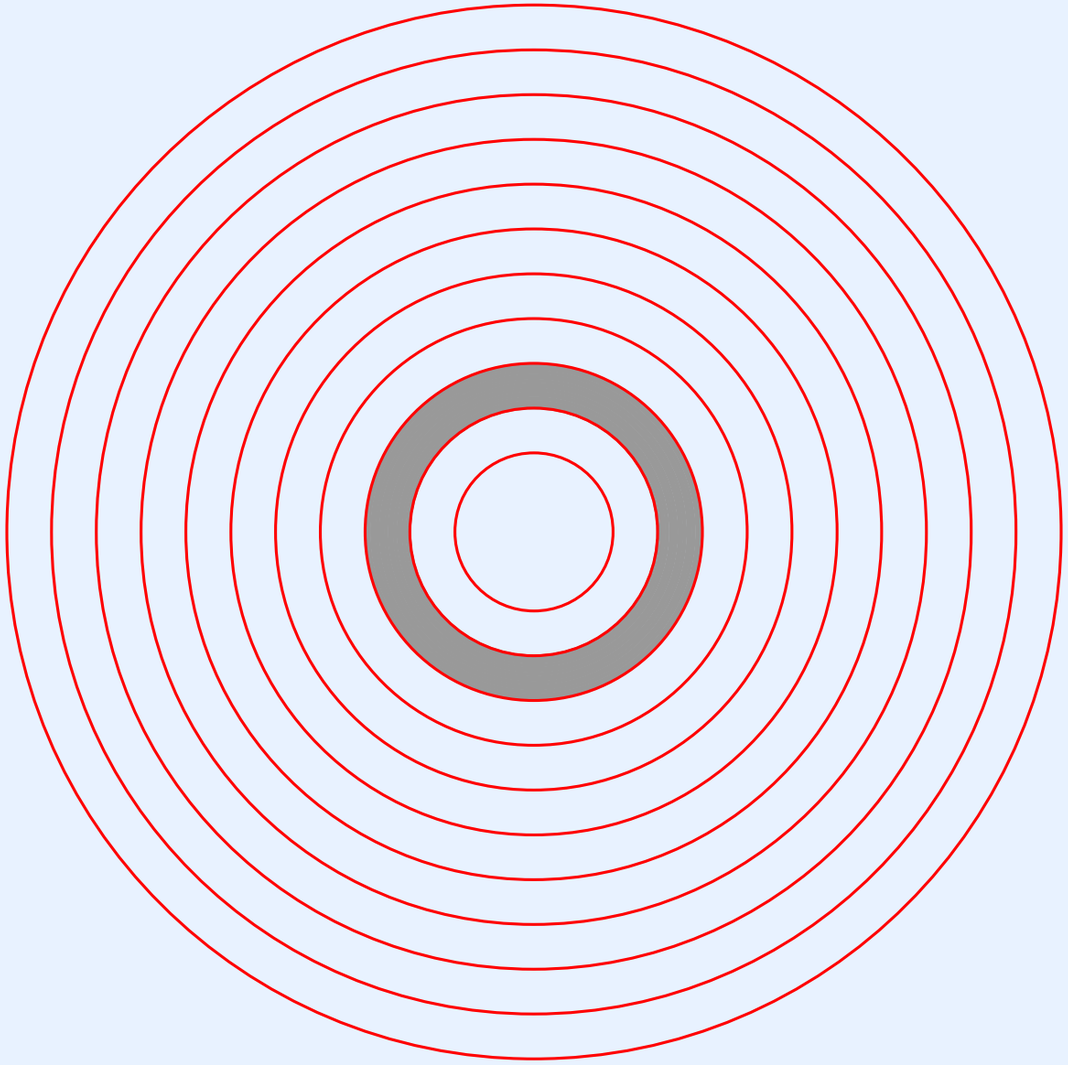
# Location Of Process Manager In The Hierarchy



# Review: What Is A Process?

- An abstraction known only to operating system
- The “running” of a program
- Runs concurrently with other processes

# A Fundamental Principle

- All computation must be done by a process
  - No execution can be done by the operating system itself
  - No execution can occur "outside" of a process
- Key consequence
  - At any time, a process *must* be running
  - An operating system cannot stop running a process unless it switches to another process

