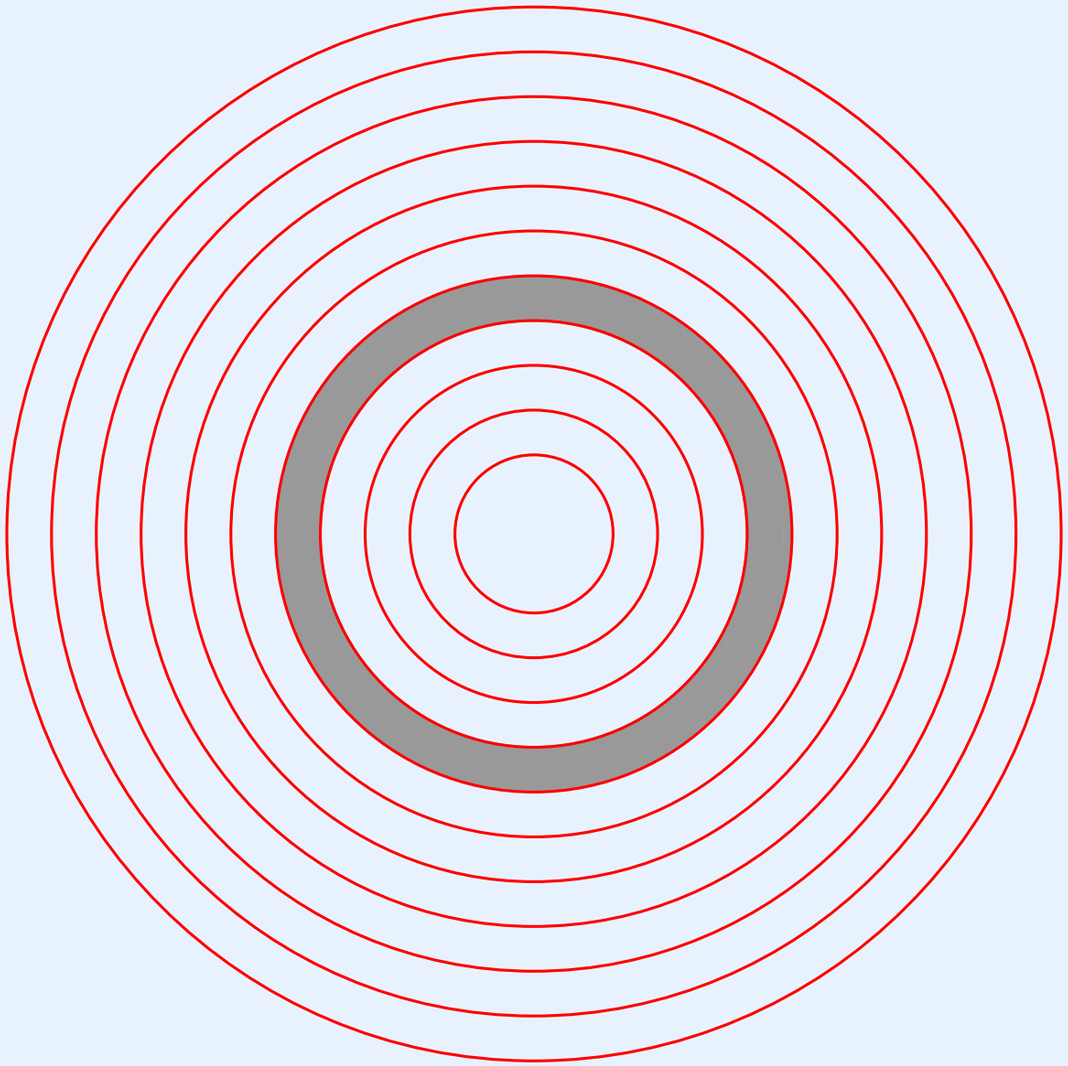
Inter-Process Communication (Message Passing)

Location Of Inter-Process Communication In The Hierarchy



Inter-Process Communication

- Can be used for
  - Exchange of (nonshared) data among processes
  - Some forms of process coordination
- The general technique is known as *message passing*

Two Approaches To Message Passing

- Approach #1
  - Message passing is one of many services the operating system offers
  - Messages are basically data items sent from one process to another, and are independent of both normal I/O and process synchronization services
  - Message passing functions are implemented using lower-level mechanisms
- Approach #2
  - The entire operating system is *message-based*
  - Messages, not function calls, provide the fundamental building block
  - Messages are used to coordinate and control processes
- Note: a few research projects used approach #2, but most systems use approach #1

