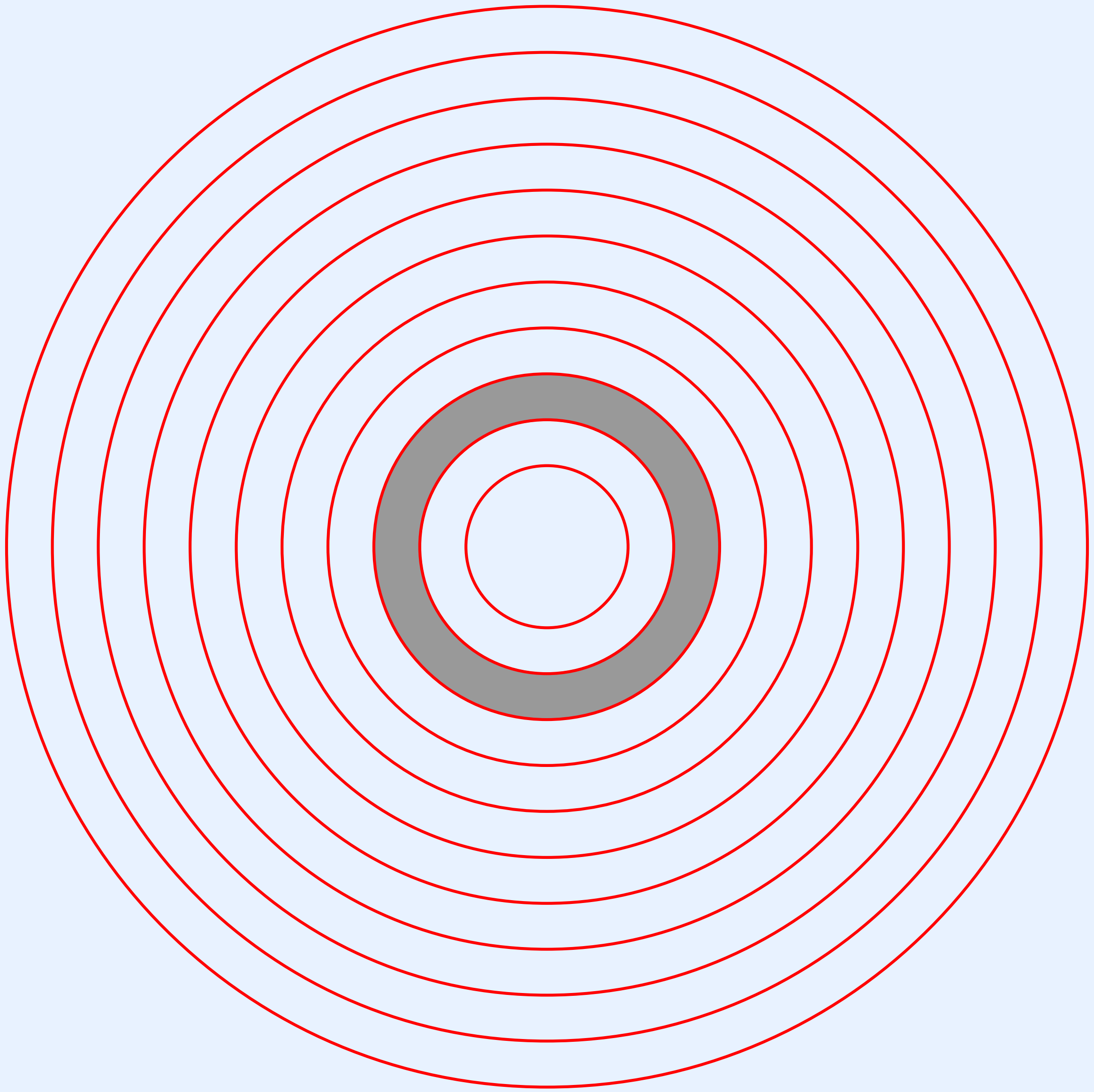
Module V

Process Management: Scheduling And Context Switching

Location Of The 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

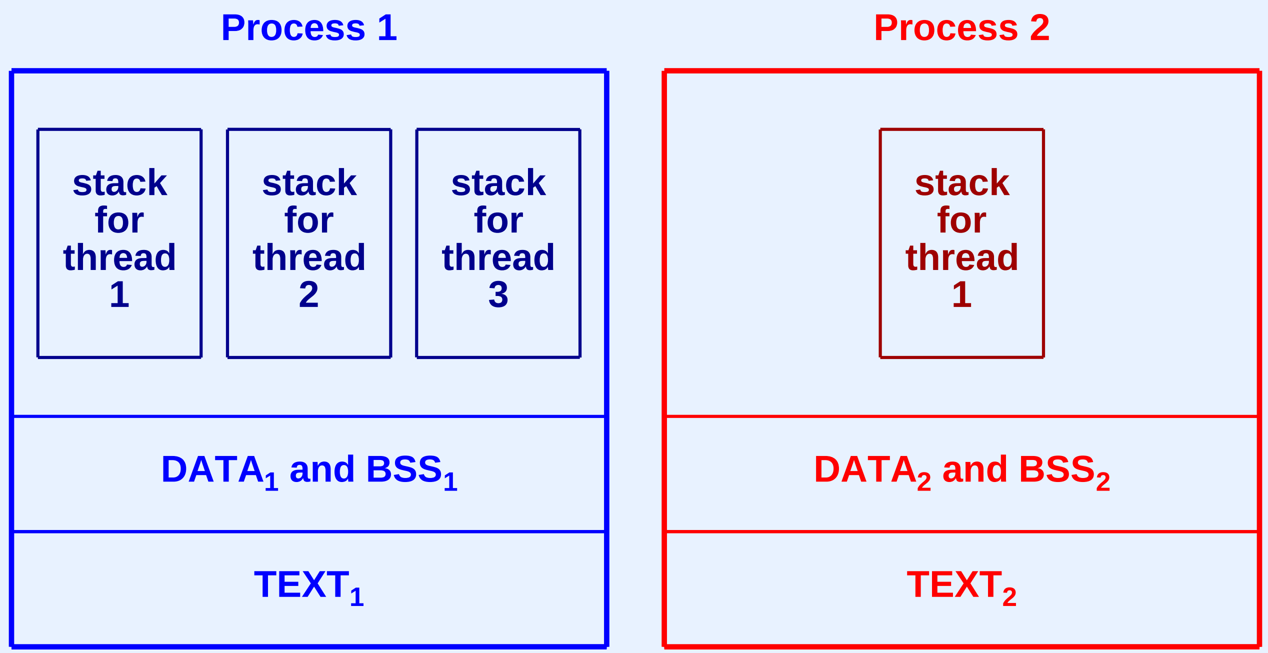
Lightweight Process

- AKA *thread of execution*
- Can share data (data and bss segments) with other threads
- Each thread 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
  - 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, the OS must keep the following
  - A unique *process identifier*
  - The owner of the process (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 must store information for each thread
- Commercial systems may keep additional information, such as measurements of the process used for accounting and billing

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

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 Field

- 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 for a given process
  – The meaning of an operation

Process States In Xinu

Constant	Meaning
PR_FREE	The entry in the process table is unused (not really a process state)
PR_CURR	The process is currently executing
PR_READY	The process is ready to execute
PR_RECV	The process is waiting for a message
PR_SLEEP	The process is waiting for a timer
PR_SUSP	The process is suspended
PR_WAIT	The process is waiting on a semaphore
PR_RECTIM	The process is waiting for a timer or a message, whichever occurs first

Constants For Process States

- Each process state is assigned a unique constant
- Definitions of 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    */
```

- 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

Ready And Current States

- When a process is ready to run, but not actually running, the process is placed in the *ready* state
- When a process is currently using the processor, the process is placed in the *current* state
- On a single core processor, many processes can be in the ready state, but at most one process can be *current*

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 Scheduling Policy In Xinu

- Follows a real-time approach