# Process Terminology

- Various terms have been used to denote a process
  - *Job*
  - *Task*
  - *Heavyweight process*
  - *Lightweight process / thread*
- Some of the differences are
  - Address space allocation and variable sharing
  - Longevity
  - Whether the process is declared at compile time or created at run time

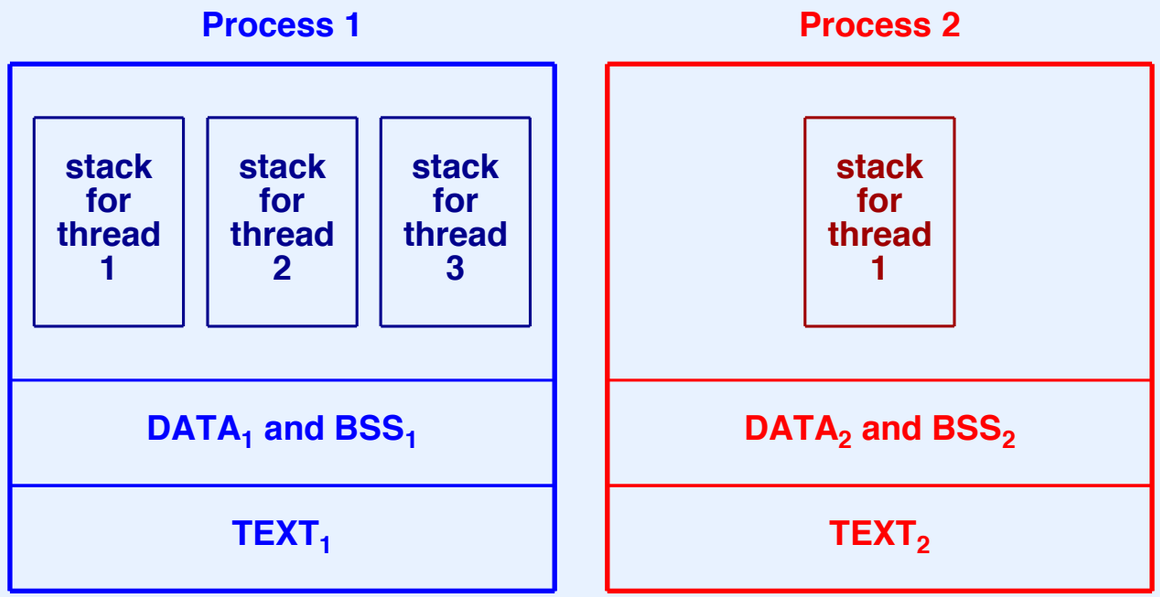
# Lightweight Process

- AKA *thread of execution*
- Can share data (data and bss segments) with other threads
- Has a private stack segment for
  - Local variables
  - Function calls

# Heavyweight Process

- AKA *Process* with an uppercase "P"
- Pioneered in Mach and adopted by Linux
- A single address space with one or more threads
- One data segment per Process
- One bss segment per Process
- Each thread is bound to a single Process, and cannot move to another

# Illustration Of Two Heavyweight Processes And Their Threads




- Threads within a Process share *text*, *data*, and *bss*
- No sharing between Processes
- Threads within a Process cannot share stacks

# Our Terminology

- The distinctions among *task*, *thread*, *lightweight process*, and *heavyweight process* are important to some groups
- For this course, we will use the term "process" unless we are specifically talking about the facilities in a specific system, such as Unix/Linux

# Maintaining Processes

- Remember that a process is
  - An OS abstraction unknown to hardware
  - Created dynamically
- The pertinent information must be kept by OS
- The OS stores information in a central data structure
- The data structure that hold
  - Is called a *process table*
  - Usually part of OS address space that is not accessible to applications

# Information Kept In A Process Table

- For each process
  - A unique *process identifier*
  - The owner (e.g., a user)
  - A scheduling priority
  - The location of the code and all data (including the stack)
  - The status of the computation
  - The current program counter
  - The current values for general-purpose registers

# Information Kept In A Process Table
## (continued)

- If a heavyweight process contains multiple threads, the process table stores for each thread
  - The owning process
  - The thread's scheduling priority
  - The location of the thread's stack
  - The status of the computation
  - The current program counter
  - The current values of registers
- Commercial systems may keep additional information, such as measurements of the process used for accounting