An Example Design For A Message Passing Facility

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

An Example Design For A Message Passing Facility

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

An Example Design For A Message Passing Facility

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

- A synchronous interface
  - An operation blocks until the operation is performed
  - A sending process is blocked until the recipient accepts the message being sent
  - A receiving process is blocked until a message arrives
  - Is easy to understand and use
  - A programmer can create extra processes to obtain asynchrony

Synchronous vs. Asynchronous Interface (continued)

- An asynchronous interface
  - A process starts an operation
  - The initiating process continues execution
  - A notification arrives when the operation completes
    * The notification can arrive at any time
    * Typically, notification entails abnormal control flow (e.g., "callback" mechanism)
  - Is more difficult to understand and use
  - Polling can be used to determine the status

Why Message Passing Choices Are Difficult

- Message passing interacts with scheduling
  - Process *A* sends a message to process *B*
  - Process *B* does not check messages
  - Process *C* sends a message to process *B*
  - Process *B* eventually checks its messages
  - If process *C* has higher priority than *A*, should *B* receive the message from *C* first? (NO!!!, Xinu follows FIFO, a msg list per receiver)
- Message passing affects memory usage
  - If messages are stored with a receiver, senders can use up all the receiver's memory by flooding the receiver with messages
  - If messages are stored with a sender, receivers can use up all the sender's memory by not accepting messages

An Example Message Passing Facility

- We will examine a basic, low-level mechanism
- The facility provides direct process-to-process communication
- Each message is one word (e.g., an integer)
- A message is stored with the receiving process
- A process only has a one-message buffer
- Message reception is synchronous and buffered
- Message transmission is asynchronous
- The facility includes a “reset” operation

An Example Message Passing Facility (continued)

- The interface consists of three system calls

```
send(pid, msg); to whom and what; send is non-blocking, so if message buffer is
full, we will immediately return and indicate it's full
msg = receive(); //blocking
msg = recvclr(); //non-blocking
```

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

An Example Message Passing Facility
(continued)

- The system uses "first-message" semantics
  - The first message sent to a process is stored until it has been received
  - Subsequent attempts to send to the process fail

How To Use First-Message Semantics

- The idea: wait for one of several events to occur
- Example events
  - I/O completes
  - A user presses a key
  - Data arrives over a network
  - A hardware indicator signals a low battery
- To use message passing facility to wait for the first event
  - Create a process for each event
  - When the process detects its event, have it send a message

How To Use First-Message Semantics
(continued)

- The idiom a receiver uses to identify the first event that occurs

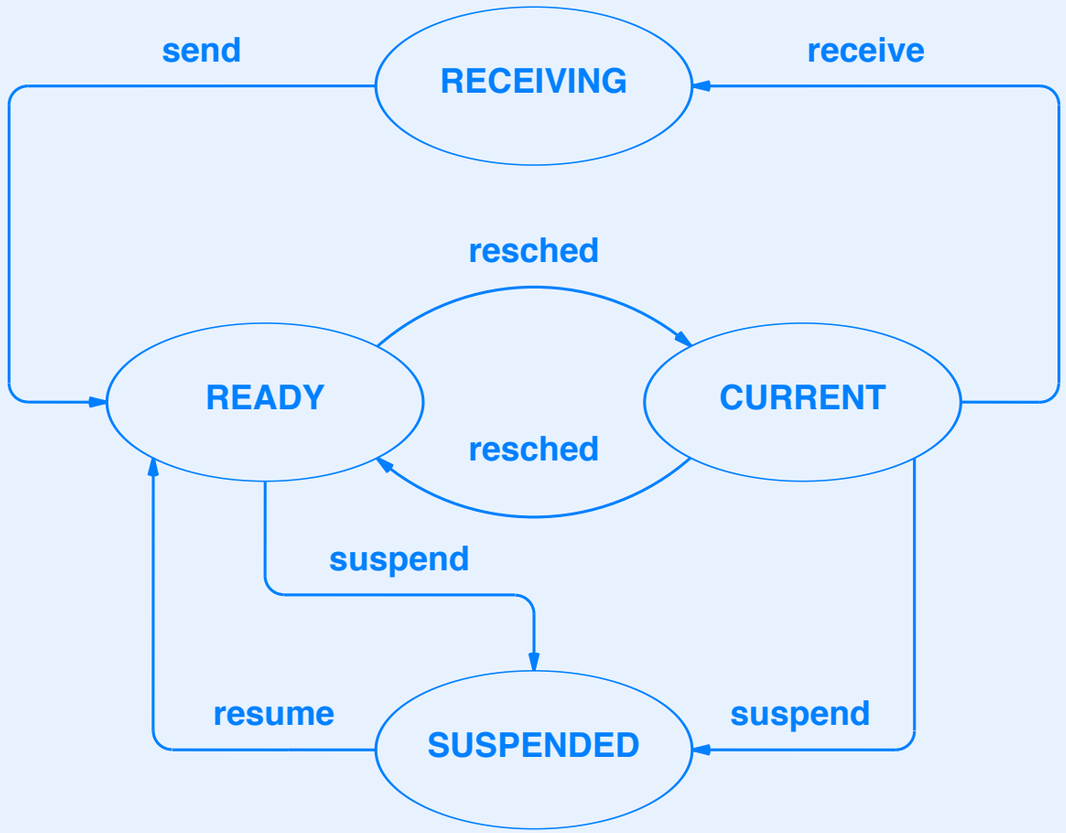
```
recvclr();  /* prepare to receive a message */
... /* allow other processes to send messages */
msg = receive();
```

