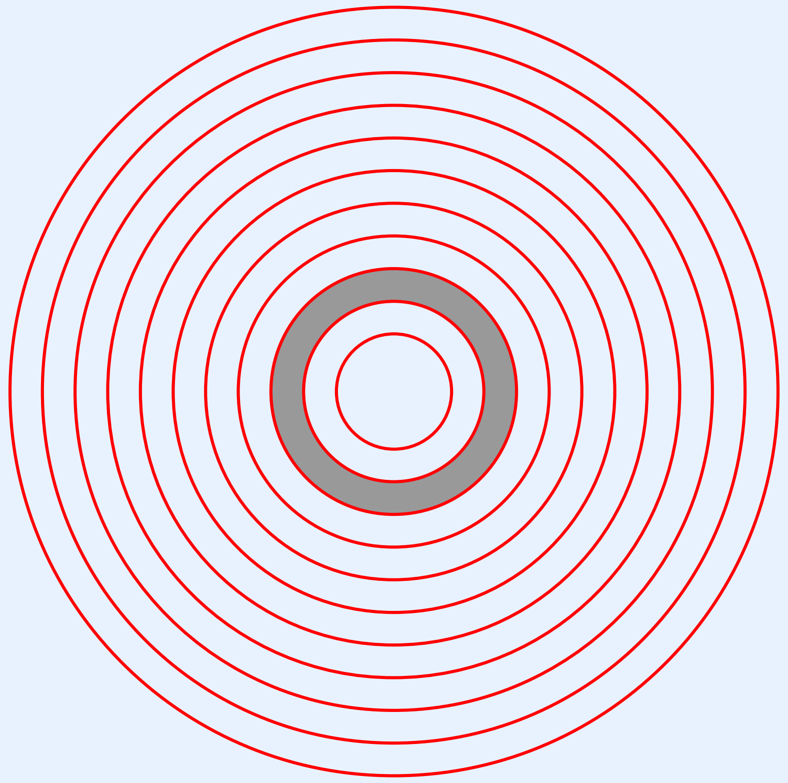
Module IV

Process Management: Scheduling, Context Switching, Process Suspension, Process Resumption, And Process Creation

Terminology

- The term *process management* has been used for decades to encompass the part of an operating system that manages concurrent execution, including both processes and the threads within them
- The term *thread management* is newer, but sometimes leads to confusion because it appears to exclude processes
- The best approach is to be aware of the controversy, but not worry about it

Location Of Process Manager In The Hierarchy



Process

- Unit of computation
- Abstraction of a processor
  - Known only to operating system
  - Not known by hardware
- Runs concurrently with other processes

A Fundamental Principle

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

Concurrency Models

- Many variations have been used
  - *Job*
  - *Task*
  - *Thread*
  - *Process*
- Differ in
  - Address space allocation and sharing
  - Coordination and communication mechanisms
  - Longevity
  - Dynamic vs. static definition