# The Xinu Process Model

- Xinu uses the simplest possible scheme
- Xinu is a single-user system, so there is no ownership
- Xinu uses one global context
- Xinu places all code and data in one global address space with
  - No boundary between the OS and applications
  - No protection
- Note: a Xinu process *can* access OS data structures directly, but good programming practice requires applications to use system calls

# Example Items In A Xinu Process Table

| Field | Purpose |
|---|---|
| prstate | The current status of the process (e.g., whether the process is currently executing or waiting) |
| prprio | The scheduling priority of the process |
| prstkptr | The saved value of the process's stack pointer when the process is not executing |
| prstkbase | The address of the base of the process's stack |
| prstklen | A limit on the maximum size that the process's stack can grow |
| prname | A name assigned to the process that humans use to identify the process's purpose |

# Process State

- Used by the OS to manage processes
- Is set by the OS whenever process changes status (e.g., waits for I/O)
- Consists of a small integer value stored in the process table
- Is tested by the OS to determine
  - Whether a requested operation is valid
  - The meaning of an operation

# The Set Of All Possible Process States

- Must be specified by designer when the OS is created
- One "state" is assigned per activity
- The value in process table is updated when an activity changes
- Example values
  - *Current* (process is currently executing)
  - *Ready* (process is ready to execute)
  - *Waiting* (process is waiting on semaphore)
  - *Receiving* (process is waiting to receive a message)
  - *Sleeping* (process is delayed for specified time)
  - *Suspended* (process is not permitted to execute)

# Definition Of Xinu Process State Constants

```
/* Process state constants */

#define PR_FREE         0       /* process table entry is unused        */
#define PR_CURR         1       /* process is currently running         */
#define PR_READY        2       /* process is on ready queue            */
#define PR_RECV         3       /* process waiting for message          */
#define PR_SLEEP        4       /* process is sleeping                  */
#define PR_SUSP         5       /* process is suspended                 */
#define PR_WAIT         6       /* process is on semaphore queue        */
#define PR_RECTIM       7       /* process is receiving with timeout    */
```

- Recall: the possible states are defined as needed when an operating system is constructed
- We will understand the purpose of each state as we consider the system design

# Scheduling

# Process Scheduling

- A fundamental part of process management
- Is performed by the OS
- Takes three steps
  - Examine processes that are eligible for execution
  - Select a process to run
  - Switch the processor from the currently executing process to the selected process

# Implementation Of Scheduling

- An OS designer starts with a *scheduling policy* that specifies which process to select
- The designer then builds a scheduling function that
  - Selects a process according to the policy
  - Updates the process table states for the current and selected processes
  - Calls a *context switch* function to switch the processor from the current process to the selected process

# Scheduling Policy

- Determines how processes should be selected for execution
- The goal is usually *fairness*
- The selection may depend on
  - The user's priority
  - How many processes the user owns
  - The time a given process has been waiting to run
  - The priority of the process
- The policy may be complex
- Note: both hierarchical and flat scheduling have been used

# The Example Scheduling Policy In Xinu

- Each process is assigned a *priority*
  - A non-negative integer value
  - Assigned when a process is created
  - Can be changed at any time
- The scheduler always chooses to run an eligible process that has highest priority
- The policy is implemented by a system-wide invariant

# The Xinu Scheduling Invariant

**At any time, the processor must be executing a highest priority eligible process. Among processes with equal priority, scheduling is round robin.**

# The Xinu Scheduling Invariant

**At any time, the processor must be executing a highest priority eligible process. Among processes with equal priority, scheduling is round robin.**

- The invariant must be enforced whenever
  - The set of eligible processes changes
  - The priority of any eligible process changes
- Such changes only happen during a system call or an interrupt (i.e., when running operating system code)

# Implementation Of Scheduling

- A process is *eligible* if its state is *ready* or *current*
- To avoid searching the process table during scheduling
  - Keep all ready processes on linked list called a *ready list*
  - Order the ready list by process priority
  - Scheduling is efficient because selection of a highest-priority process can be performed in constant time

