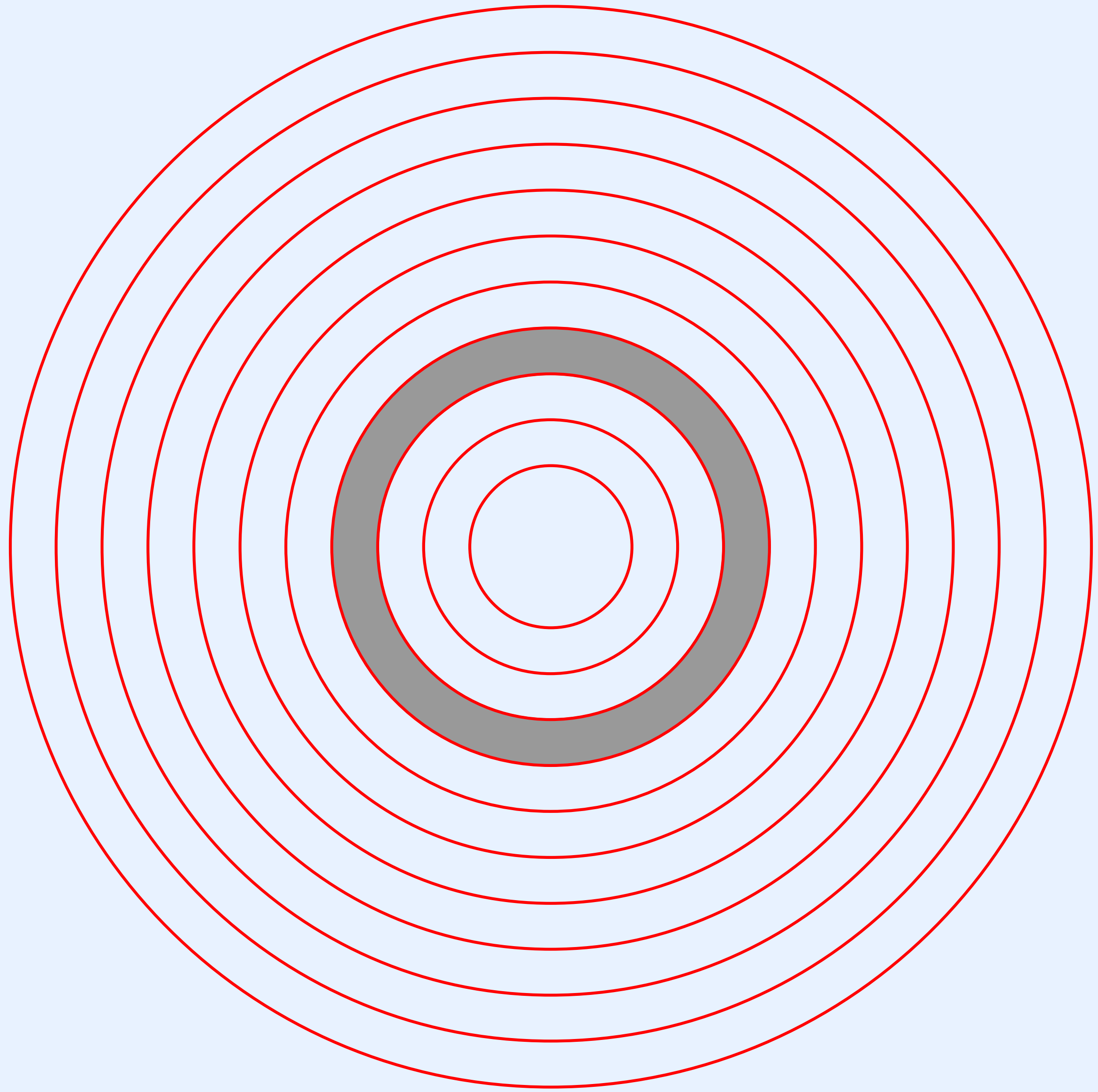
Module IV

Process Management:
Coordination And Synchronization

Location Of Process Coordination In The Hierarchy



Coordination Of Processes

- Is necessary in a concurrent system
- Avoids conflicts when multiple processes access shared items
- Allows a set of processes to cooperate
- Can also be used when
  - A process waits for I/O
  - A process waits for another process
- An example of cooperation among processes: UNIX pipes