- 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 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., can only happen when running operating system code)

Implementation Of Scheduling

- A process is *eligible* if it is either ready to run but not running (i.e., its state is *ready*) or currently executing (i.e., its state is *current*)
- To avoid searching the process table during scheduling
  - Keep all ready processes on linked list called a *ready list*
  - Order the ready list in descending order by process priority
  - Scheduling is efficient because selection of a highest-priority process can be performed in constant time merely by selecting the process at the head of the ready list

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 (current process continues to run)
  - Otherwise, extract the first process from the ready list and perform a *context switch* to switch the processor from the current process to the extracted process
- The current process will be moved back to the ready list if it remains eligible to run
- Note: the current process does not appear on the ready list; instead, the process ID of the current process is stored in global variable *currpid*

Deferred Rescheduling

- The idea: *temporarily* delay rescheduling
- Also delay enforcement of the scheduling invariant
  - A call to *resched_cntl(DEFER_START)* defers rescheduling
  - A call to *resched_cntl(DEFER_STOP)* resumes normal scheduling
- Main purpose: allow a device driver to make multiple processes ready before any of them run
- We will see an example later
- For now, just understand that the current process will not change during a deferred rescheduling period; later in the course we will see how deferred rescheduling is used

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
- When the scheduler switches context, the first process on ready list will be selected
- Note: the scheduler switches context if the first process on the ready list has priority *equal* to the current process
- We will see how the implementation results in round-robin scheduling among equally high-priority processes without a special case in the code

Xinu Scheduler Code (resched Part 1)