# Xinu's High-Speed Scheduling Decision

- Compare the priority of the currently executing process to the priority of first process on ready list
  - If the current process has a higher priority, do nothing
  - Otherwise, extract the first process from the ready list and perform a *context switch* to switch the processor to the extracted process

# Xinu Scheduler Details

- The scheduler uses an unusual argument paradigm
- Before calling the scheduler
  - Global variable *currpid* gives ID of process that is currently executing
  - *proctab[currpid].prstate* must be set to desired *next* state for the current process
- If current process remains eligible and has highest priority, the scheduler does nothing (i.e., merely returns)
- Otherwise, the scheduler moves the current process to the specified state and runs the highest priority ready process

# Round-Robin Scheduling Of Equal-Priority Processes

- When inserting a process on the ready list, insert the process "behind" other processes with the same priority
- If scheduler switches context, the first process on ready list will be selected
- Note: the scheduler should switch context if the first process on the ready list has priority *equal* to the current process
- We will see how the implementation results in switching without a special case in the code

# Example Scheduler Code (resched Part 1)

```
/* resched.c - resched */

#include <xinu.h>

struct  defer   Defer;

/*------------------------------------------------------------------------
 *  resched  -  Reschedule processor to highest priority eligible process
 *------------------------------------------------------------------------
 */
void    resched(void)           /* Assumes interrupts are disabled      */
{
        struct procent *ptold;  /* Ptr to table entry for old process   */
        struct procent *ptnew;  /* Ptr to table entry for new process   */

        /* If rescheduling is deferred, record attempt and return */

        if (Defer.ndefers > 0) {
                Defer.attempt = TRUE;
                return;
        }

        /* Point to process table entry for the current (old) process */

        ptold = &proctab[currpid];
```

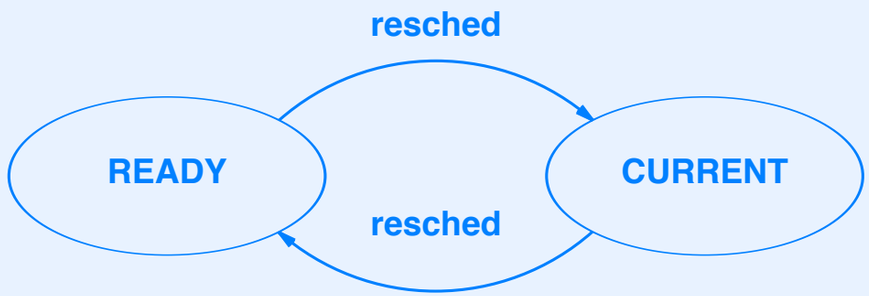
# Example Scheduler Code (resched Part 2)

```
        if (ptold->prstate == PR_CURR) {  /* Process remains eligible */
                if (ptold->prprio > firstkey(readylist)) {
                        return;
                }

                /* Old process will no longer remain current */

                ptold->prstate = PR_READY;
                insert(currpid, readylist, ptold->prprio);
        }

        /* Force context switch to highest priority ready process */

        currpid = dequeue(readylist);
        ptnew = &proctab[currpid];
        ptnew->prstate = PR_CURR;
        preempt = QUANTUM;              /* Reset time slice for process */
        ctxsw(&ptold->prstkptr, &ptnew->prstkptr);

        /* Old process returns here when resumed */

        return;
}
```

# Process State Transitions

- Recall that each process has a “state”
- The state (*prstate* in the process table) determines
  - Whether an operation is valid
  - The semantics of each operation
- A transition diagram documents valid operations

# Illustration Of Transitions Between The Current And Ready States




- Single function (*resched*) moves a process in either direction between the two states

