Module III

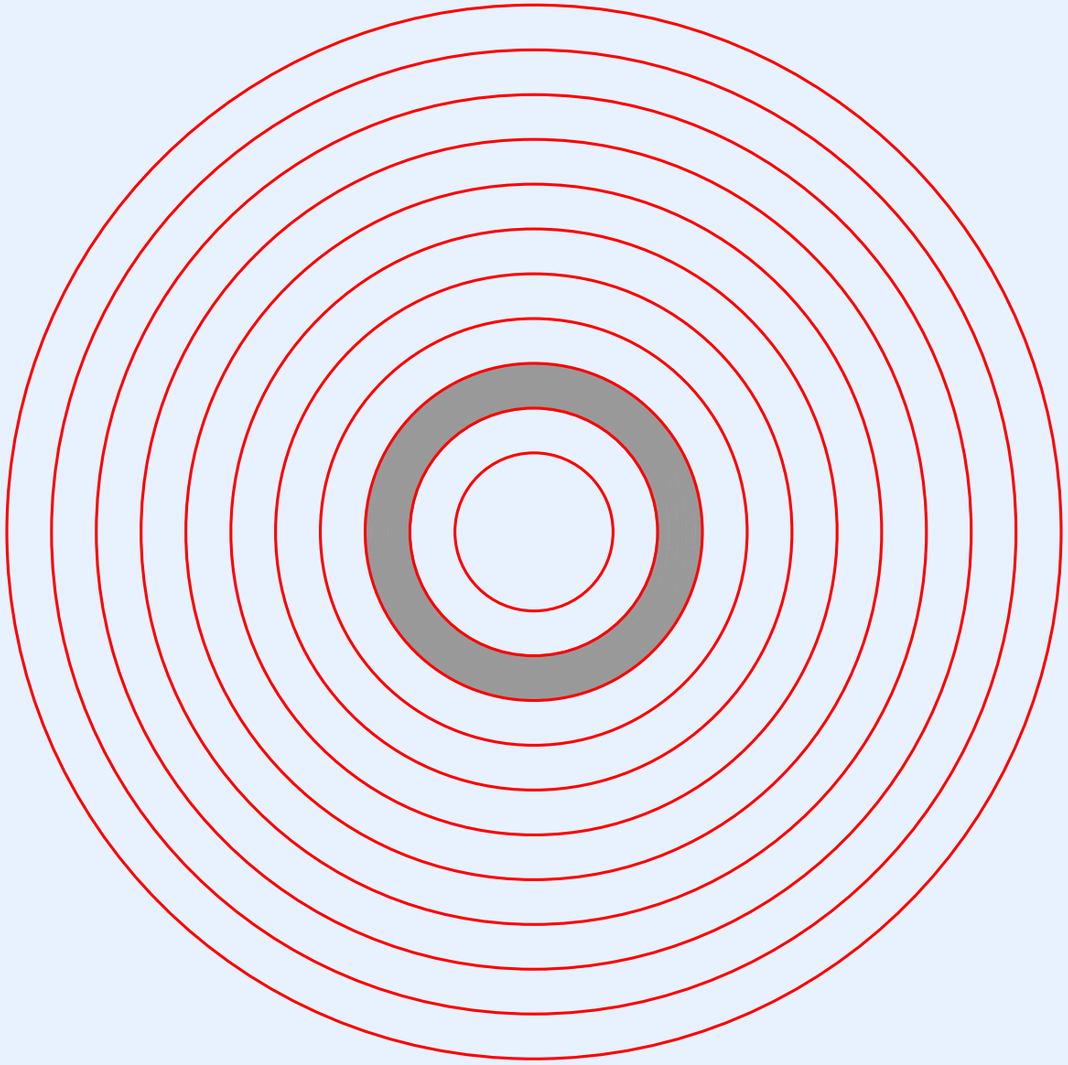
More Process Management: Process Suspension/Resumption And Inter-Process Communication

Process Manipulation

- An OS needs system calls that can be used to control processes
- Example operations
  - Suspend a process (keep it from running)
  - Resume a previously-suspended process
  - Block a process to receive a message from another process
  - Send a message to another process
- The OS uses the process state variable to record the status of the process

Process Suspension And Resumption

Location Of Process Suspension And Resumption In The Hierarchy



Process Suspension And Resumption

- The idea
  - Temporarily “stop” a process
  - Allow the process to be resumed later
- Questions
  - What happens to the process while it is suspended?
  - Can and process be suspended at any time?
  - What happens if an attempt is made to resume a process that is not suspended?