```
/* resched.c - resched, resched_cntl */

#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];
```

Xinu 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;
}
```

Xinu Scheduler Code (resched Part 3)

```
/*------------------------------------------------------------------------
 *  resched_cntl  -  Control whether rescheduling is deferred or allowed
 *------------------------------------------------------------------------
 */
status  resched_cntl(           /* Assumes interrupts are disabled      */
          int32 defer           /* Either DEFER_START or DEFER_STOP     */
        )
{
        switch (defer) {

            case DEFER_START:   /* Handle a deferral request */

                if (Defer.ndefers++ == 0) {
                        Defer.attempt = FALSE;
                }
                return OK;

            case DEFER_STOP:    /* Handle end of deferral */
                if (Defer.ndefers <= 0) {
                        return SYSERR;
                }
                if ( (--Defer.ndefers == 0) && Defer.attempt ) {
                        resched();
                }
                return OK;

            default:
                return SYSERR;
        }
}
```

Contents Of resched.h

```
/* resched.h */

/* Constants and variables related to deferred rescheduling */

#define DEFER_START     1       /* Start deferred rescehduling          */
#define DEFER_STOP      2       /* Stop  deferred rescehduling          */

/* Structure that collects items related to deferred rescheduling       */

struct  defer   {
        int32   ndefers;        /* Number of outstanding defers         */
        bool8   attempt;        /* Was resched called during the        */
                                /*   deferral period?                   */
};

extern  struct  defer   Defer;
```

- Note: Defer.ndefers is set to zero when the system boots

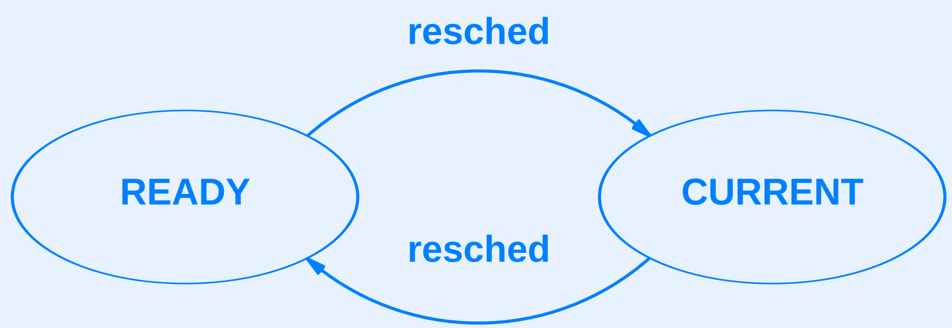
The Importance/Unimportance Of Process Scheduling

- Facts
  - At one time, process scheduling was the primary research topic in operating systems.
  - Extremely complex scheduling algorithms were created to keep processes proceeding
  - By the 1990s, interest in scheduling algorithms had faded
  - Now, almost no one uses complex scheduling algorithms
- Why did the topic fade?
- Was the problem completely solved?
- Answer: processors became so fast that processing is no longer a scarce resource