# Context Switch

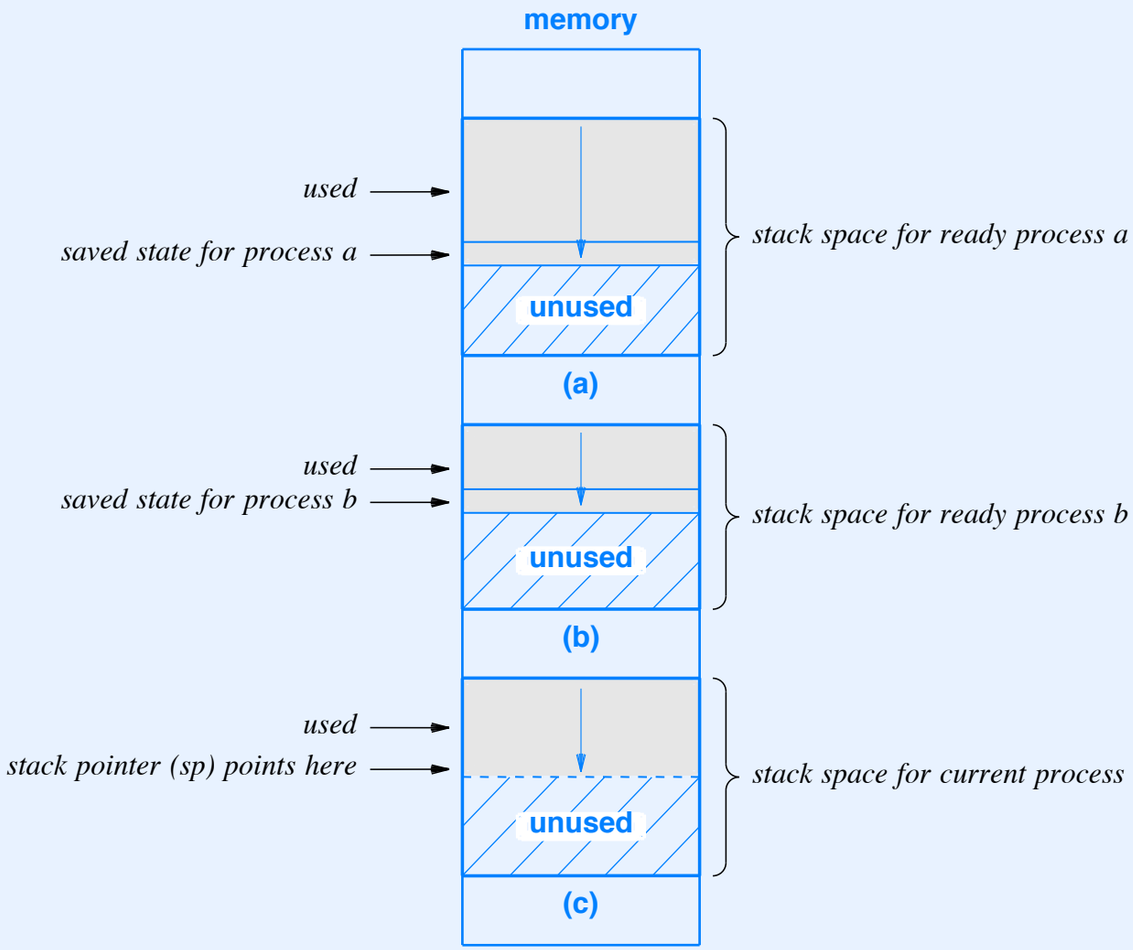
# Context Switch

- Forms a basic part of the process manager
- Is low-level (i.e., manipulates the underlying hardware directly)
- Must be written in assembly language
- Is only called by the scheduler
- Actually moves the processor from one process to another

# Saving State

- Recall: the processor only has one set of general-purpose registers
- The hardware may contain additional registers associated with a process (e.g., the interrupt mode)
- When switching from one process to another, the operating system must
  - Save a copy of all data associated with the current process
  - Pick up all the previously-saved data associated with the new process
- Xinu uses the process stack to save the state

# Illustration Of State Saved On Process Stack




- The stack of each *ready* process contains saved state

# Context Switch Operation

- Arguments specify the locations in the process table where the "old" process's stack and the "new" process's stack are saved
- Push a copy of all information pertinent to the old process on its stack
  - Contents of hardware registers
  - The program counter (instruction pointer)
  - Hardware privilege level and status
  - The memory map and address space information
- Save the current stack pointer in the process table entry for the old process

**...and then**

# Context Switch Operation
## (continued)

- Pick up the stack pointer that was saved in the process table entry for the new process and set the hardware stack pointer (i.e., switch the hardware from the old process's stack to the new process's stack)
- Pop the previously saved information for the new process from its stack and place the values in the hardware registers
- Resume execution at the place where the new process was last executing (i.e., return from the context switch to *resched*)