Two Approaches To Process Coordination

- Use a hardware mechanism
  - Most useful / important on multiprocessor hardware
  - Often relies on *busy waiting*
- Use an operating system mechanism
  - Works well with single processor hardware
  - Does not entail unnecessary execution

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

Two Key Situations That Process Coordination Mechanisms Handle

- Producer / consumer interaction
- Mutual exclusion

Producer-Consumer Synchronization

- Typical scenario: a FIFO buffer shared by multiple processes
  - Processes that deposit items into the buffer are called *producers*
  - Processes that extract items from the buffer are called *consumers*
- The programmer must guarantee
  - When the buffer is full, a producer will block until space is available
  - When the buffer is empty, a consumer will block until an item has been deposited
- A given process may act as a consumer for one buffer and a producer for another
- Example: in Unix pipeline, a process may read input from one pipe and write output to another

```
cat employees | grep Name: | sort
```

Mutual Exclusion

- In a concurrent system, multiple processes may attempt to access shared data items
- If one process starts to change a data item and then a context switch allows another process to run and access the data item, the results can be incorrect
- We use the term *atomic* to refer to an operation that is indivisible (i.e., the hardware performs the operation in a single instruction that cannot be interrupted)
- Many data operations are non-atomic, which means a sequence of multiple operations are used to change a data item
- Programmers must take steps to ensure that when one process executes a sequence of operations to change a data item, no other process can attempt to make changes concurrently

Recall

- Even trivial changes to a shared variable (e.g., x++) can require a sequence of hardware operations
- Anyone working with concurrent processes must guard *every* access to shared data items

To Prevent Problems

- A programmer must ensure that only one process accesses a shared item at any time
- General approach
  - Once a process obtains access, make all other processes wait
  - When a process finishes accessing the item, grant access to one of the waiting processes
- Three techniques are available
  - Hardware mechanisms that disable and restore interrupts
  - Hardware spin lock instructions
  - Semaphores (implemented in software)

Handling Mutual Exclusion With Spin Locks

- Used in multicore CPUs; does *not* work for a single processor
- A special hardware operation allows a core to test and/or set a special *lock* atomically
- The lock may consist of special hardware or may be a location in memory
- The hardware guarantees that only one core will be allowed to set the lock at any time
- The mechanism is known as a *spin lock* because a core uses *busy waiting* to gain access
- Busy waiting literally means the core executes a loop that tests the spin lock repeatedly until access is granted
- The approach was once known as *test-and-set*

An Example Of A Spin Lock (x86)

- An instruction performs an atomic compare and exchange (*cmpxchg*)
- Spin loop: repeat the following
  - Place an “unlocked” value (e.g., 0) in register *eax*
  - Place a “locked” value (e.g., 1) in register *ebx*
  - Place the address of a memory location to be used as a lock in register *ecx*
  - 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 the lock, assign the “unlocked” value to the lock location in memory

