts can be recognized; on other hardware, one means interrupts can be recognized
- OS can
  - Examine current interrupt mask (find out whether interrupt recognition is enabled)
  - Set interrupt mask to prevent interrupt recognition
  - Clear interrupt mask to allow interrupt recognition

Masking Interrupts

- Important principle:

  No operating system function should contain code to explicitly enable interrupt recognition.

- Technique used: given function
  - Saves current interrupt status
  - Disables interrupt recognition
  - Proceeds through critical section
  - Restores interrupt recognition status from saved copy
- Key insight: save / restore allows arbitrary call nesting

Why Interrupt Masking is Insufficient

- It works! But...
- Disabling interrupt recognition penalizes all processes when one process executes a critical section
  - Prevents recognition of I /O activity reports (i.e. interrupts)
  - Restricts execution to one process for the entire system
- Can interfere with the scheduling invariant (low-priority process can block a high-priority process for which I /O has completed)
- Does not provide a policy that controls which process can access the critical section at a given time

High-level Mutual Exclusion

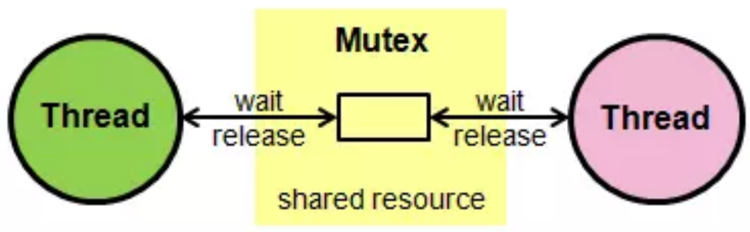
- Idea is to create a facility with the following properties
  - Permit designer to specify multiple critical sections
  - Allow independent control of each critical section
  - Provide an access policy (e.g., FIFO)
- A single mechanism, the *counting semaphore*, suffices
  - A semaphore, s, consists of an integer count and a set of blocked processes.
  - Once a semaphore has been created, processes use two functions, *wait* and *signal*.

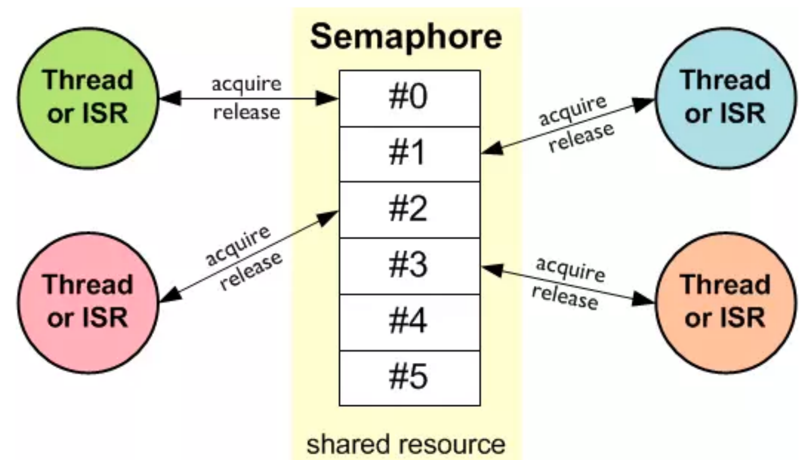
Counting Semaphore

- Operating system abstraction
- Instance can be created dynamically
- Each instance given unique name
  - Typically an integer
  - Known as a *semaphore ID*
- Instance consists of a tuple (count, set)
  - *Count* is an integer
  - *Set* is a set of processes waiting on the semaphore

Question

- What is the difference between a mutex and a semaphore?





https://www.keil.com/pack/doc/CMSIS/RTOS/html/group__CMSIS__RTOS__SemaphoreMgmt.html