- The above code returns first message that is sent, even if a higher priority process attempts to send later
- The receiver will block until a message arrives

A Process State For Message Reception

- While receiving a message, a process is not
  - Executing
  - Ready
  - Suspended
- Therefore, a new state is needed for message passing
- The state is named *RECEIVING*
- The state is entered when *receive* called
- The code uses constant *PR_RECV* to denote a *receiving* state

State Transitions With Message Passing



The Steps Taken To Receive A Message

- The current process calls *receive* which checks its own process table entry
- If no message has arrived, *receive* blocks the calling process to wait until a message to arrive
- Once this step has been reached, a message is present
- *Receive* extracts a copy of the message from the process table entry and resets the process table entry to indicate that no message is present
- *Receive* returns the message to its caller

Blocking To Wait For A Message

- We have seen how a process suspends itself
- Blocking to receive a message is almost the same
  - Find the current process's entry in the process table, *proctab[currpid]*
  - Set the state in the process table entry to *PR_RECV*, indicating that the process will be receiving
  - Call *resched*

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) if a process does't have msg yet, make its state to receiving
{                     if a process has a msg in buffer, return the msg
        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; why we store msg to local var instead of return prptr->prmsg?
}                   because we restore mask before return, if resched happens
                    right after restore, another process overwrites the
                    msg(because it sees prhasmsg is false), then we can be
                    returning the wrong msg.
```

Message Transmission

- To send a message, a process calls *send* specifying a destination process and a message to send to the process
- The code
  - Checks arguments
  - Returns an error if the process already has a message waiting
  - Deposits the message
  - Makes the process ready if it is in the receiving state
- Note: the code also handles a receive-with-timeout state, but we will consider that state later

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
 *------------------------------------------------------------------------
 */            if the dest proc's prhasmsg is true, return syserr
syscall send(  if dest proc doesn't have msg, write the msg and set prhasmsg to true
          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 dest proc is in blocking-receive state, make it
        if (prptr->prstate == PR_RECV) {  ready (ready will run resched)
                ready(pid);
        } else if (prptr->prstate == PR_RECTIM) {
                unsleep(pid);
                ready(pid);
        }
        restore(mask);           /* Restore interrupts */
        return OK;
}
```

Xinu Code For Clearing Messages

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

#include <xinu.h>

/*------------------------------------------------------------------------
 *  recvclr  -  Clear incoming message, and return message if one waiting
 *------------------------------------------------------------------------
 */                  non-blocking version of receive.
umsg32  recvclr(void) return immediately (even if buffer is empty)
{
        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 Of Message Passing