# Example Context Switch Code (Intel Part 1)

```
/* ctxsw.S - ctxsw (for x86) */

                .text
                .globl  ctxsw

/*------------------------------------------------------------------------
 * ctxsw -  X86 context switch; the call is ctxsw(&old_sp, &new_sp)
 *------------------------------------------------------------------------
 */
ctxsw:
                pushl   %ebp            /* Push ebp onto stack          */
                movl    %esp,%ebp       /* Record current SP in ebp     */
                pushfl                  /* Push flags onto the stack    */
                pushal                  /* Push general regs. on stack  */

                /* Save old segment registers here, if multiple allowed */

                movl    8(%ebp),%eax    /* Get mem location in which to */
                                        /*   save the old process's SP  */
                movl    %esp,(%eax)     /* Save old process's SP        */
                movl    12(%ebp),%eax   /* Get location from which to   */
                                        /*   restore new process's SP   */
```

# Example Context Switch Code (Intel Part 2)

```
/* The next instruction switches from the old process's */
/*   stack to the new process's stack.                  */

movl    (%eax),%esp     /* Pop up new process's SP      */

/* Restore new seg. registers here, if multiple allowed */

popal                   /* Restore general registers    */
movl    4(%esp),%ebp    /* Pick up ebp before restoring */
                        /*   interrupts                 */
popfl                   /* Restore interrupt mask       */
add     $4,%esp         /* Skip saved value of ebp      */
ret                     /* Return to new process        */
```

# The Null Process

- Does not compute anything useful
- Is present merely to ensure that at least one process remains ready at all times
- Simplifies scheduling (i.e., there are no special cases)

# Code For The Null Process

- The easiest way to code a null process is an infinite loop:

```
while(1)
    ; /* Do nothing */
```

- A loop may not be optimal because fetch-execute takes bus cycles that compete with I/O devices using the bus
- There are two ways to optimize
  - Some processors offer a special *pause* instruction that stops the processor until an interrupt occurs
  - Other processors have an instruction cache that means fetching the same instructions repeatedly will not access the bus

# Summary

- Process management is a fundamental part of an operating system
- Information about processes is kept in process table
- A state variable associated with each process records the process's activity
  - Currently executing
  - Ready, but not executing
  - Suspended
  - Waiting on a semaphore
  - Receiving a message

# Summary
## (continued)

- Scheduler
  - Is a key part of the process manager
  - Implements a scheduling policy
  - Chooses the next process to execute
  - Changes information in the process table
  - Calls the context switch to change from one process to another
  - Is usually optimized for high speed