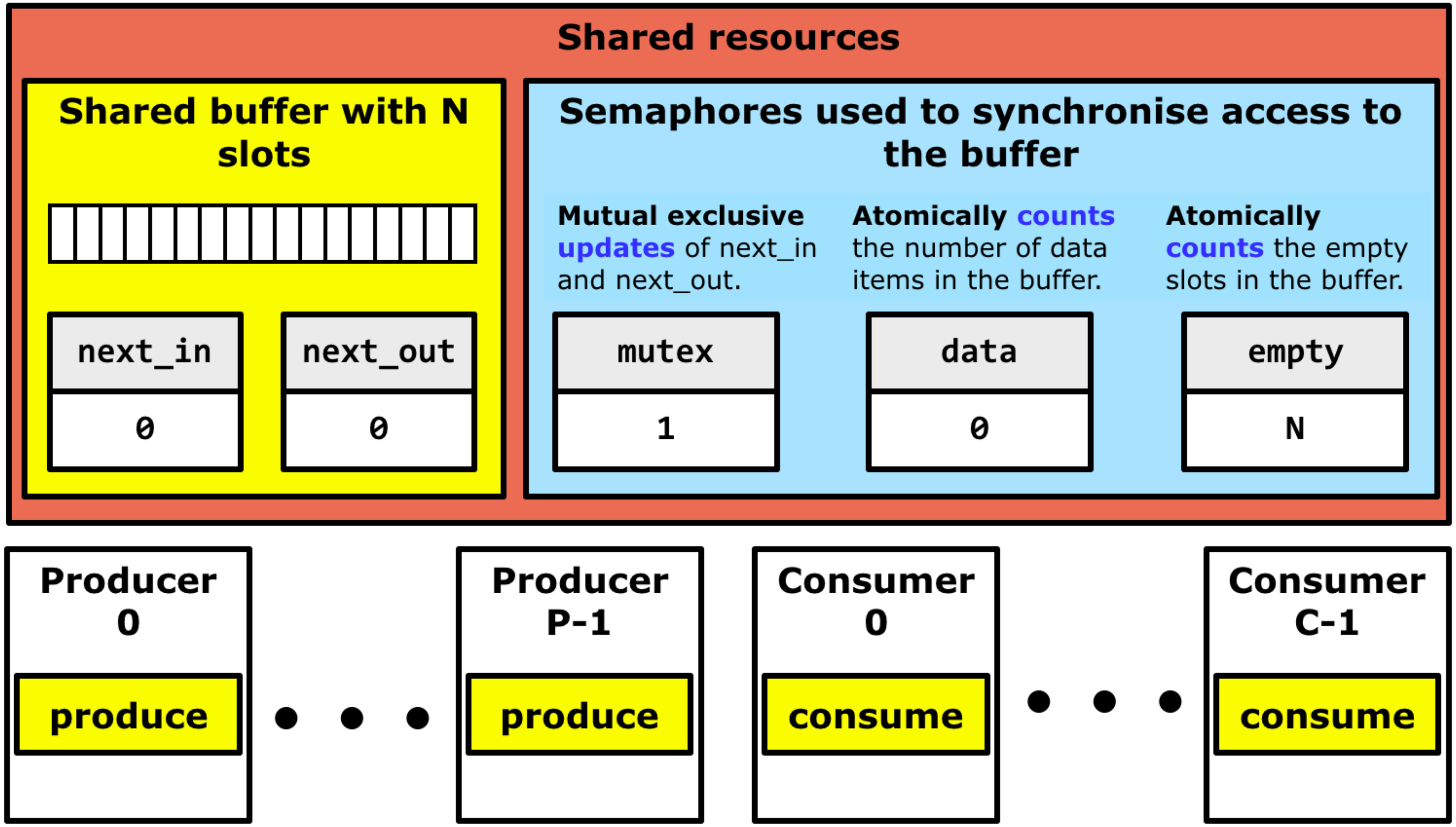
Operations on Semaphores

- *Create* a new semaphore
- *Delete* an existing semaphore
- *Wait* on an existing semaphore (also *P* or *down*)
  - Decrements count and continue (> 0)
  - Adds calling process to set of waiting processes if resulting count is negative (< 0)
- *Signal* an existing semaphore (also *V* or *up*)
  - Increments count
  - Makes a process ready if any are waiting

Key Uses of Counting Semaphores

- Three semaphores are used in the solution:
  - `empty`: (initially buffer size N) contains the number of unused slots in the buffer
  - `full`: (initially 0) contains the number of filled slots in the buffer
  - `mutex`: (initially 1), controls access to buffer
- Two basic paradigms
  - Cooperative mutual exclusion: `mutex`
  - Direct synchronization (e.g., producer-consumer): `empty` and `full`

