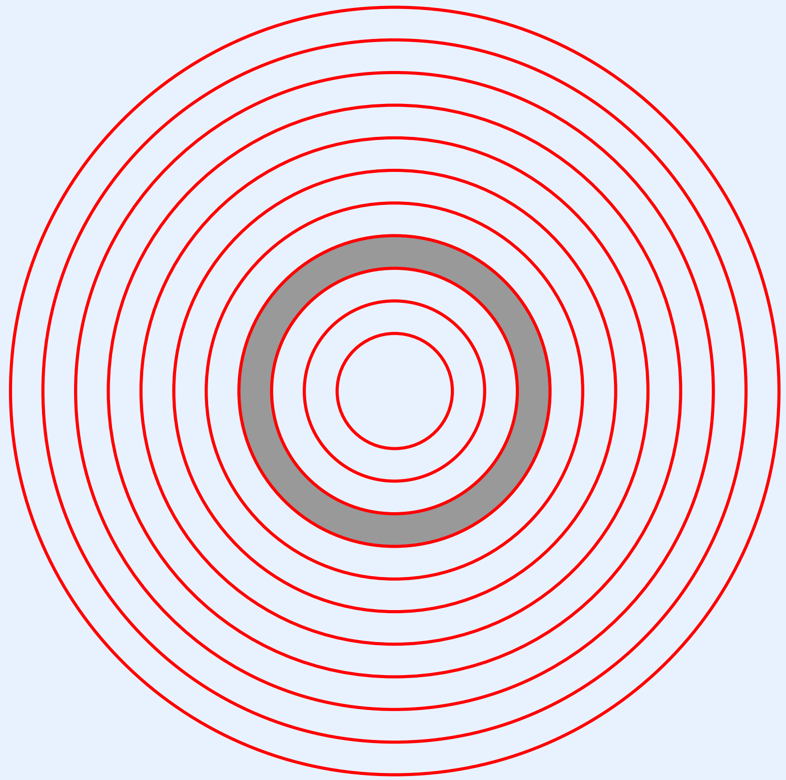
Module V

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
  - Most useful for multiprocessor systems
  - May rely on *busy waiting*
- Use operating system mechanisms
  - Works well with a single processor
  - No unnecessary execution