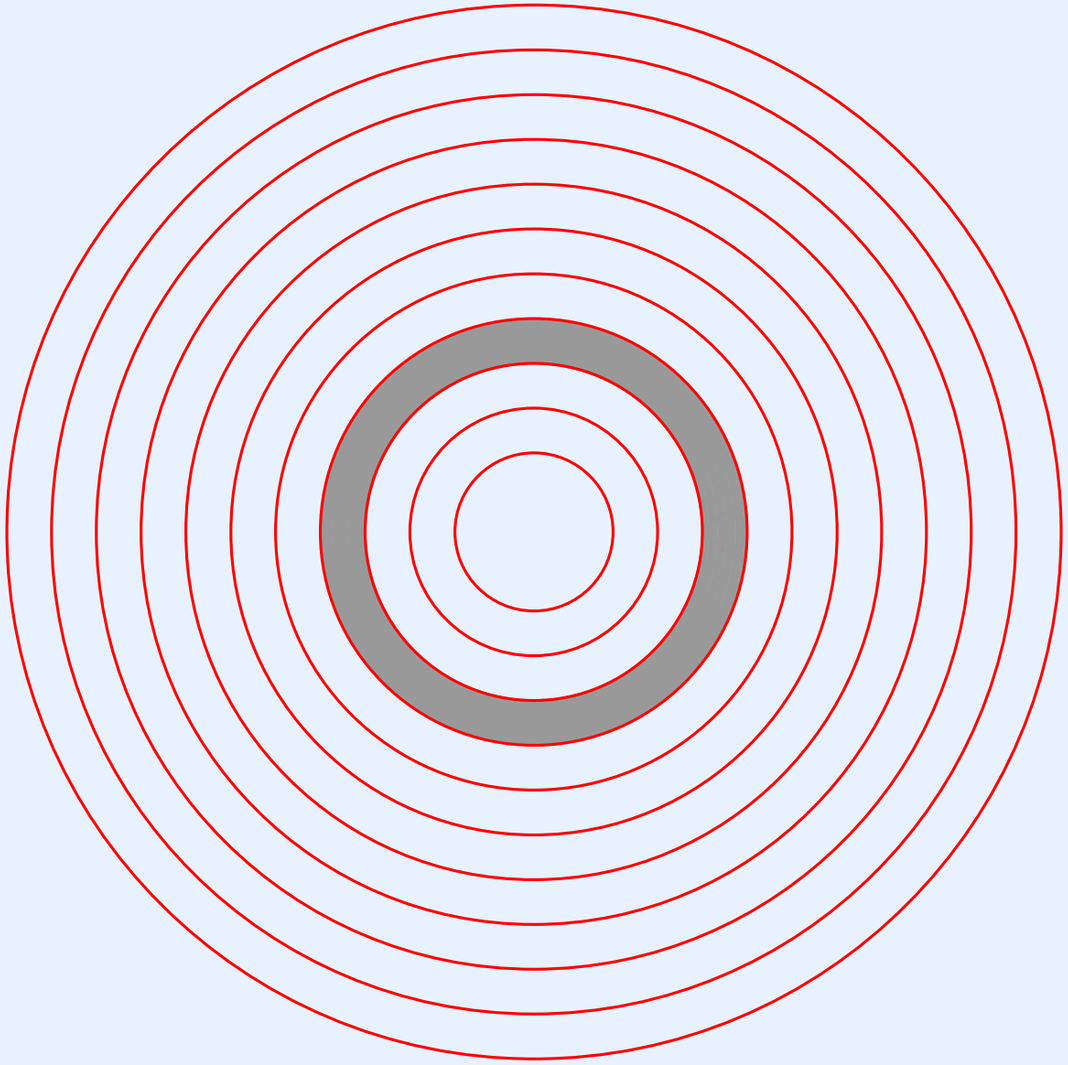
- Message passing offers an inter-process communication system
- The interface can be synchronous or asynchronous
- A synchronous interface is the easiest to use
- Xinu uses synchronous reception and asynchronous transmission
- An asynchronous operation allows a process to clear any existing message without blocking
- The Xinu message passing system only allows one outstanding message per process, and uses first-message semantics


Questions?

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 must block until space is available
  - When the buffer is empty, a consumer must block until an item has been deposited
- A given process may act as a consumer for one buffer and a producer for another
- Example: a Unix pipeline