Implementation Overview of Counting Semaphores



http://www.it.uu.se/education/course/homepage/os/vt18/module-4/bounded-buffer/

Mutual Exclusion with Semaphores in Xinu

- Initialize: create a mutex semaphore

```
sid = semcreate (1);
```

  An integer identifier   initial count

- Use: bracket critical sections of code with calls to *wait* and *signal*

```
wait(sid); //decrease a semaphore
… critical section (use shared resource) …
signal(sid);
```

- Guarantees only one process can access the critical section at any time (others will be blocked)

Producer-Consumer Synchronization with Semaphores in Xinu

- Two semaphores suffice
- Initialize: create producer and consumer semaphores

```
psem = semcreate (BUFFER_SIZE);
csem = semcreate (0);
```

- Producer algorithm

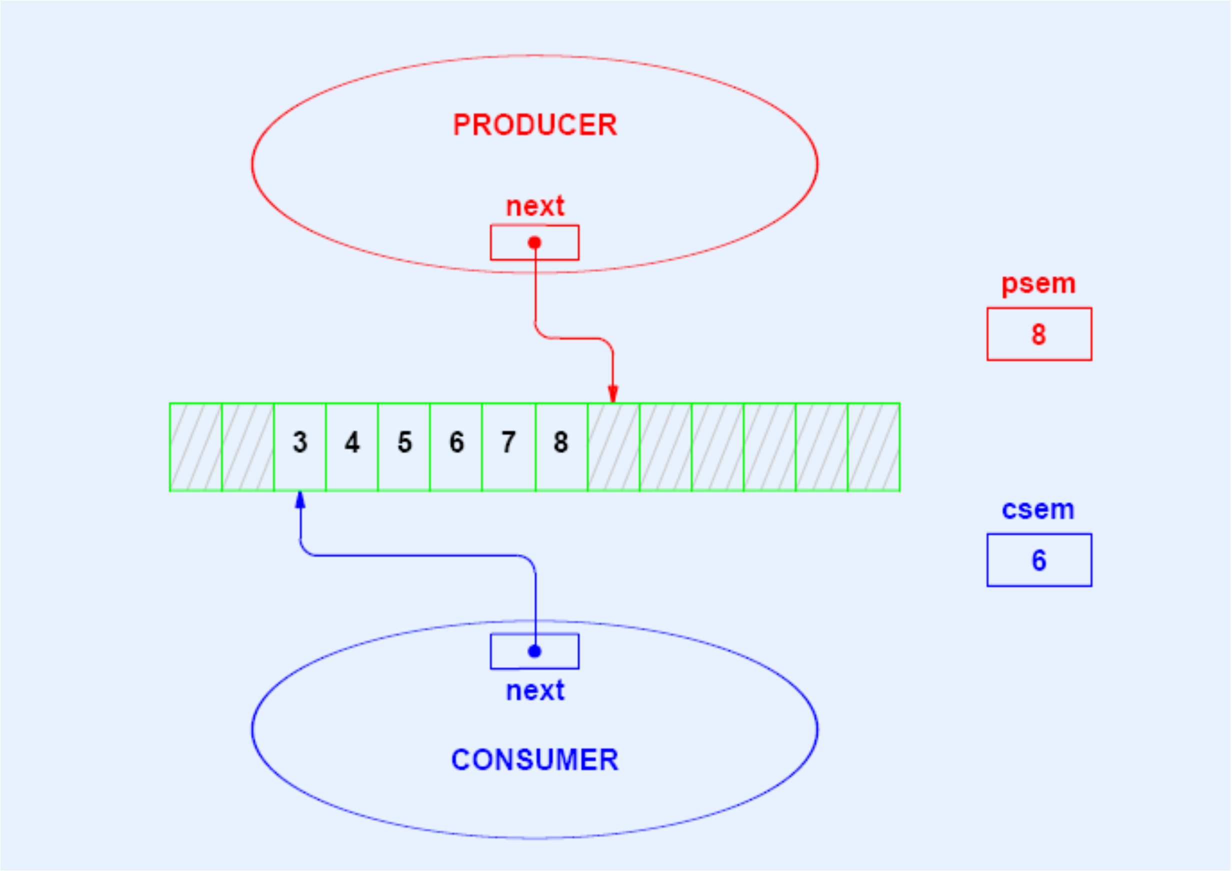
```
repeat forever {
    wait(psem);
    fill_next_buffer_slot();
    signal(csem);
}
```