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

Key Situations That Process Coordination Mechanisms Handle

- Producer / consumer interaction
- Mutual exclusion

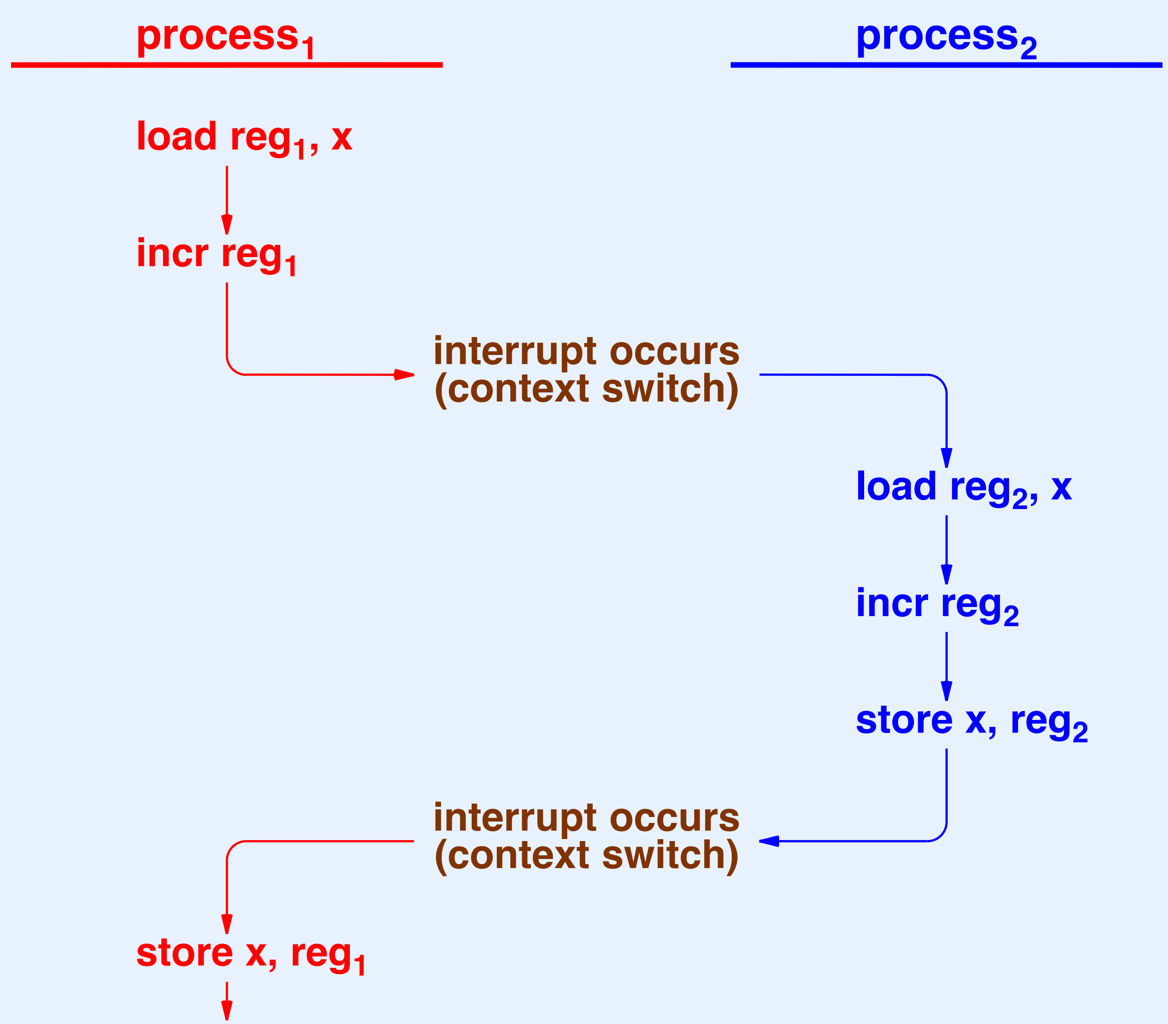
Producer-Consumer Synchronization

- 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
- Nonatomic 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



To Prevent Problems

- Ensure that only one process accesses a shared item at any time
- Trick: once a process obtains access, make all other processes wait
- Three solutions
  - Spin lock hardware instructions
  - Disabling all interrupts
  - Semaphores (implemented in software)

Handling Mutual Exclusion With Spin Locks

- Used in multiprocessors; does not work for single processor
- Atomic hardware operation tests a memory location and changes it
- Known as *test-and-set*
- Also called *spin lock* because it involves *busy waiting*

Example Of A Spin Lock (x86)

- Instruction performs atomic compare and exchange (*cmpxchg*)
- 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 %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

- Semaphore allocated for item to be protected
- Known as a *mutex* semaphore
- Applications must be programmed to use the mutex semaphore before accessing shared item
- Operating system guarantees only one process can access the shared item at a given time
- Implementation avoids busy waiting

Definition Of Critical Section

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

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

Low-Level Mutual Exclusion

- Mutual exclusion needed
  - By application processes
  - Inside operating system
- Mutual exclusion can be guaranteed provided no context switching occurs
- Context changed by
  - Interrupts
  - Calls to *resched*
- Low-level mutual exclusion: mask interrupts and avoid rescheduling

Interrupt Mask

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

Masking Interrupts

- Important principle:

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

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

Why Interrupt Masking Is Insufficient

- It works! But...
- Stopping interrupts penalizes all processes when one process executes a critical section
  - Stops all I/O activity
  - 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

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

Operations On Semaphores

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

Key Uses Of Counting Semaphores

- Two basic paradigms
  - Cooperative mutual exclusion
  - Direct synchronization (e.g., producer-consumer)

Mutual Exclusion With Semaphores

- Initialize: create a mutex semaphore

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

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

```
 wait(sid);
...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

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