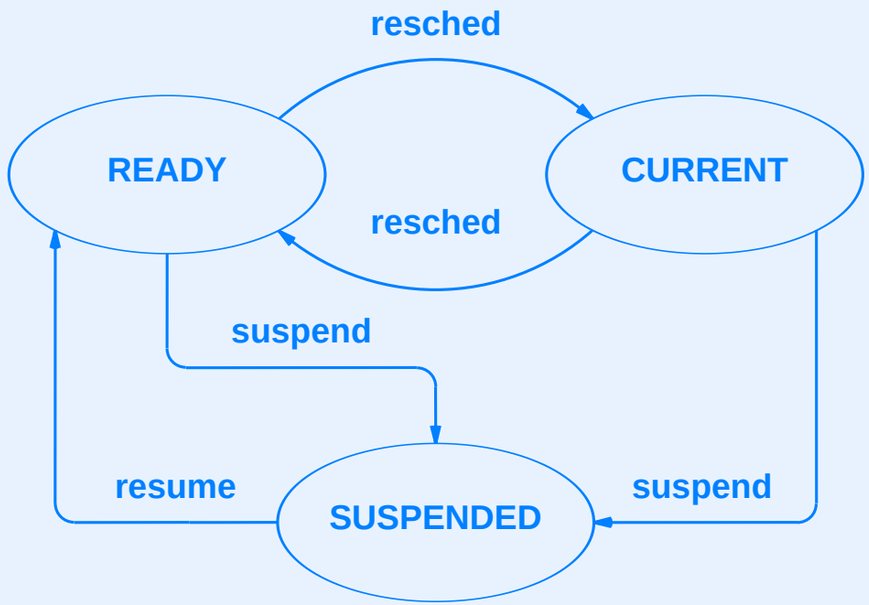
Steps In Suspension And Resumption

- Suspending a process simply means prohibiting the process from using the processor
- When suspending, the operating system must
  - Save pertinent information about the state of the process, such as where it is executing, the contents of general purpose registers, etc.
  - Set the state variable in the process table entry to indicate that the process is suspended
- When resuming, the operating system must
  - Allow the process to use the processor once again
  - Change the state to indicate that process is eligible

A State For Suspended Processes

- A suspended process is not ready, nor is it current
- Therefore, a new process state is needed
- The code uses constant *PR_SUSP* to indicate that a process is in the suspended state

State Transitions For Suspension And Resumption



- As the diagram shows, only a current or ready process can be suspended
- Only a suspended process can be resumed
- System calls *suspend* and *resume* handle the transitions

Suspended Processes

- Where is a process kept when it is suspended?
- Answer:
  - Unlike ready processes, there is no list of suspended processes
  - However, information about a suspended process remains in the process table
  - The process's stack remains allocated in memory

Suspending One's Self

- The currently executing process can suspend itself!
- Self-suspension is straightforward
- The current process
  - Finds its entry in the process table, *proctab[currpid]*
  - Sets the state in its process table entry to *PR_SUSP*, indicating that it should be suspended
  - Calls *resched* to reschedule to another process

A Note About System Calls

- An operating system contains many functions
- OS functions can be divided into two basic categories
  - Some functions are defined to be *system calls*, which means that applications can call them to access services
  - Other functions are merely internal functions used by other operating system functions
- We use the type *syscall* to distinguish system calls
- Note: although Xinu does not prohibit applications from making direct calls to internal operating system functions, good programming practice restricts applications to system calls

Concurrent Execution Of System Calls

- Important concept: multiple processes can attempt to execute a given system call concurrently
- Concurrent execution can result in problems
  - Process A starts to change variables, such as process table entries
  - The OS switches to another process, B
  - When process B examines variables, they are inconsistent

Preventing Concurrent Execution By Disabling Interrupts

- To prevent other processes from changing global data structures, a system call function disables interrupts
- A later section of the course will explain interrupts; for now, it is sufficient to know that a system call must use two functions related to interrupts
  - Function *disable* is called to turn off hardware interrupts, and the function returns a mask value
  - Function *restore* takes as an argument a mask value that was previously obtained from *distable*, and sets the hardware interrupt status according to the specified mask
- Basically, a system call uses *disable* upon being called, and uses *restore* just before it returns
- Note that *restore* must be called before *any* return
- The next slide illustrates the general structure of a system call