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

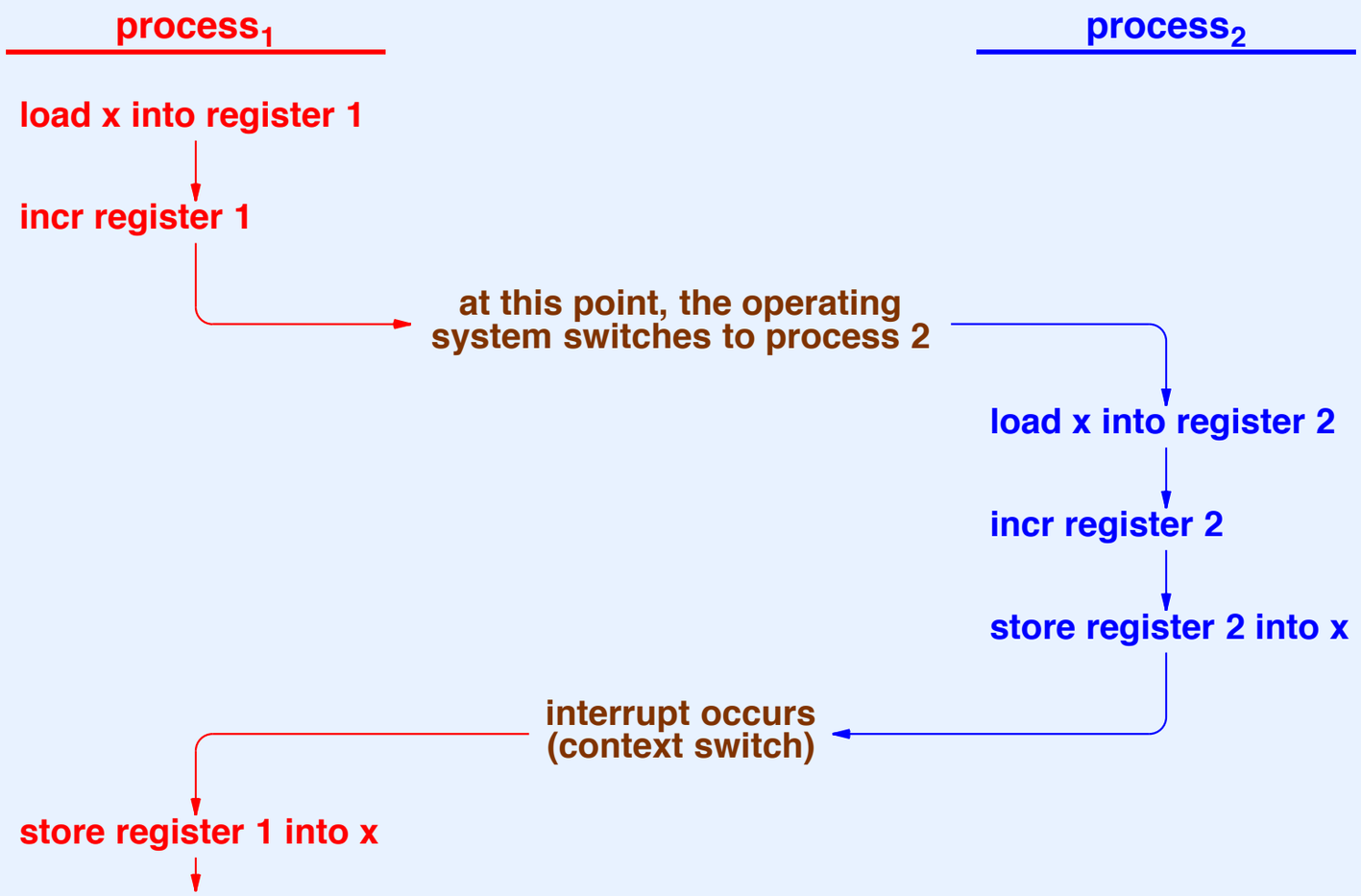
Mutual Exclusion

- 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)
- Programmers must take steps to ensure that when a sequence of non-atomic operations is used to change a data item, the sequence is not executed concurrently
- Even trivial operations can be non-atomic!

The Classic Example Of A Concurrent Access Problem

- Consider an integer, *x*
- To increment *x*, a programmer writes *x*++
- On most hardware architectures. three instructions are required
  - Load variable *x* into a register
  - Add 1 to the register
  - Store the register into variable *x*
- An operating system can switch from one process to another between any two instructions
- Surprising consequence: if two processes attempt to increment a shared integer concurrently, errors can result

Illustration Of What Can Happen When Two Processes Attempt To Increment Integer x Concurrently



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 spin lock instructions
  - Hardware mechanisms that disable and restore interrupts
  - Semaphores (implemented in software)