```
psem = semcreate(buffer-size);
csem = semcreate(0);
```

- Producer algorithm

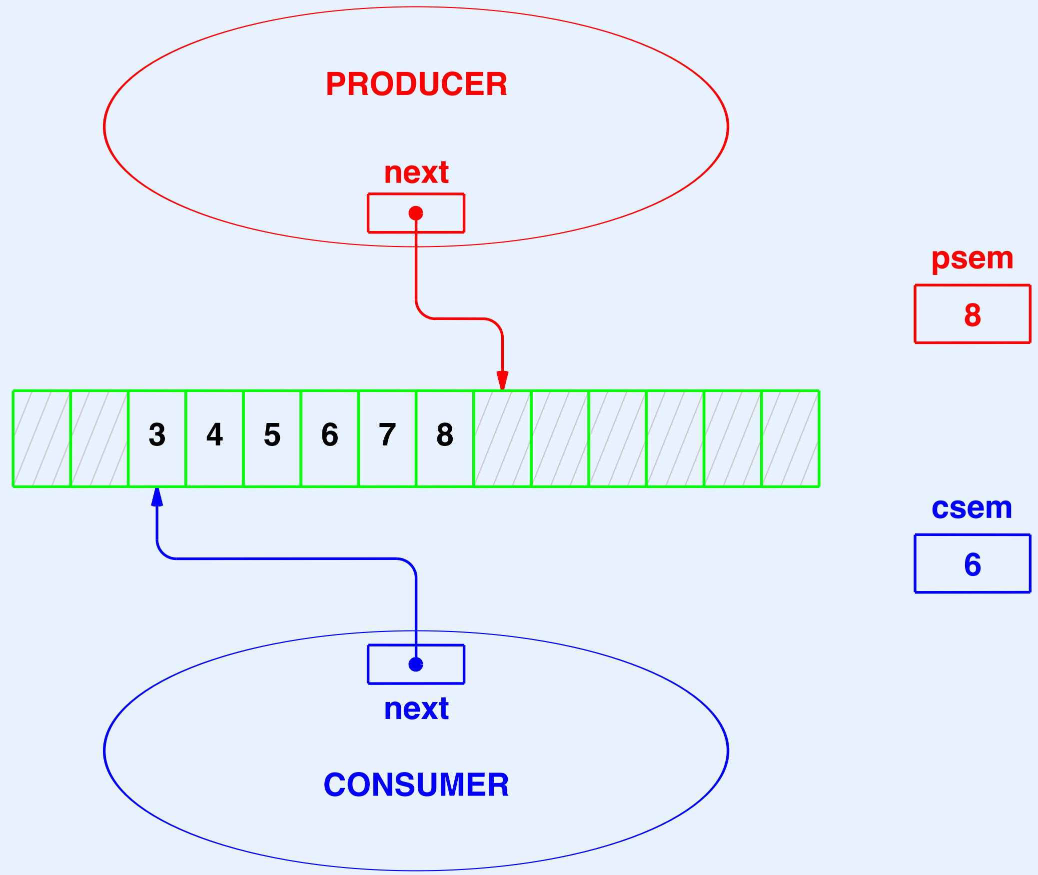
```
repeat forever {
        wait(psem);
        fill_next_buffer_slot;
        signal(csem);
}
```

Producer-Consumer Synchronization
(continued)

- Consumer algorithm

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

Interpretation Of Producer-Consumer Semaphores



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

Semaphore Invariant

Semaphore Invariant

- Establishes relationship between semaphore concept and implementation

Semaphore Invariant

- Establishes relationship between semaphore concept and implementation
- Makes code easy to create and understand

Semaphore Invariant

- Establishes relationship between semaphore concept and implementation
- Makes code easy to create and understand