Producer-Consumer Synchronization in Xinu (continued)

- Consumer algorithm

```
repeat forever {
    wait (csem);
    extract_from_buffer_slot();
    signal(psem);
}
```

Interpretation of Producer-Consumer Semaphores in Xinu



- *psem* counts items currently in the buffer
- *csem* counts unused slots in the buffer

Semaphore Invariant in Xinu

- Establishes relationship between semaphore concept and implementation
- Makes code easy to create and understand
- Must be reestablished after each operation
- *Wait* and *signal* maintain the following invariant regarding the count of a semaphore:

Semaphore invariant: a nonnegative semaphore count means that the queue is empty; a semaphore count of negative N *means that the queue contains* N *waiting processes.*

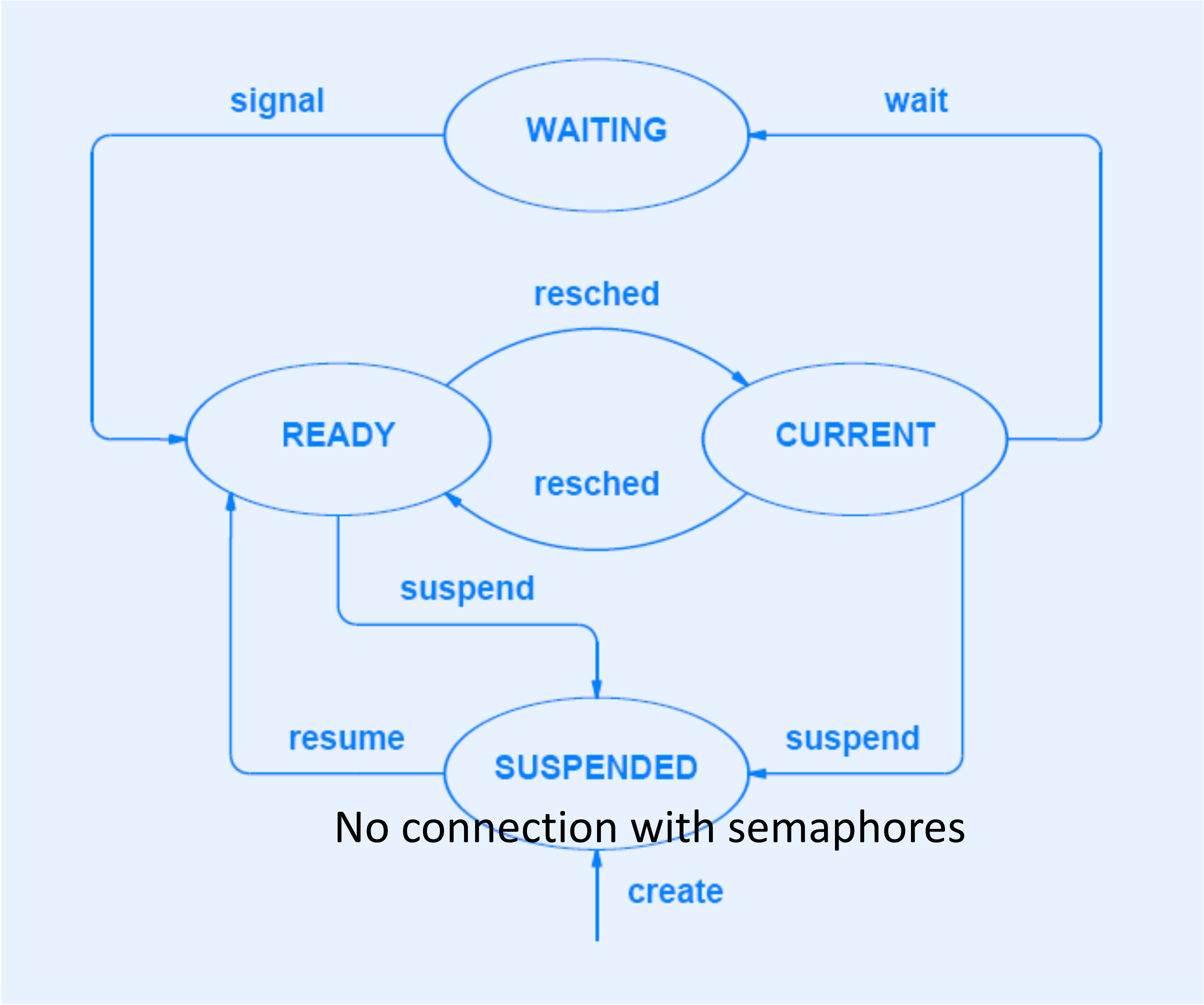
Counting Semaphores in Xinu

- Stored in an array of semaphore entries
- Each entry, a semaphore, ***s*,**
  - Corresponds to one instance (one semaphore)
  - Contains an integer count and pointer to list of processes
  - Decrement/increment the count of semaphore *s*
- Semaphore ID is index into array
- Policy for management of waiting processes
  - Create a *FIFO queue* for each semaphore
  - Use the queue to store processes that are waiting

Process State Used with Semaphores

- When process is waiting on a semaphore, process is NOT
  - Executing
  - Ready
  - Suspended
- Suspended state is only used by *suspend* and *resume*
- A new state is needed
  - We will use the *WAITING* state for a process blocked a semaphore and and use symbolic constant *PR_WAIT*

State Transitions with Waiting State



No connection with semaphores

Semaphore Definitions

Semaphore table entry structure declaration; semaphore constants.

```
/* semaphore.h - isbadsem */

#ifndef NSEM
#define NSEM      120  /* Number of semaphores, if not defined */
#endif

/* Semaphore state definitions */

#define S_FREE 0      /* Semaphore table entry is available   */
#define S_USED 1      /* Semaphore table entry is in use      */

/* Semaphore table entry */
struct sentry {
       byte sstate;    /* Whether entry is S_FREE or S_USED   */
       int32 scount;   /* current integer count for the semaphore */
       qid16 squeue;   /* Queue of processes that are waiting */
                        /*    on the semaphore                 */
};

extern struct sentry semtab[];

#define isbadsem(s)    ((int32)(s) < 0 || (s) >= NSEM)