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 %eax with memory (%ecx)      */
                                /* if equal                             */
                                /*      load %ebx in memory (%ecx)      */
                                /* else                                 */
                                /*      load %ebx in %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
```

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

- A programmer must allocate a semaphore for each item to be protected
- The semaphore acts as a *mutual exclusion* semaphore, and is known colloquially as a *mutex* semaphore
- All applications must be programmed to use the mutex semaphore before accessing the shared item
- The operating system guarantees that only one process can access the shared item at a given time
- The implementation avoids busy waiting

Definition Of Critical Section

- Each piece of shared data must be protected from concurrent access
- A programmer inserts mutex operations
  - Before access to the shared item
  - After access to the shared item
- The protected code is known as a *critical section*
- Mutex operations must be placed in each function that accesses the shared item

Mutual Exclusion Inside An Operating System

- Several possible approaches have been used
- Examples: allow only one process at a time to
  - Run operating system code
  - Run a given operating system function
  - Access a given operating system component (a single component may comprise multiple functions)
- Allowing more processes to execute concurrently increases performance
- The general principle is:

to maximize performance, choose the smallest possible granularity for mutual exclusion

Low-Level Mutual Exclusion

- Mutual exclusion is needed in two places
  - In application processes
  - Inside the operating system
- On a single-processor system, mutual exclusion can be guaranteed provided that no context switching occurs
- A context switch can only occur when
  - A device interrupts
  - A process calls *resched*
- Low-level mutual exclusion technique: turn off interrupts and avoid rescheduling

Interrupt Mask

- A hardware mechanism that controls interrupts
- Implemented by an internal machine register, and may be part of *processor status word*
- On some hardware, a zero value means interrupts can occur; on other hardware, a non-zero value means interrupts can occur
- The OS can
  - Examine the current interrupt mask (find out whether interrupts are enabled)
  - Set the interrupt mask to prevent interrupts
  - Clear the interrupt mask to allow interrupts

Masking Interrupts

- Important principle:

  No operating system function should contain code to explicitly enable interrupts.

- Technique used: a given function
  - Saves the current interrupt status
  - *Disables* interrupts
  - Proceeds through a critical section
  - *Restores* the interrupt status from the saved copy
- Key insight: save / restore allows nested calls

Why Interrupt Masking Is Insufficient

- It works! But...
- Stopping interrupts penalizes *all* processes when one process executes a critical section
  - It stops all I/O activity (and some device interrupts must be serviced within a specifies period)
  - It restricts execution to one process for the entire system
- Disabling interrupts can interfere with the scheduling invariant and lead to a *priority inversion* where a low-priority process prevents execution of a high-priority process for which I/O has completed
- Disabling interrupts does not provide a policy that controls which process can access a critical section at a given time
- When used, a programmer must minimize the amount of time interrupts remain disabled

High-Level Mutual Exclusion

- The idea is to create an operating system facility with the following properties
  - Permit applications to define multiple, independent critical sections
  - Allow processes to compete for access to each critical section independent of other critical sections
  - Provide an access policy that specifies how waiting processes gain access
- Good news: a single mechanism, the *counting semaphore*, solves the problem

Counting Semaphore

- An operating system abstraction
- An instance can be created dynamically
- Each instance is given a unique name
  - Typically an integer
  - Known as a *semaphore ID*
- An instance consists of a 2-tuple (count, set)
  - *Count* is an integer
  - *Set* is a set of processes that are waiting on the semaphore

Operations On Semaphores

- *Create* a new semaphore
- *Delete* an existing semaphore
- *Wait* on an existing semaphore
  - Decrements the count
  - Adds the calling process to set of waiting processes if the resulting count is negative
- *Signal* an existing semaphore
  - Increments the count
  - Makes a process ready if any are waiting

Xinu Semaphore Functions

semid = semcreate(initial_count)	Creates a semaphore and returns an ID
semdelete(semid)	Deletes the specified semaphore
wait(semid)	Waits on the specified semaphore
signal(semid)	signals the specified semaphore

Key Uses Of Counting Semaphores

- Semaphores have many potential uses
- However, using semaphores to solve complex coordination problems can be intellectually challenging
- We will consider two straightforward ways to use semaphores
  - Cooperative mutual exclusion
  - Producer-consumer synchronization (direct synchronization)

Cooperative Mutual Exclusion With Semaphores

- A set of processes use a semaphore to guard a shared item
- Initialize: create a mutex semaphore

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

- Use: bracket each critical section in the code with calls to *wait* and *signal*

```
 wait(sid);
...critical section to use the shared item...
signal(sid);
```

- All processes must agree to use semaphores (hence the term *cooperative*)
- Only one process will access the critical section at any time (others will be blocked)

A Potential Problem: Deadlock

- Consider two processes that use semaphores to protect two data items, x and y
- The two semaphores are created