Handling Mutual Exclusion With Spin Locks

- Used in multicore CPUs; does *not* work for a single processor
- An atomic hardware operation allows a core to test a memory location and change it
- The hardware guarantees that only one core will be allowed to make the change
- It is called a *spin lock* mechanism 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 is also known as *test-and-set*

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 function
  - Access a given operating system component
- 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 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 current interrupt status
  - Disables interrupts
  - Proceeds through a critical section
  - *Restores* the interrupt status from the 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
  - It stops all I/O activity
  - It restricts execution to one process for the entire system
- Disabling interrupts can interfere with the scheduling invariant (e.g., a low-priority process can block 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

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 access each critical section independent of other sections
  - Provide an access policy that specifies how waiting processes gain access (e.g., FIFO)
- Good news: a single mechanism, the *counting semaphore*, suffices

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

P1 writes and P2 reads from the FIFO buffer chat a[6]
once P1 write, it writes to the next available slot and update FP(first pointer) to the next slot
once P2 read, it reads the next occupied data and update LP to next slot.
Read and Write is not atomic:
If FP and LP meets, then there is a possibility that P1 was ctxsw before write, so P2 read the old (wrong) data.
there is a possibility that P2 was ctxsw before read, so P1 overwrite the data that P2 wants to read
What can we do?
Two semaphore
psem (how much free space is there)
csem (how much data is there)

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, and then the two processes run

```
sidx = semcreate(1);        sidy = semcreate(1);
```

- The processes take the following steps

```
/* Process 1 */
...
wait(sidx);
start to modify x
wait(sidy);
modify y
signal(sidy);
finish modifying x
signal(sidx);
```

A Potential Problem: Deadlock

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

```
sidx = semcreate(1);        sidy = semcreate(1);
```

- The processes take the following steps

```
/* Process 2 */
...
wait(sidy);
start to modify y
wait(sidx);
modify x
signal(sidx);
finish modifying y
signal(sidy);
```

A Potential Problem: Deadlock

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

```
sidx = semcreate(1);        sidy = semcreate(1);
```

- The processes take the following steps

```
/* Process 1 */
...
wait(sidx);
start to modify x
wait(sidy);
modify y
signal(sidy);
finish modifying x
signal(sidx);
```

```
/* Process 2 */
...
wait(sidy);
start to modify y
wait(sidx);
modify x
signal(sidx);
finish modifying y
signal(sidy);
```

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

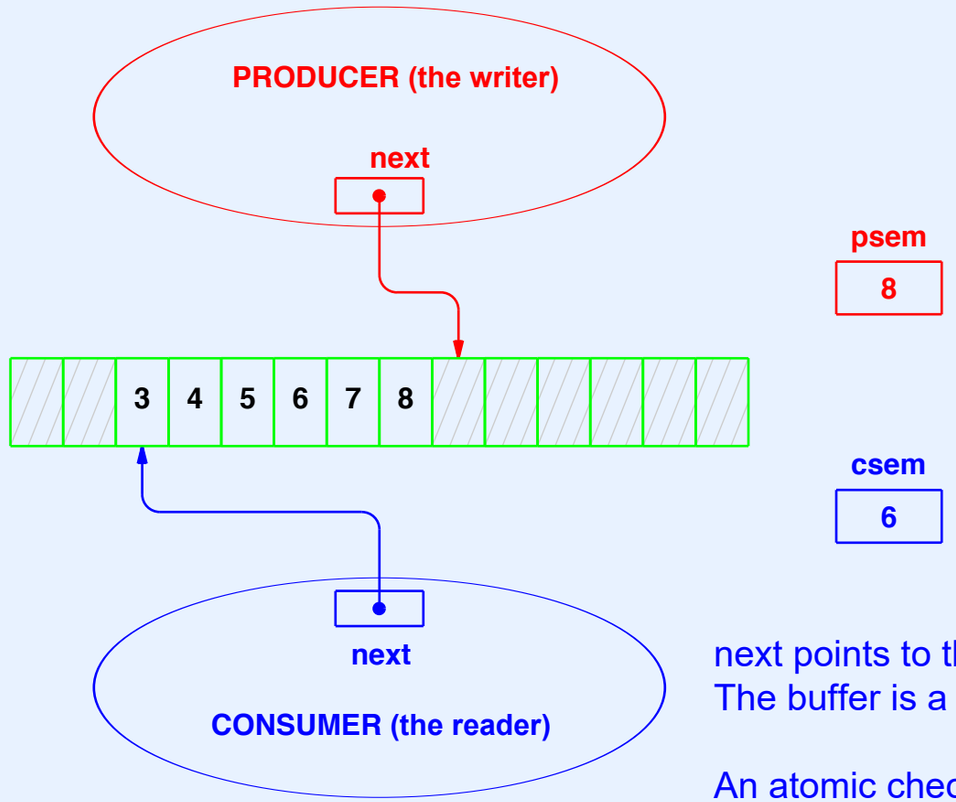
```
repeat forever {
        generate an item to be added to the buffer;
        wait(psem); //wait decrements psem, if psem
        is not negative, it returns(doesn't block)
        fill_next_buffer_slot;//CS
        signal(csem);  //signal increments csem, and
}       if csem is non-negative, it checks if there
        is a process being blocked on csem.
        if so, make it ready
```