A Template For System Calls

```
syscall function_name( args )  {
        intmask mask;           /* interrupt mask*/
        mask = disable( );      /* disable interrupts at start of function*/
        if ( args are incorrect ) {
                restore(mask); /* restore interrupts before error return*/
                return(SYSERR);
        }
        ... other processing ...
        if ( an error occurs ) {
                restore(mask); /* restore interrupts before error return*/
                return(SYSERR);
        }
        ... more processing ...
        restore(mask);          /* restore interrupts before normal return*/
        return( appropriate value );
}
```

The Suspend System Call (Part 1)

```
/* suspend.c - suspend */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  suspend  -  Suspend a process, placing it in hibernation
 *------------------------------------------------------------------------
 */
syscall suspend(
          pid32         pid             /* ID of process to suspend     */
        )
{
        intmask mask;                   /* Saved interrupt mask         */
        struct  procent *prptr;         /* Ptr to process' table entry  */
        pri16   prio;                   /* Priority to return           */

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

The Suspend System Call (Part 2)

```
        /* Only suspend a process that is current or ready */

        prptr = &proctab[pid];
        if ((prptr->prstate != PR_CURR) && (prptr->prstate != PR_READY)) {
                restore(mask);
                return SYSERR;
        }
        if (prptr->prstate == PR_READY) {
                getitem(pid);                   /* Remove a ready process   */
                                                /*   from the ready list    */
                prptr->prstate = PR_SUSP;
        } else {
                prptr->prstate = PR_SUSP;       /* Mark the current process */
                resched();                      /*   suspended and resched. */
        }
        prio = prptr->prprio;
        restore(mask);
        return prio;
}
```

Process Resumption

- The idea: resume execution of previously suspended process
- A detail: *resume* returns the priority of the resumed process
- Method
  - Make the process eligible to use the processor again
  - Re-establish scheduling invariant
- Steps
  - Move the suspended process back to the ready list
  - Change the state from *suspended* to *ready*
  - Call *resched*
- Note: resumption does *not* guarantee instantaneous execution of the resumed process

Moving A Process To The Ready List

- We will see that several system calls are needed to make a process ready
- To make it easy, Xinu includes an internal function named *ready* that makes a process ready
- *Ready* takes a process ID as an argument, and makes the process ready
- The steps are
  - Change the process's state to *PR_READY*
  - Insert the process onto the ready list
  - Ensure that the scheduling invariant is enforced

An Internal Function To Make A Process Ready

```
/* ready.c - ready */

#include <xinu.h>

qid16   readylist;                      /* Index of ready list          */

/*------------------------------------------------------------------------
 *  ready  -  Make a process eligible for CPU service
 *------------------------------------------------------------------------
 */
status  ready(
          pid32         pid             /* ID of process to make ready  */
        )
{
        register struct procent *prptr;

        if (isbadpid(pid)) {
                return SYSERR;
        }

        /* Set process state to indicate ready and add to ready list */

        prptr = &proctab[pid];
        prptr->prstate = PR_READY;
        insert(pid, readylist, prptr->prprio);
        resched();

        return OK;
}
```

Enforcing The Scheduling Invariant

- When a process is moved to the ready list, the process becomes eligible to use the processor again
- Recall that when the set of eligible processes changes, the scheduling invariant specifies that we must be check whether a new process should execute
- Consequence: after it moves a process to the ready list, *ready* must re-establish the scheduling invariant
- Surprising, *ready* does not explicitly check the scheduling invariant, but instead simply calls *resched*
- We can now appreciate the design of *resched*: if the newly ready process has a lower priority than the current process, *resched* returns without switching context, and the current process remains running

Example Resumption Code (Part 1)

```
/* resume.c - resume */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  resume  -  Unsuspend a process, making it ready
 *------------------------------------------------------------------------
 */
pri16   resume(
          pid32         pid             /* ID of process to unsuspend   */
        )
{
        intmask mask;                   /* Saved interrupt mask         */
        struct  procent *prptr;         /* Ptr to process' table entry  */
        pri16   prio;                   /* Priority to return           */

        mask = disable();
        if (isbadpid(pid)) {
                restore(mask);
                return (pri16)SYSERR;
        }
```

Example Resumption Code (Part 2)