- Must be re-established after each operation

Semaphore Invariant

- Establishes relationship between semaphore concept and implementation
- Makes code easy to create and understand
- Must be re-established after each operation
- 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.

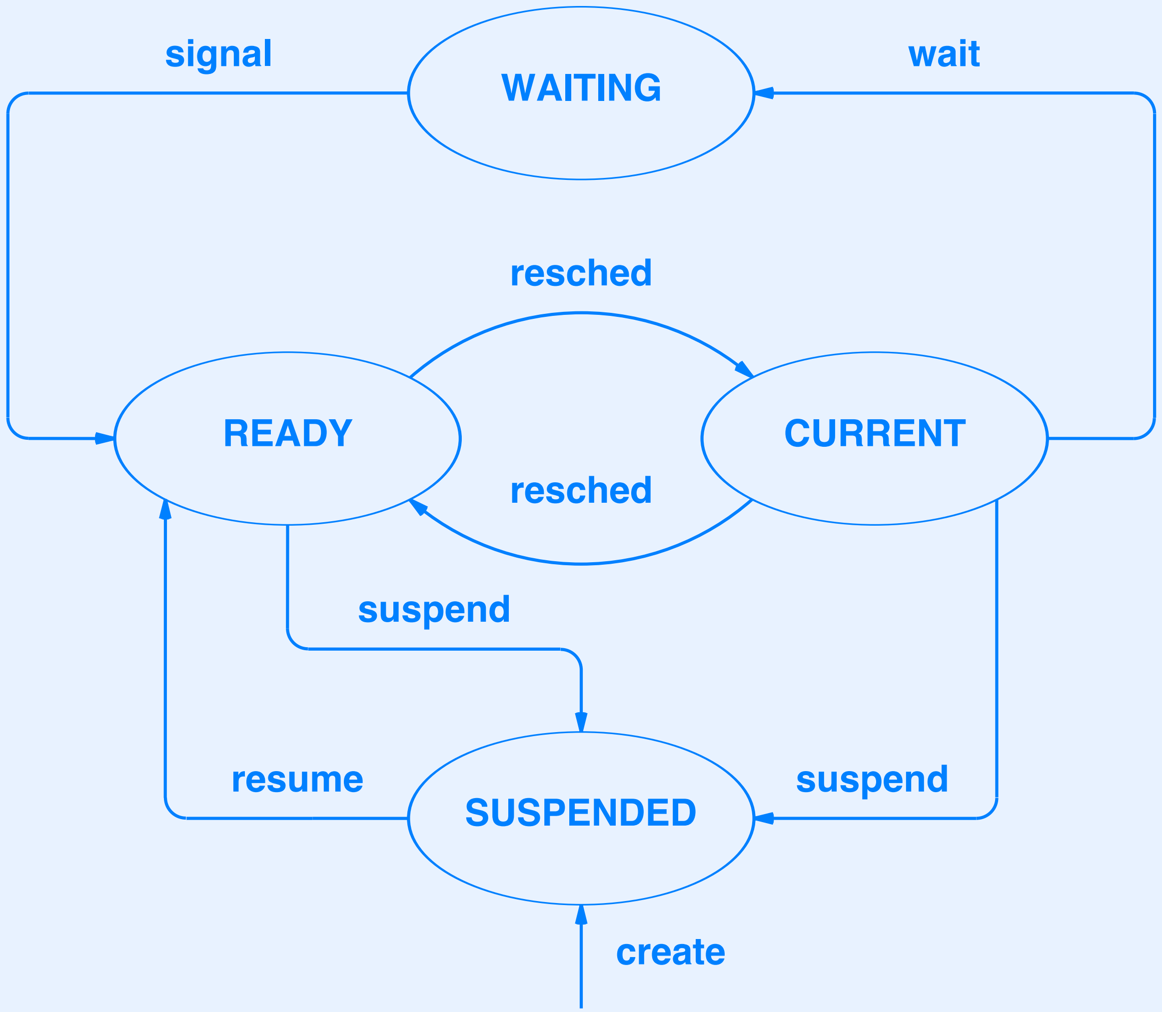
Counting Semaphores In Xinu

- Stored in an array of semaphore entries
- Each entry
  - Corresponds to one instance (one semaphore)
  - Contains an integer count and pointer to list of processes
- Semaphore ID is index into array
- Policy for management of waiting processes is FIFO

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*
- Therefore a new state is needed
- We will uses the *WAITING* state for a process blocked 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;
}
```