```
semidx = semcreate(1);   semidy = semcreate(1);
```

- Then the two processes take the following steps

/* Process 1 */		/* Process 2 */
...		...
wait(semidx);		wait(semidy);
start to modify x		start to modify y
wait(semidy);	**deadlock!** →	wait(semidx);
modify y		modify x
signal(semidy);		signal(semidx);
finish modifying x		finish modifying y
signal(semidx);		signal(semidy);

When Using Semaphores For Mutual Exclusion

- Good news: counting semaphores work well when a set of processes needs exclusive access to a single resource
- Bad news: using semaphores with multiple resources can be tricky
- To avoid trouble
  - Limit mutual exclusion to a single resource at any time, when possible
  - When processes must obtain exclusive access to multiple resources, insure that all processes access and release the resources in the same order

Producer-Consumer Synchronization With Semaphores

- Two semaphores suffice to control processes accessing a shared buffer
- Initialize: create producer and consumer semaphores

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

- The producer algorithm

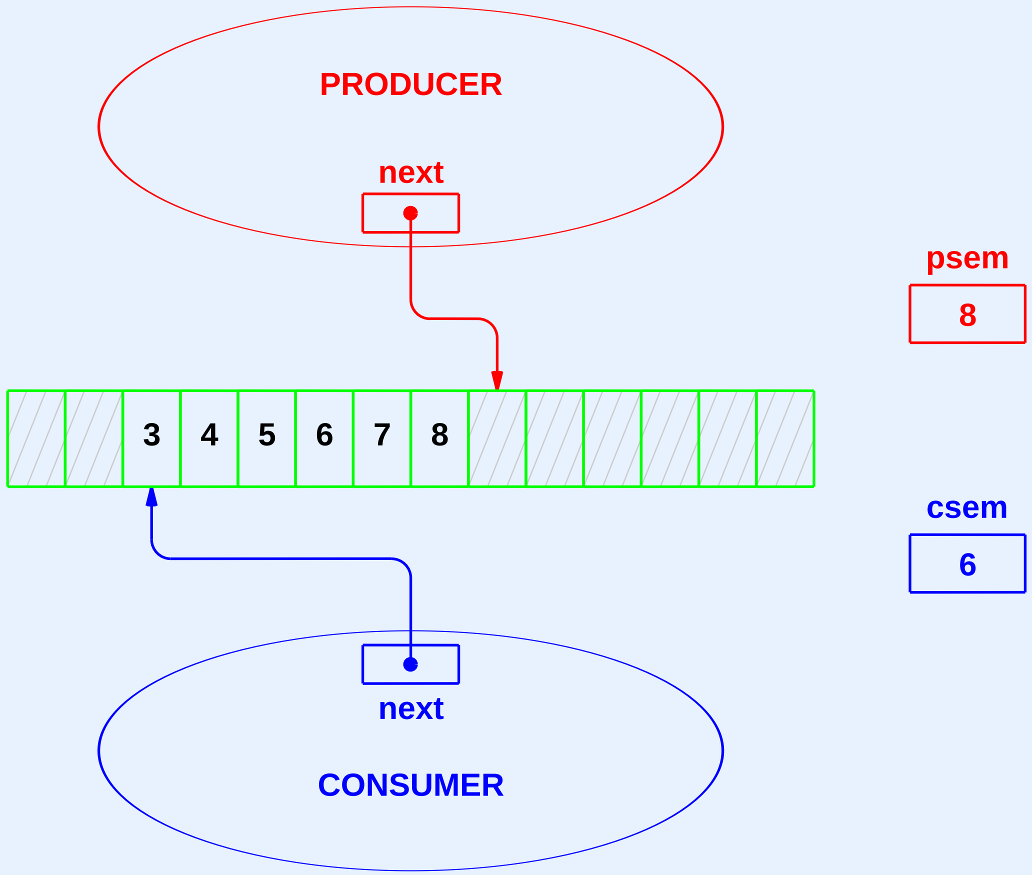
```
repeat forever {
        generate an item to be added to the buffer;
        wait(psem);
        fill_next_buffer_slot;
        signal(csem);
}
```

Producer-Consumer Synchronization With Semaphores