```

- Store semaphore information
- Each entry in *semtab* corresponds to one semaphore

Implementation of Wait (part 1)

- Decrements the count of a semaphore
- If the count remains nonnegative, *wait* returns to the caller immediately.

```
/* wait.c - wait */

#include <xinu.h>

/*------------------------------------------------------------------------
 * wait - Cause current process to wait on a semaphore
 *------------------------------------------------------------------------
 */
syscall wait(
        sid32 sem /* Semaphore on which to wait */
       )
{
        intmask mask; /* Saved interrupt mask */
        struct procent *prptr; /* Ptr to process' table entry */
        struct sentry *semptr; /* Ptr to sempahore table entry */

        mask = disable();
        if (isbadsem(sem)) {
            restore(mask);
            return SYSERR;
        }
        semptr = &semtab[sem];
        if (semptr->sstate == S_FREE) {
            restore(mask);
            return SYSERR;
        }
```

Implementation of Wait (part 2)

```
        if (--(semptr->scount) < 0) {          /* If caller must block */
            prptr = &proctab[currpid];
            prptr->prstate = PR_WAIT;          /* Set state to waiting */
            prptr->prsem = sem;                /* Record semaphore ID  */
            enqueue(currpid,semptr->squeue);   /* Enqueue on semaphore */
            resched();                         /*   and reschedule     */
        }

        restore(mask);
        return OK;
}
```

A call to ctxsw

A process remains in the waiting state until the process reaches the head of the queue and some other process signals the semaphore.

Semaphore Queuing Policy

- Determines which process to select among those waiting
- Needed when *signal* called
- A question arises: if multiple processes are waiting, which one should *signal* select? Several policies have been used:
  - First-Come-First-Served (FCFS or FIFO)
  - Process priority
  - Random

Question

- Assume the scheduling goal is "fairness."
- Which semaphore queuing policy best implements the goal?
  - In other words, how should we interpret "fairness?"
- Semaphore policy can interact with scheduling policy
  - Should a low-priority process be allowed to access a resource if a high-priority process is also waiting?
  - Should a low-priority process be blocked forever if high-priority processes continually use a resource?