```
        prptr = &proctab[pid];
        if (prptr->prstate != PR_SUSP) {
                restore(mask);
                return (pri16)SYSERR;
        }
        prio = prptr->prprio;           /* Record priority to return    */
        ready(pid);
        restore(mask);
        return prio;
}
```

- Consider the code for *resume* and *ready*
- By calling *ready*, *resume* does not need code to insert a process on the ready list, and by calling *resched*, *ready* does not need code to re-establish the scheduling invariant
- The point: choosing OS functions carefully means software at successive levels will be small and elegant

Keeping Processes On A List

- We have seen that suspended processes are not placed on any list
- Why not?
  - Function *resume* requires the caller to supply an argument that specifies the ID of the process to be resumed
  - We will see that no other operating system functions operate on suspended processes or handle the entire set of suspended processes
- Consequence: there is no reason to keep a list of suspended processes
- In general: an operating system only places a process on a list if a function needs to handle an entire set of processes that are in a given state (e.g., like the set of all ready processes that are handled by the scheduler)

Summary Of Process Suspension And Resumption

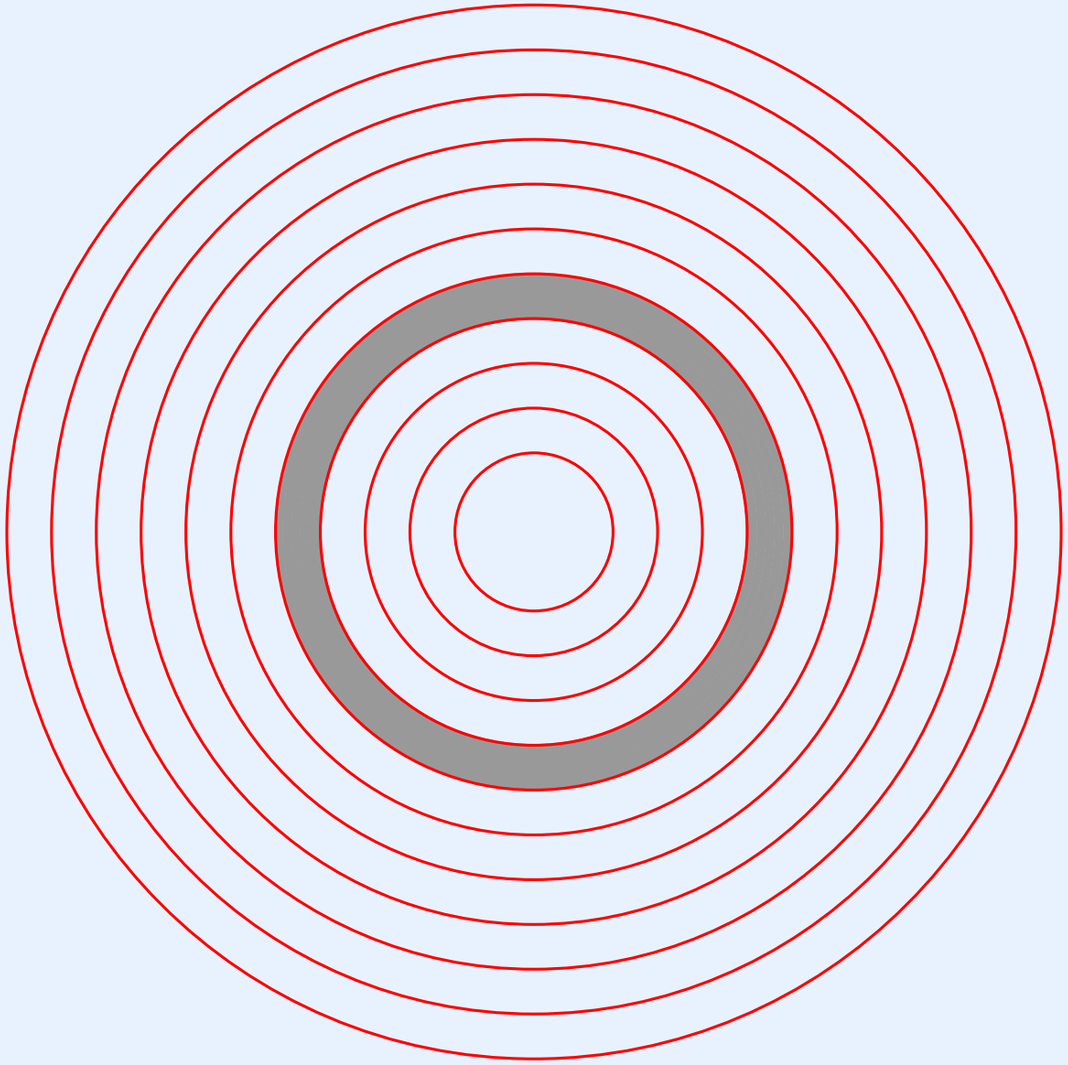
- An OS offers functions that can change a process's state
- Xinu allows a process to be
  - Suspended temporarily
  - Resumed later
- A state variable associated with each process records the process's current status
- When resuming a process, the scheduling invariant must be re-established

Something To Think About

- Resume returns the priority of the resumed process
- The code
  - Extracts the priority from the process table entry
  - Makes the process ready
  - Returns the extracted priority to its caller
- Is the value returned guaranteed to be the priority of the process?
- Remember that in a concurrent environment, other processes can run at any time, and an arbitrary amount of time can pass between any two instructions

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
- 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?
- 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);

msg = receive();

msg = recvclr();
```