(continued)

- The consumer algorithm

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

An Interpretation Of Producer-Consumer Semaphores



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

The Semaphore Invariant

- Establishes a relationship between the semaphore concept and its implementation
- Makes the code easy to create and understand
- Must be re-established after each semaphore operation
- Is surprisingly elegant:

A nonnegative semaphore count means that the set of processes is empty. A count of negative *N* means that the set contains *N* waiting processes.

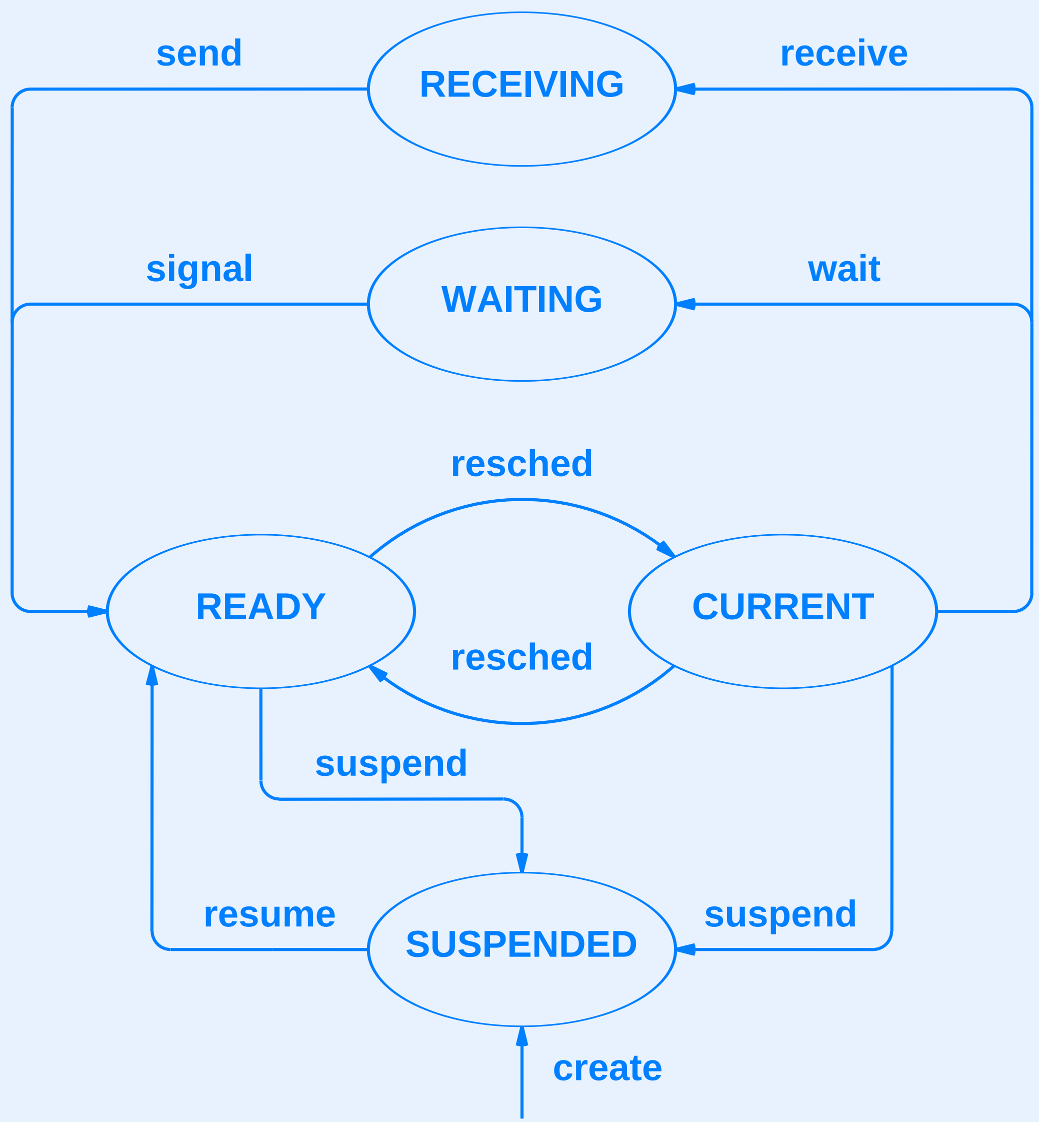
Counting Semaphores In Xinu

- Are stored in an array of semaphore entries
- Each entry
  - Corresponds to one instance (one semaphore)
  - Contains an integer count and pointer to a list of processes
- The ID of a semaphore is its index in the array
- The policy for management of waiting processes is FIFO

A Process State Used With Semaphores

- When a process is waiting on a semaphore, the process is not
  - Executing
  - Ready
  - Suspended
  - Receiving
- Note: the suspended state is only used by *suspend* and *resume*
- Therefore a new state is needed
- We will use the *WAITING* state for a process blocked by a semaphore

State Transitions With Waiting State



Semaphore Definitions