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 Switching

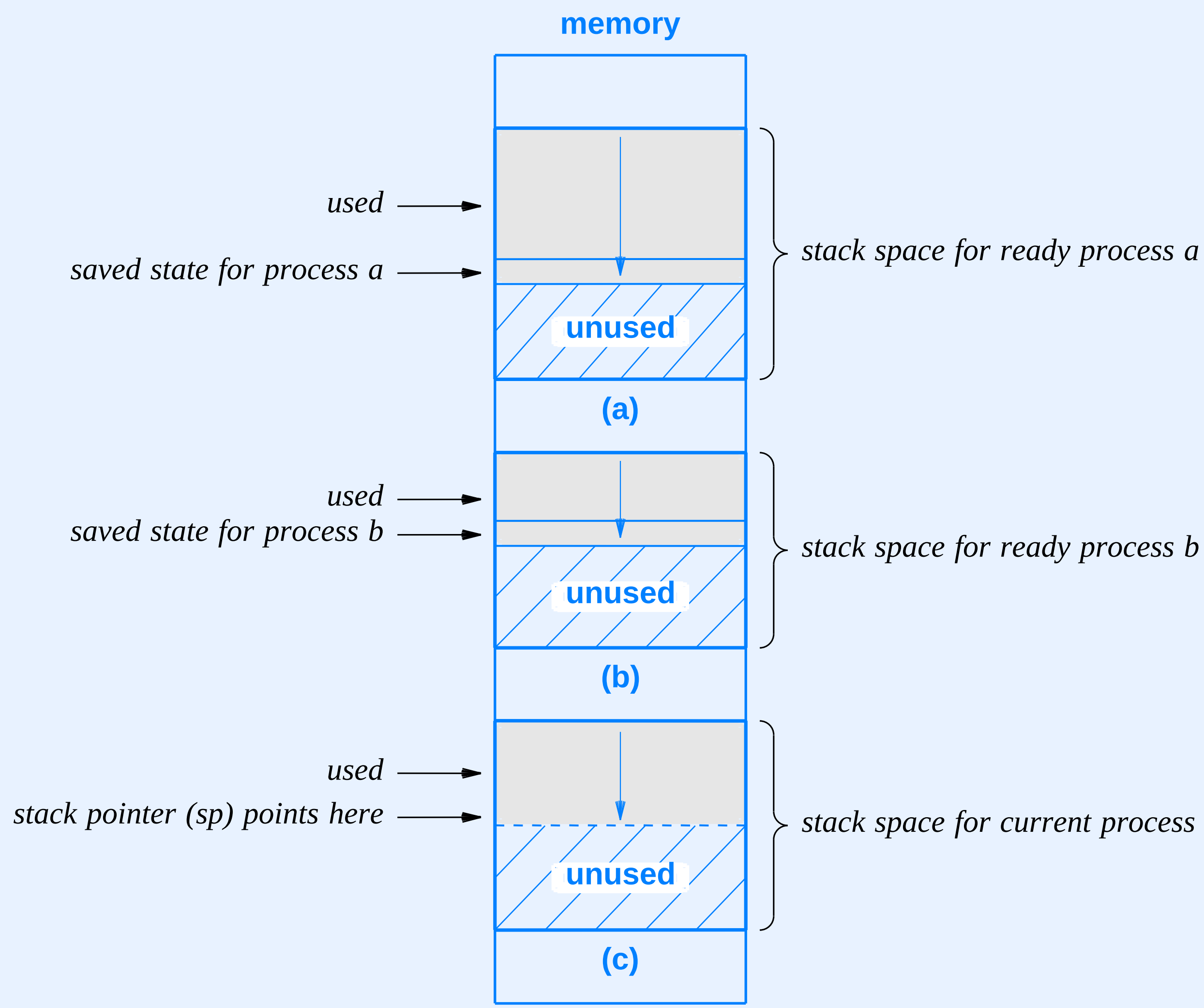
Context Switch

- 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: a 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 (including the contents of the general-purpose registers)
  - Pick up all the previously-saved data associated with the new process (including the contents of the general-purpose registers)
- 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 pointer will be saved and the “new” process’s stack pointer was previously 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)

...actually switch to the new process

- 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*)

Xinu Context Switch Code (Intel x86 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   */
```

Xinu Context Switch Code (Intel x86 Part 2)

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

movl    (%eax),%esp     /* Pick 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        */
```

Xinu Context Switch Code (ARM)

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

        .text
        .globl  ctxsw

/*------------------------------------------------------------------------
 * ctxsw -  ARM context switch; the call is ctxsw(&old_sp, &new_sp)
 *------------------------------------------------------------------------
 */

ctxsw:
        push    {r0-r11, lr}            /* Push regs 0 - 11 and lr      */
        push    {lr}                    /* Push return address          */
        mrs     r2, cpsr                /* Obtain status from coprocess.*/
        push    {r2}                    /*   and push onto stack        */
        str     sp, [r0]                /* Save old process's SP        */
        ldr     sp, [r1]                /* Pick up new process's SP     */
        pop     {r0}                    /* Use status as argument and   */
        bl      restore                 /*   call restore to restore it */
        pop     {lr}                    /* Pick up the return address   */
        pop     {r0-r12}                /* Restore other registers      */
        mov     pc, r12                 /* Return to the new process    */
```

Puzzle #1

- The Intel x86 is a CISC architecture with powerful instructions that may require many clock cycles to execute
- ARM is a RISC architecture where each instruction performs one basic operation and only requires one clock cycle
- Why is the Intel context switch code longer if instructions are more powerful?

Puzzle #2

- Our invariant says that at any time, a process must be executing
- The context switch code moves from one process to another
- Question: which process executes the context switch code?

Puzzle #3

- Our invariant says that at any time, one process must be executing
- Consider a situation in which all user processes are blocked (e.g., waiting for input)
- Which process executes?

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 consumes power and 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

A Dilemma And A Clever Solution

- Consider saving the instruction pointer
- If ctxsw saves it along with other state information, the instruction pointer will specify an instruction in the middle of ctxsw
- But then when ctxsw switches back to the process, the process will continue running in ctxsw
- Solution: instead of saving the actual instruction pointer, save the *return address* (i.e., the address to which ctxsw will return
- result: when a process runs again, it will start immediately after the call to ctxsw (i.e., as if ctxsw returned to resched)

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?