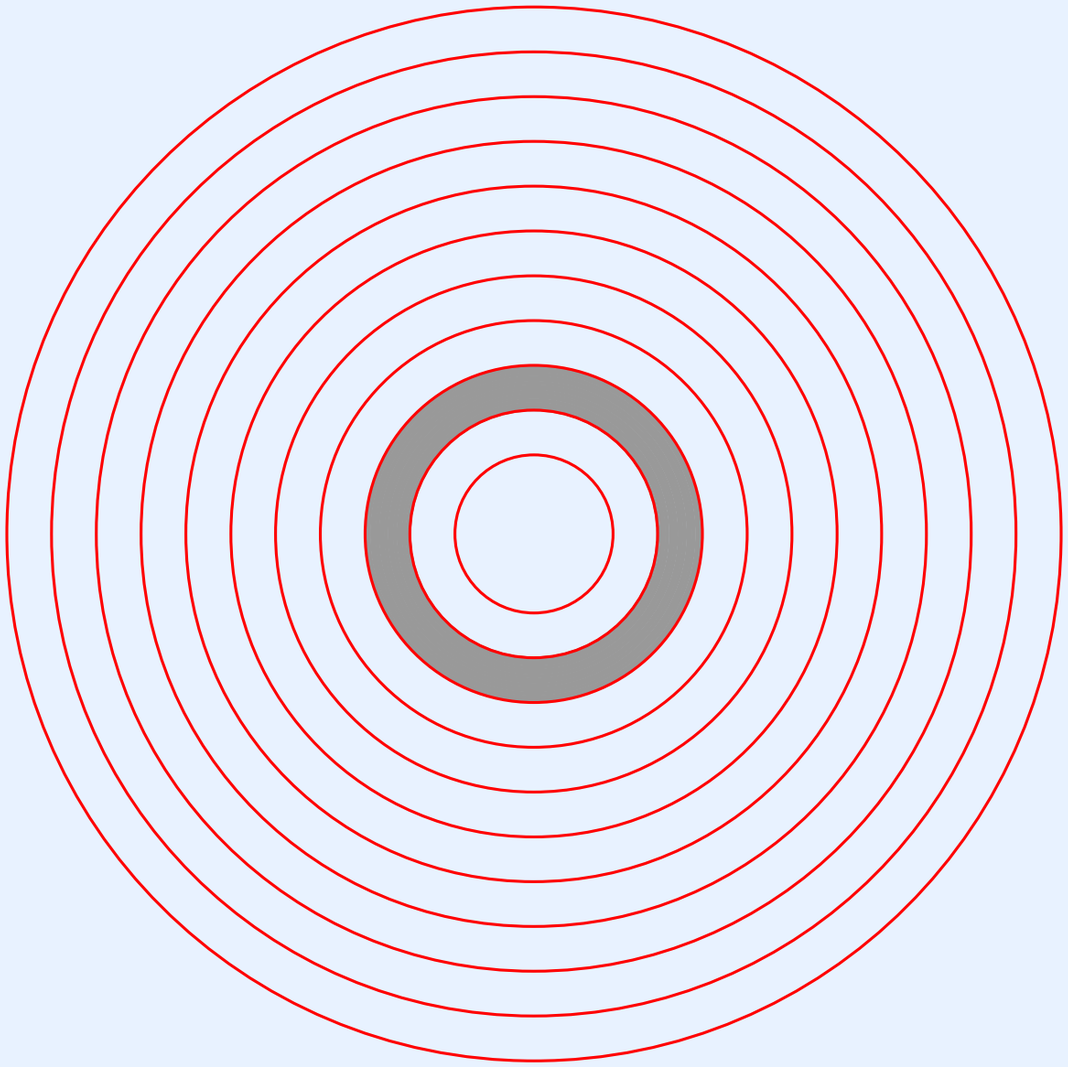
# Summary
## (continued)

- Context switch
  - Is a low-level part of a process manager
  - Moves the processor from one process to another
  - Involves saving and restoring hardware register contents
- The null process
  - Is needed so the processor has something to run when all user processes block to wait for I/O
  - Consists of an infinite loop
  - Runs at the lowest priority


Questions?

# 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?

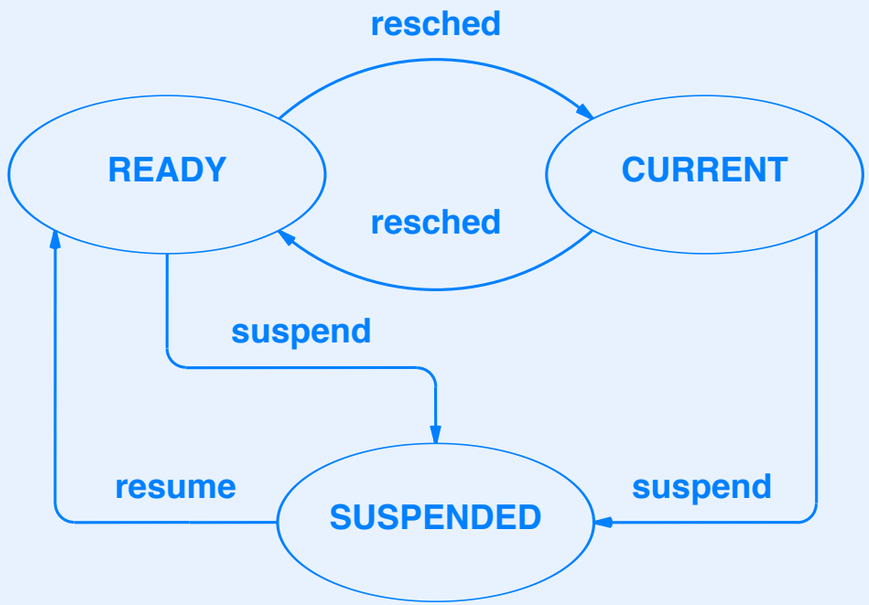
# Steps In Suspension And Resumption

- 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