```
/* 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;         /* 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)
```

Implementation Of Wait (Part 1)

```
/* 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;
}
```

- Moving a process to the waiting state only requires a few lines of code
  - Set the state of the current process to PR_WAIT
  - Record the ID of the semaphore on which the process is waiting in field *prsem*
  - Call *resched*

The Semaphore Queuing Policy

- Determines which process to select among those that are waiting
- Is only used when *signal* is called and processes are waiting
- Examples of possible policies
  - First-Come-First-Served (FCFS or FIFO)
  - Process priority
  - Random

Consequences Of A Semaphore Queuing Policy

- The goal is “fairness”
- Which semaphore queuing policy implements the goal the best?
- In other words, how should we interpret fairness?
- The 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 use a resource?

Choosing A Semaphore Queueing Policy

- The choice is difficult
- There is no single best answer
  - Fairness not easy to define
  - Scheduling and coordination interact in subtle ways
  - The choice may affect other OS policies
- The interactions of heuristic policies may produce unexpected results

The Semaphore Queuing Policy In Xinu

- Xinu uses first-come-first-served
- The approach has several advantages
  - Is straightforward to implement
  - Is extremely efficient
  - Works well for traditional uses of semaphores
  - Guarantees all contending processes will obtain access
- The FIFO approach has an interesting disadvantage: a low-priority process can obtain access to a resource while a high-priority process remains blocked

Implementation Of Xinu's FIFO Semaphore Policy

- Recall: each semaphore has a list of processes
- For a FIFO policy, the list is treated as a queue
- When it needs to insert the current process on a list, *wait* enqueues the calling process at the tail of the queue
- When it chooses a waiting process to run, *signal* selects the process at the head of the queue
- The code for signal follows

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;
}
```

- Notice how little code is required to signal a semaphore

Possible Semaphore Creation Strategies

- Static
  - All semaphores are defined at compile time
  - The approach is more efficient, but less powerful
- Dynamic
  - Semaphores are created at runtime
  - The approach is more flexible
- Xinu supports dynamic semaphore allocation, but to achieve efficiency preallocates a fixed-size array of possible semaphores

Xinu Semcreate (Part 1)

```
/* 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      */
        )
{
        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;
}
```

Xinu Semcreate (Part 2)

```
/*------------------------------------------------------------------------
 *  newsem  -  Allocate an unused semaphore and return its index
 *------------------------------------------------------------------------
 */
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++) {
		sem = nextsem++;
		if (nextsem >= NSEM)
			nextsem = 0;
		if (semtab[sem].sstate == S_FREE) {
			semtab[sem].sstate = S_USED;
			return sem;
		}
	}
	return SYSERR;
}
```

Semaphore Deletion

- Wrinkle: one or more processes may be waiting when a semaphore is deleted
- We must choose how to dispose of each waiting process
- The Xinu disposition policy: if a process is waiting on a semaphore when the semaphore is deleted, the process becomes 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;
```

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;
}
```

- Deferred rescheduling allows all waiting processes to be made ready before any of them to run
- Before it ends deferred rescheduling, semdelete ensures the semaphore data structure is ready for other processes to use

Do you understand semaphores?

Semaphore Behavior (A True Story)

- A process creates a semaphore

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

- Three processes then execute the following code

```
process convoy(char_to_print)
   do forever {
       think (i.e., use CPU);
       wait(mutex);
       print(char_to_print);
       signal(mutex);
   }
```

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

The Convoy

- The initial output is
  - 20 *A*'s, 20 *B*'s, 20 *C*'s, 20 *A*'s, etc.
- After tens of seconds, however, the output becomes *ABCABCABC*...
- Facts
  - Everything is correct
  - No other processes are executing
  - The output is nonblocking (i.e., it uses polled I/O)

The Convoy
(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

- Process synchronization is used in two ways
  - As a service supplied to applications
  - As an internal facility used inside the OS itself
- Low-level mutual exclusion
  - Masks hardware interrupts
  - Avoids rescheduling
  - Is insufficient for all coordination needs

Summary
(continued)

- High-level process coordination is
  - Used by subsets of processes
  - Available inside and outside the OS
  - Implemented with counting semaphore
- Counting semaphore
  - A powerful abstraction implemented in software
  - Provides mutual exclusion and producer / consumer synchronization

Questions?