Producer-Consumer Synchronization With Semaphores (continued)

- The consumer algorithm

```
repeat forever {
        wait(csem); //wait decrements csem, if csem is
        negative, it blocks and context-switched out.
        Once producer signal(csem) and make this blocking
        process ready, it can run the rest.
}       extract_from_buffer_slot;
        handle the item;
        signal(psem);
```

An Interpretation Of Producer-Consumer Semaphores



next points to the next slot to operate on
The buffer is a circular buffer

An atomic check: If there is a positive gap between producer's next and consumer's next, this means that there is no overwrite, it's safe.
Introducing: psem and csem

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

The Semaphore Invariant

The Semaphore Invariant

- Establishes a relationship between the semaphore concept and its implementation

The Semaphore Invariant

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

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

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.

Lab 4: bsend()

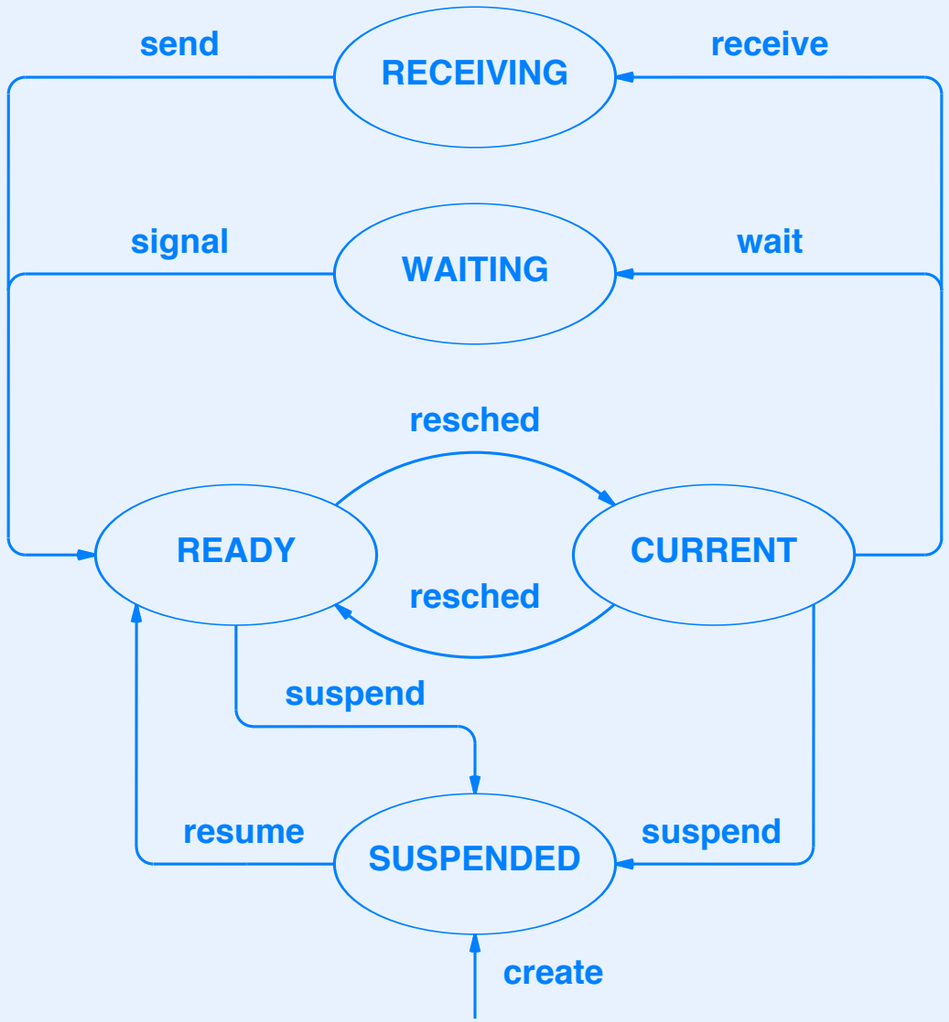
Counting Semaphores In Xinu

- 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

Semaphore Policy Consequences

Semaphore Policy Consequences

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