Semaphore Queuing Policy

- Determines which process to select among those waiting
- Needed when *signal* called
- Examples
  - First-Come-First-Served (FCFS or FIFO)
  - Process priority
  - Random

Question

Question

- The goal is “fairness”

Question

- The goal is “fairness”
- Which semaphore queuing policy implements goal best?

Question

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

Question

- The goal is “fairness”
- Which semaphore queuing policy implements goal best?
- In other words, how should we interpret fairness?
- Semaphore policy can interact with scheduling policy

Question

- The goal is "fairness"
- Which semaphore queuing policy implements goal best?
- 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 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
- Interactions of heuristic policies may produce unexpected results

Semaphore Queuing Policy In Xinu

- First-come-first-serve
- Straightforward to implement
- Extremely efficient
- Works well for traditional uses of semaphores
- Potential problem: low-priority process can access a resource while a high-priority process remains blocked

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
- Example implementation 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;
}
```

Semaphore Allocation

- Static
  - Semaphores are defined at compile time
  - More efficient, but less powerful
- Dynamic
  - Semaphores are created at runtime
  - More flexible
- Xinu supports dynamic allocation

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 semaphore is deleted
- Must choose a disposition for each
- Xinu policy: 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;
```

Xinu Semdelete (part 2)

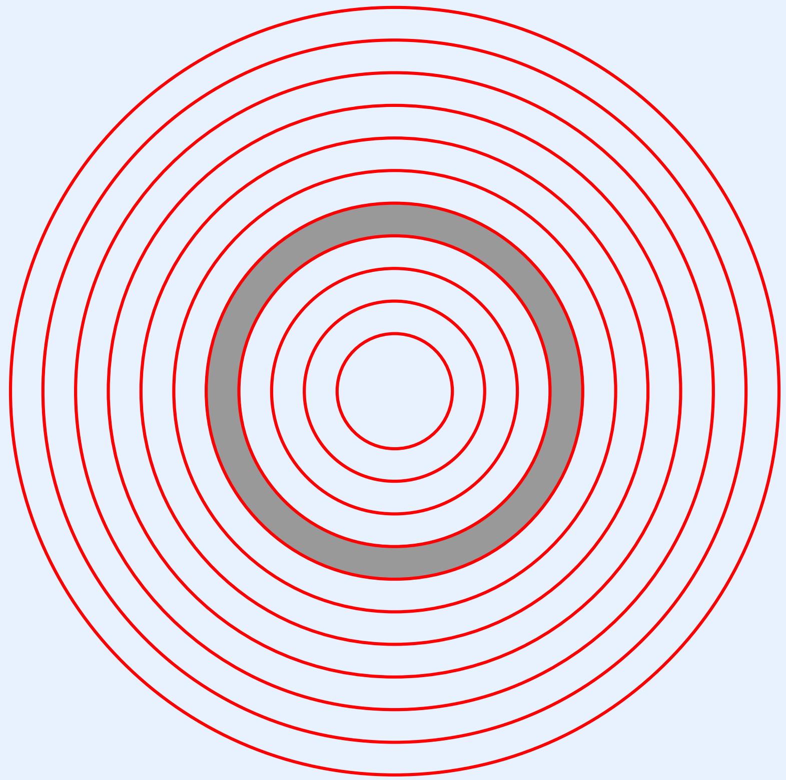
```
        resched_cntl(DEFER_START);
        while (semptr->scount++ < 0) {  /* Free all waiting processes   */
                ready(getfirst(semptr->squeue));
        }
        resched_cntl(DEFER_STOP);
        restore(mask);
        return OK;
}
```

Do you understand semaphores?

Module VI

Inter-Process Communication

Location Of Inter-process Communication In The Hierarchy



Inter-process Communication

- Used for
  - Exchange of (nonshared) data
  - Process coordination
- General technique: *message passing*

Two Approaches To Message Passing

- Approach #1
  - Message passing is one of many services
  - Messages are separate from I/O and process synchronization services
  - Messages implemented using lower-level mechanisms, such as semaphores
- Approach #2
  - The entire operating system is *message-based*
  - Messages, not function calls, provide the fundamental building block
  - Messages, not semaphores, used for process synchronization

Design Of A Message Passing Facility

Design Of A Message Passing Facility

- To understand the issues, we will begin with a trivial message passing facility

Design Of A Message Passing Facility

- To understand the issues, we will begin with a trivial message passing facility
- We want to allow a process to send a message directly to another process

Design Of A Message Passing Facility

- To understand the issues, we will begin with a trivial message passing facility
- We want to allow a process to send a message directly to another process
- In principle,

Design Of A Message Passing Facility

- To understand the issues, we will begin with a trivial message passing facility
- We want to allow a process to send a message directly to another process
- In principle, the design should be straightforward

Design Of A Message Passing Facility

- To understand the issues, we will begin with a trivial message passing facility
- We want to allow a process to send a message directly to another process
- In principle, the design should be straightforward
- In practice,

Design Of A Message Passing Facility

- To understand the issues, we will begin with a trivial message passing facility
- We want to allow a process to send a message directly to another process
- In principle, the design should be straightforward
- In practice, many design decisions arise

Message Passing Design Decisions

- Are messages fixed or variable size?
- What is the maximum message size?
- How many messages can be outstanding at a given time?
- Where are messages stored?
- How is a recipient specified?
- Does a receiver know the sender's identity?
- Are replies supported?
- Is the interface synchronous or asynchronous?