- *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

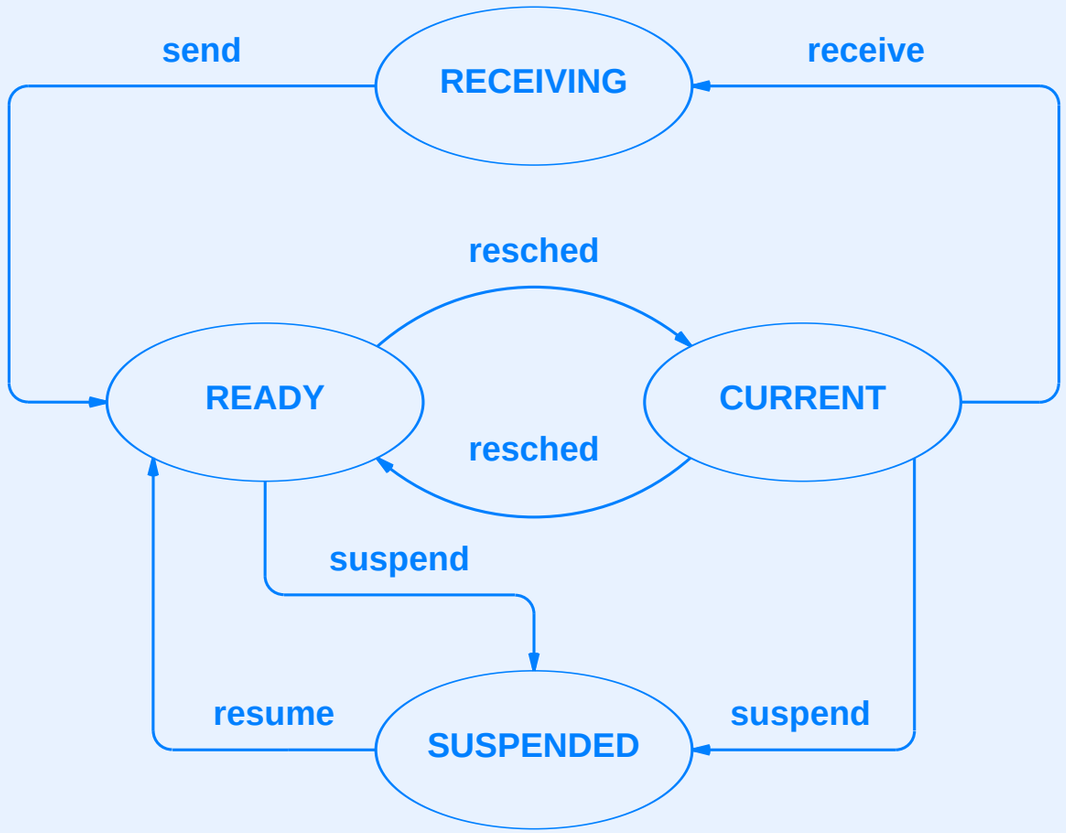
```
recvclr();  /* prepare to receive a message */
... /* allow other processes to send messages */
msg = receive();
```

- 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)
{
        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;
}
```

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
 *------------------------------------------------------------------------
 */
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;
}
```

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 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?