Semaphore Policy Consequences

- 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

- First-come-first-serve
- Has several advantages
  - Is straightforward to implement
  - Is extremely efficient
  - Works well for traditional uses of semaphores
  - Guarantees all contending processes will obtain access
- Has an interesting consequence: a low-priority process can access a resource while a high-priority process remains blocked

Implementation Of A FIFO Semaphore Policy

- Each semaphore uses a list to manage waiting processes
- As we have seen Xinu manages the list of processes as a queue
  - Wait enqueues a process at one end of the queue
  - Signal chooses a process at the other end 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 Allocation 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 allocation, but preallocates the data structure to achieve efficiency

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

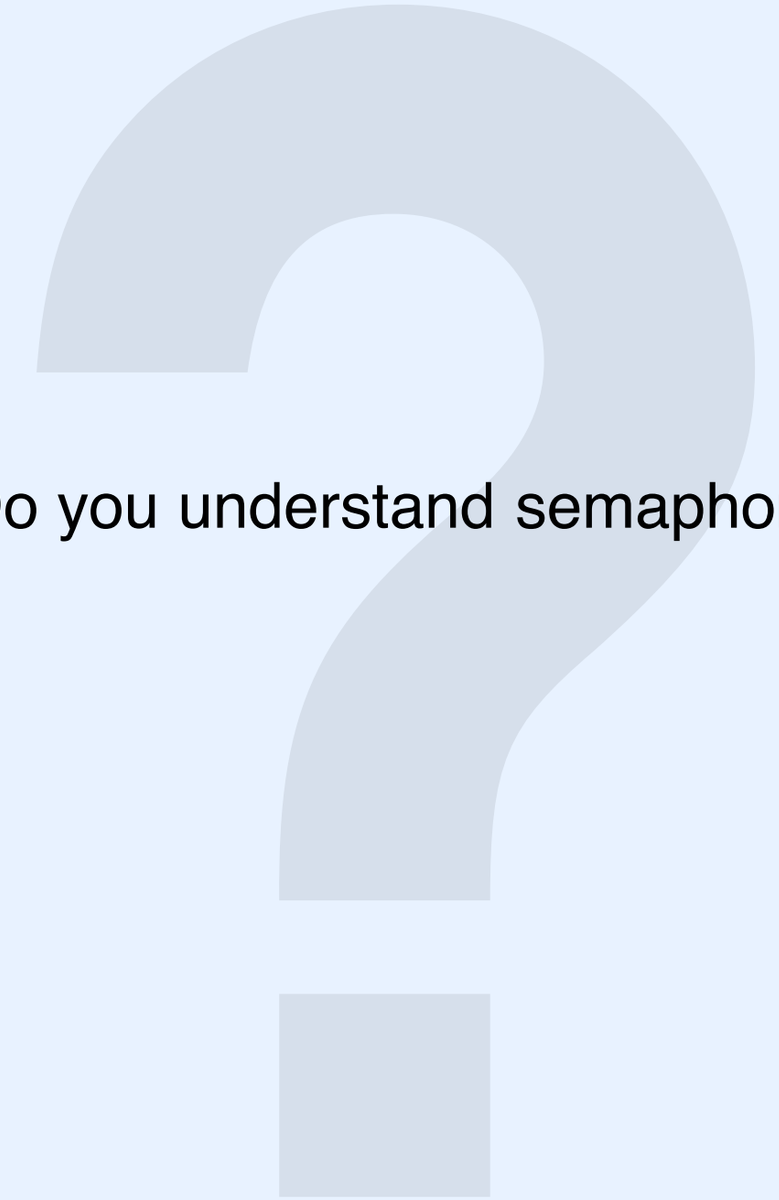
Xinu Semdelete (Part 1)

```
/* 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);    resched_cntl makes sure that we can ready a group of
        restore(mask);               processes and then call resched
        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?

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?