Synchronous vs. Asynchronous Interface

- Synchronous interface
  - Blocks until the operation is performed
  - Easy to understand / program
  - Extra processes can be used to obtain asynchrony

Synchronous vs. Asynchronous Interface
(continued)

- Asynchronous interface
  - Process starts an operation
  - Initiating process continues execution
  - Notification arrives when operation completes
    * Happens at any time
    * May entail abnormal control flow (e.g., software interrupt or “callback” mechanism)
  - Polling can be used to determine status

Why Is A Message Passing Facility So Difficult To Design?

- Interacts with
  - Process coordination subsystem
  - Memory management subsystem
- Affects user’s perception of system

Example Inter-process Message Passing Design

- Simple, low-level mechanism
- Direct process-to-process communication
- One-word messages
- Message stored with receiver
- One-message buffer
- Synchronous, buffered reception
- Asynchronous transmission and “reset” operation

Example Inter-process Message Passing Design (continued)

- Three functions

```
send(msg, pid);

msg = receive();

msg = recvclr();
```

- Message stored in *receiver's* process table entry
- *Send* transmits message to specified process
- *Receive* blocks until a message arrives
- *Recvclr* removes existing message, if one has arrived, but does not block

Example Inter-process Message Passing Design (continued)

- First-message semantics
  - First message sent to a process is stored until it has been received
  - Subsequent attempts to send fail
- Idiom

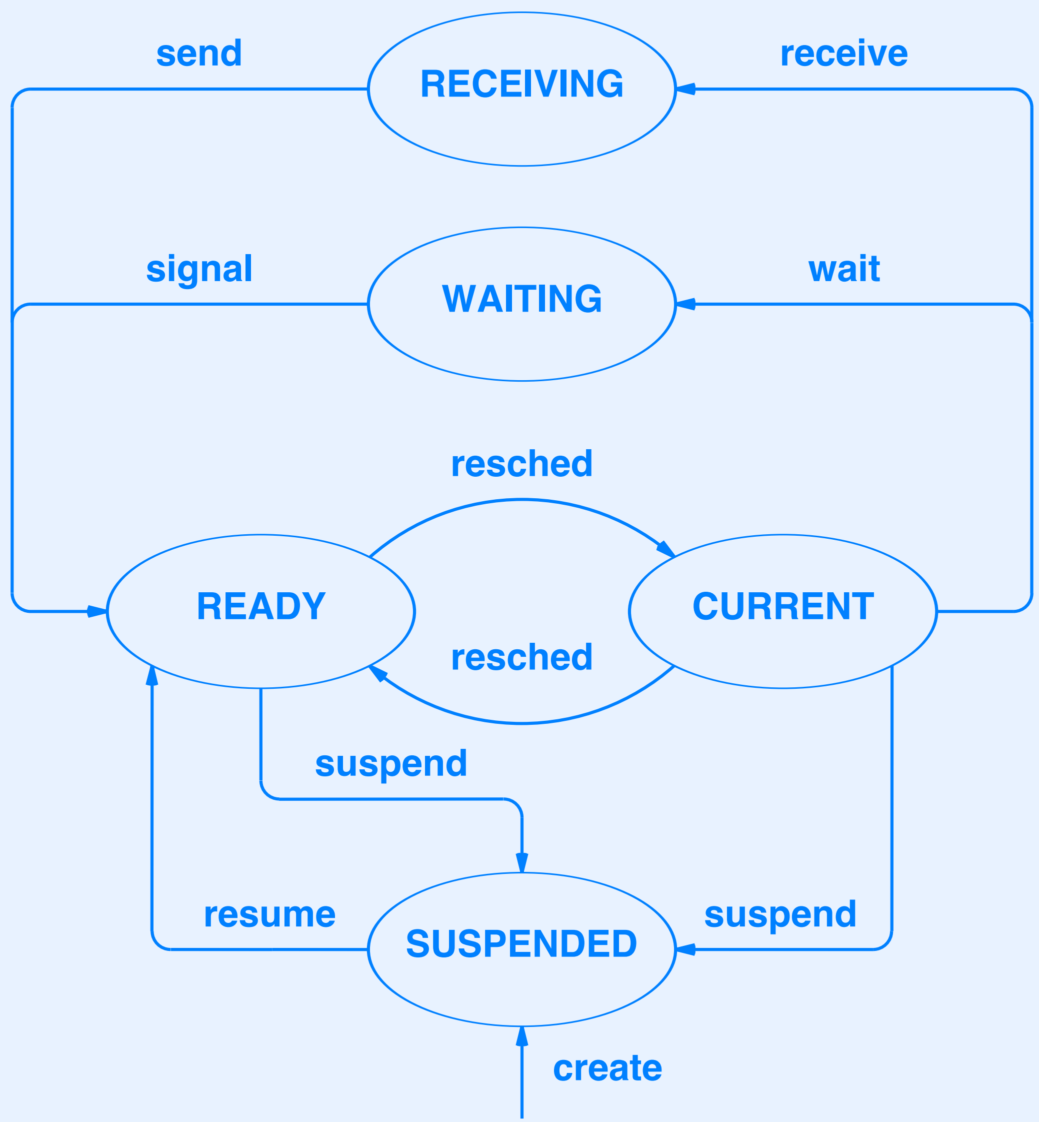
```
recvclr();  /* prepare to receive a message */
... /* allow other processes to send messages */
msg = receive();
```