Choosing a Semaphore Queueing Policy

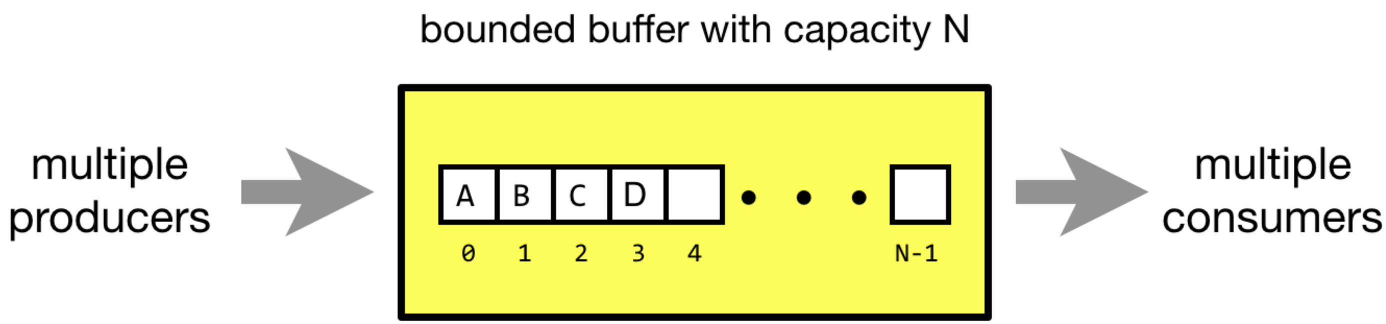
- Difficult
- No single best answer
  - Fairness not easy to define
  - Scheduling and coordination interact in subtle ways
  - May affect other OS policies
- Selecting the highest priority waiting process
  - Violate the principle of fairness
- Interactions of heuristic policies may produce unexpected results

Semaphore Queuing Policy in Xinu

- First-come-first-serve
  - Choose the process that has been waiting the longest.
- Straightforward to implement
  - Create a FIFO queue for each semaphore
- Extremely efficient
- Potential problem: low-priority process can access a resource while a high-priority process remains blocked, leading to *priority inversion problem*
  - One alternative: *Random*

Implementation of FIFO Semaphore Policy

- Each semaphore uses *a list* to manage waiting processes
- List is run as a queue: insertions at one end and deletions at the other
  - *Signal* operation removes (FIFO) first process from the queue and puts it on list of ready processes.



http://www.it.uu.se/education/course/homepage/os/vt18/module-4/bounded-buffer/

Implementation of Signal (part 1)

```
/* signal.c - signal */

#include <xinu.h>

/*------------------------------------------------------------------------
 * signal - Signal a semaphore, releasing a process if one is waiting
 *------------------------------------------------------------------------
 */
syscall signal(
        sid32 sem                               /* ID of semaphore to signal    */
        )
{
        intmask mask;                           /* Saved interrupt mask         */
        struct sentry *semptr;                  /* Ptr to sempahore table entry */

        mask = disable();
        if (isbadsem(sem)) {
            restore(mask);
            return SYSERR;
        }
        semptr= &semtab[sem];
        if (semptr->sstate == S_FREE) {
            restore(mask);
            return SYSERR;
        }
```

Implementation of Signal (part 2)

```
        if ((semptr->scount++) < 0) {       /* Release a waiting process */
            ready(dequeue(semptr->squeue));
        }
        restore(mask);
        return OK;
}
```

Add the current process to the queue if the new count is negative if the queue is nonempty

Semaphore Allocation

- Static
  - A fixed set of semaphores are defined at compile time
  - Saving space, reduce processing overhead and more efficient, but less powerful
- Dynamic
  - Semaphores are created at runtime and and deallocated when they are no longer needed
  - The ability to accommodate new uses, more flexible
- Xinu supports dynamic allocation

Xinu Semcreate (part 1)

- Processes can create semaphores dynamically
- A given process can create multiple semaphores

```
/* semcreate.c - semcreate, newsem */

#include <xinu.h>

local sid32 newsem(void);
/*------------------------------------------------------------------------
 * semcreate - Create a new semaphore and return the ID to the caller
 *------------------------------------------------------------------------
 */
sid32 semcreate(
        int32 count                  /* Initial semaphore count (nonnegative)*/
      )
{
        intmask mask;                          /* Saved interrupt mask   */
        sid32 sem;                             /* Semaphore ID to return */

        mask = disable();
        if (count < 0 || ((sem=newsem())==SYSERR)) {
            restore(mask);
            return SYSERR;
        }
        semtab[sem].scount = count;            /* Initialize table entry */
        restore(mask);
        return sem;
}
```

Searches the semaphore table, semtab, for an unused entry and initializes the count.