- Above code returns first message that was sent, even if a high priority process attempts to send later

Process State For Message Reception

- While receiving a message, a process is not
  - Executing
  - Ready
  - Suspended
  - Waiting on a semaphore
- Therefore, a new state is needed for message passing
- Named *RECEIVING*
- Entered when *receive* called

State Transitions With Message Passing



Xinu Code For Message Reception

```
/* receive.c - receive */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  receive  -  Wait for a message and return the message to the caller
 *------------------------------------------------------------------------
 */
umsg32  receive(void)
{
        intmask mask;                   /* Saved interrupt mask         */
        struct  procent *prptr;         /* Ptr to process' table entry  */
        umsg32  msg;                    /* Message to return            */

        mask = disable();
        prptr = &proctab[currpid];
        if (prptr->prhasmsg == FALSE) {
                prptr->prstate = PR_RECV;
                resched();              /* Block until message arrives  */
        }
        msg = prptr->prmsg;             /* Retrieve message             */
        prptr->prhasmsg = FALSE;        /* Reset message flag           */
        restore(mask);
        return msg;
}
```

Xinu Code For Message Transmission (part 1)

```
/* send.c - send */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  send  -  Pass a message to a process and start recipient if waiting
 *------------------------------------------------------------------------
 */
syscall send(
          pid32           pid,              /* ID of recipient process      */
          umsg32          msg               /* Contents of message          */
        )
{
        intmask mask;                       /* Saved interrupt mask         */
        struct  procent *prptr;             /* Ptr to process' table entry  */

        mask = disable();
        if (isbadpid(pid)) {
                restore(mask);
                return SYSERR;
        }

        prptr = &proctab[pid];
        if ((prptr->prstate == PR_FREE) || prptr->prhasmsg) {
                restore(mask);
                return SYSERR;
        }
```

Xinu Code For Message Transmission (part 2)

```
        prptr->prmsg = msg;             /* Deliver message              */
        prptr->prhasmsg = TRUE;         /* Indicate message is waiting  */

        /* If recipient waiting or in timed-wait make it ready */

        if (prptr->prstate == PR_RECV) {
                ready(pid);
        } else if (prptr->prstate == PR_RECTIM) {
                unsleep(pid);
                ready(pid);
        }
        restore(mask);          /* Restore interrupts */
        return OK;
}
```

- Note: we will discuss receive-with-timeout later

Xinu Code For Clearing Messages

```
/* recvclr.c - recvclr */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  recvclr  -  Clear incoming message, and return message if one waiting
 *------------------------------------------------------------------------
 */
umsg32  recvclr(void)
{
        intmask mask;                   /* Saved interrupt mask         */
        struct  procent *prptr;         /* Ptr to process' table entry  */
        umsg32  msg;                    /* Message to return            */

        mask = disable();
        prptr = &proctab[currpid];
        if (prptr->prhasmsg == TRUE) {
                msg = prptr->prmsg;     /* Retrieve message             */
                prptr->prhasmsg = FALSE;/* Reset message flag           */
        } else {
                msg = OK;
        }
        restore(mask);
        return msg;
}
```

Summary

- Inter-process communication
  - Implemented by message passing
  - Can be synchronous or asynchronous
- Synchronous interface is the simplest
- Xinu uses synchronous reception and asynchronous transmission

Questions?