Xinu Semcreate (part 2)

```
/*------------------------------------------------------------------------
 * newsem - Allocate an unused semaphore and return its index
 *------------------------------------------------------------------------
 */
```

To search the table, *semcreate* calls function *newsem*, which iterates through all *NSEM* entries of the table.

Optimize searching: allow a search to start where the last search left off

```
local sid32 newsem(void)
{
        static sid32 nextsem = 0; /* Next semaphore index to try */
        sid32 sem; /* Semaphore ID to return */
        int32 i; /* Iterate through # entries */

        for (i=0 ; i<NSEM ; i++) {   /* all NSEM entries of the table */
            sem = nextsem++;
            if (nextsem >= NSEM)
                nextsem = 0;
            if (semtab[sem].sstate == S_FREE) {
                semtab[sem].sstate = S_USED;
                return sem;        /* return the table index as the ID */
            }
        }
        return SYSERR;
}
```

Semaphore Deletion

- Function *semdelete* reverses the actions of *semcreate*
- One or more processes may be waiting when semaphore is deleted
- Xinu policy - deallocating a semaphore requires:
  - Verify if a valid semaphore ID and the corresponding entry in the semaphore table is currently in use.
  - Set the state of the entry to S_FREE
  - Make process *ready*

Xinu Semdelete (part 1)

```
/* semdelete.c - semdelete */

#include <xinu.h>

/*------------------------------------------------------------------------
 * semdelete - Delete a semaphore by releasing its table entry
 *------------------------------------------------------------------------
 */
syscall semdelete(
        sid32 sem                          /* ID of semaphore to delete   */
       )
{
        intmask mask;                      /* Saved interrupt mask         */
        struct sentry *semptr;             /* Ptr to semaphore table entry */

        mask = disable();
        if (isbadsem(sem)) {
            restore(mask);
            return SYSERR;
        }
        semptr = &semtab[sem];
        if (semptr->sstate == S_FREE) {
            restore(mask);
            return SYSERR;
        }
        semptr->sstate = S_FREE;
```

1. Verifies that sem specifies a valid ID and that the corresponding entry in the table is currently in use.

2. The table entry can be reused

Xinu Semdelete (part 2)

```
        resched_cntl(DEFER_START);
        while (semptr->scount++ < 0) {  /* Free all waiting processes */
            ready(getfirst(semptr->squeue));
        }
        resched_cntl(DEFER_STOP);
        restore(mask);
        return OK;
}
```

3. The set of processes are waiting on the semaphore and makes each process ready.

Do you understand semaphores?

Thought Problem (The Lock Convoy)

- Definition: a performance problem can occur when using locks for concurrency control in a multithreaded application
  - Occur when multiple threads of equal priority contend repeatedly for the same lock
- One process creates a semaphore

```
mutex = semcreate(1);
```

- Three processes execute the following

```
process convoy(char_to_print)
    do forever {
        wait(mutex);
        print(char_to_print);
        signal(mutex);
    }
```

- The processes print characters *A*, *B*, and *C*, respectively

Lock Convoy Problem
(continued)

- Initial output
  - 20 *A*’s, 20 *B*’s, 20 *C*’s, 20 *A*’s, etc.
- After tens of seconds

  ABCABCABC...
- Facts
  - Everything is correct
  - No other processes are executing
  - Print is nonblocking (polled I / O)

Lock Convoy Problem
(continued)

- Questions
  - How long is thinking time?
  - Why does convoy start?
  - Will output switch back given enough time?
  - Did knowing the policies or the implementation of the scheduler and semaphore mechanisms make the convoy behavior obvious?

Summary (1/2)

- Process synchronization fundamental
  - Supplied to applications
  - Used inside OS
- Low-level mutual exclusion
  - Masks hardware interrupts
  - Avoids rescheduling
  - Insufficient for all coordination

Summary (2/2)

- High-level coordination
  - Used by subsets of processes
  - Available inside and outside OS
  - Implemented with counting semaphore
- Counting semaphore
  - Powerful abstraction
  - Provides mutual exclusion and producer / consumer synchronization