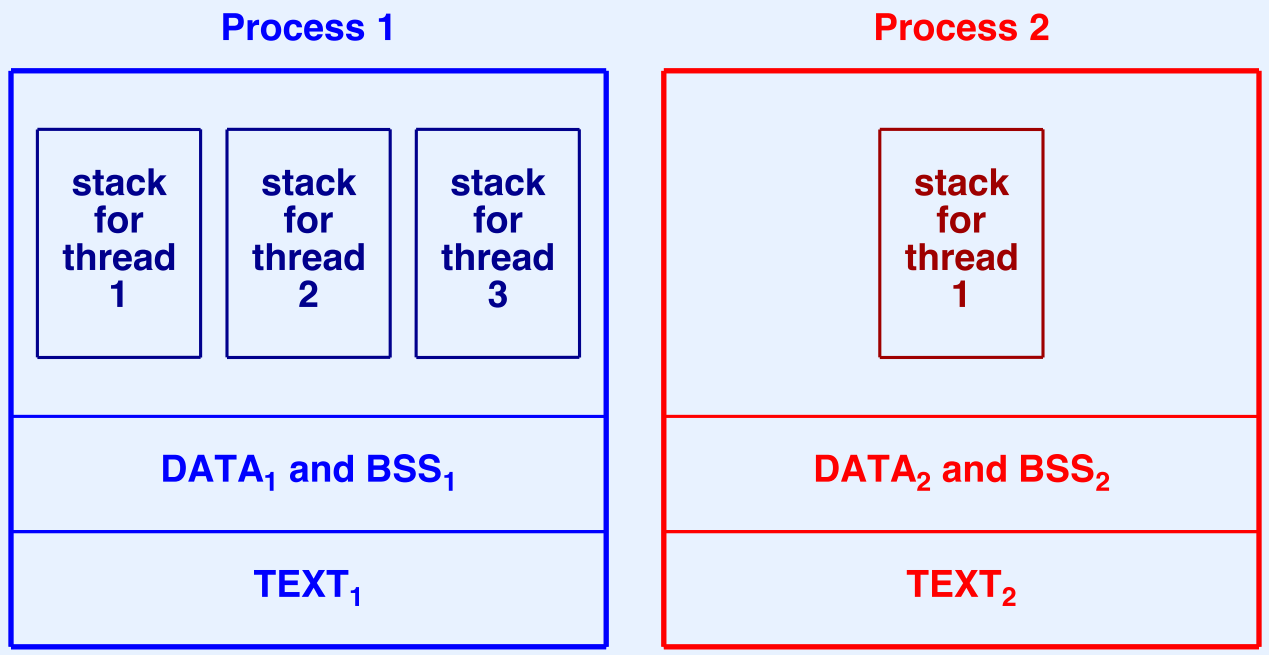
Thread Of Execution

- Single “execution”
- Sometimes called a *lightweight process*
- Can share data (data and bss segments) with other threads
- Must have private stack segment for
  - Local variables
  - Function calls

Heavyweight Process

- Pioneered in Mach and adopted by Linux
- Also called *Process* (written with uppercase “P”)
- Address space in which multiple threads can execute
- One data segment per Process
- One bss segment per Process
- Multiple threads per Process
- Given 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

Terminology

- Distinction between *process* and *Process* can be confusing
- For this course, assume generic use (“process”) unless
  - Used in context of specific OS
  - Speaker indicates otherwise

Maintaining Processes

- Process
  - OS abstraction
  - Unknown to hardware
  - Created dynamically
- Pertinent information kept by OS
- OS stores information in a central data structure
  - Called *process table*
  - Part of OS address space

Information Kept In A Process Table

- For each process
  - Unique *process identifier*
  - Owner (e.g., a user)
  - Scheduling priority
  - Location of code and all data (including the stack)
  - Status of the computation
  - Current program counter
  - Current values of registers

Information Kept In A Process Table (continued)

- If a heavyweight process contains multiple threads, keep for each thread
  - Owning process
  - Thread’s scheduling priority
  - Location of the thread’s stack
  - Status of the computation
  - Current program counter
  - Current values of registers

Xinu Process Model

- Simplest possible scheme
- Single-user system (no ownership)
- One global context
- One global address space
- No boundary between OS and applications
- Note: all Xinu processes can share data

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

Process States

- Specified by OS designer
- One “state” assigned per activity
- Value updated in process table when 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    */
```

- States are defined as needed when a system is constructed
- We will understand the purpose of each state as we consider the system design

Scheduling And Context Switching

Scheduling

- Fundamental part of process management
- Performed by OS
- Three steps
  - Examine processes that are eligible for execution
  - Select a process to run
  - Switch the processor to the selected process

Implementation Of Scheduling

- We need a *scheduling policy* that specifies which process to select
- We must then build a scheduling function that
  - Selects a process according to the policy
  - Updates the process table for the current and selected processes
  - Calls *context switch* to switch from current process to the selected process

Scheduling Policy

- Determines when process is selected for execution
- Goal is *fairness*
- May depend on
  - User
  - How many processes a user owns
  - Time a given process has been waiting to run
  - Priority of the process
- Note: hierarchical or flat scheduling can be used

Example Scheduling Policy In Xinu

- Each process assigned a *priority*
  - Non-negative integer value
  - Initialized when process created
  - Can be changed at any time
- Scheduler always chooses to run an eligible process that has highest priority
- 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.

- 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

Implementation Of Scheduling

- Process is eligible if state is *ready* or *current*
- To avoid searching process table during scheduling
  - Keep ready processes on linked list called a *ready list*
  - Order the ready list by process priority
  - Selection of highest-priority process can be performed in constant time

High-Speed Scheduling Decision

- Compare priority of current process to priority of first process on ready list
  - If 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 process

Deferred Rescheduling

- Delays enforcement of scheduling invariant
- Prevents rescheduling temporarily
  - A call to *resched_cntl(DEFER_START)* suspends 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

Xinu Scheduler Details

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

Round-Robin Scheduling

- When inserting a process on the ready list, place the process “behind” other processes with the same priority
- If scheduler switches context, first process on ready list is selected
- Note: scheduler switches context if the first process on the ready list has priority *equal* to the current process
- Later, we will see why the equal case is important

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;
}
```

Example Scheduler Code (resched part 3)

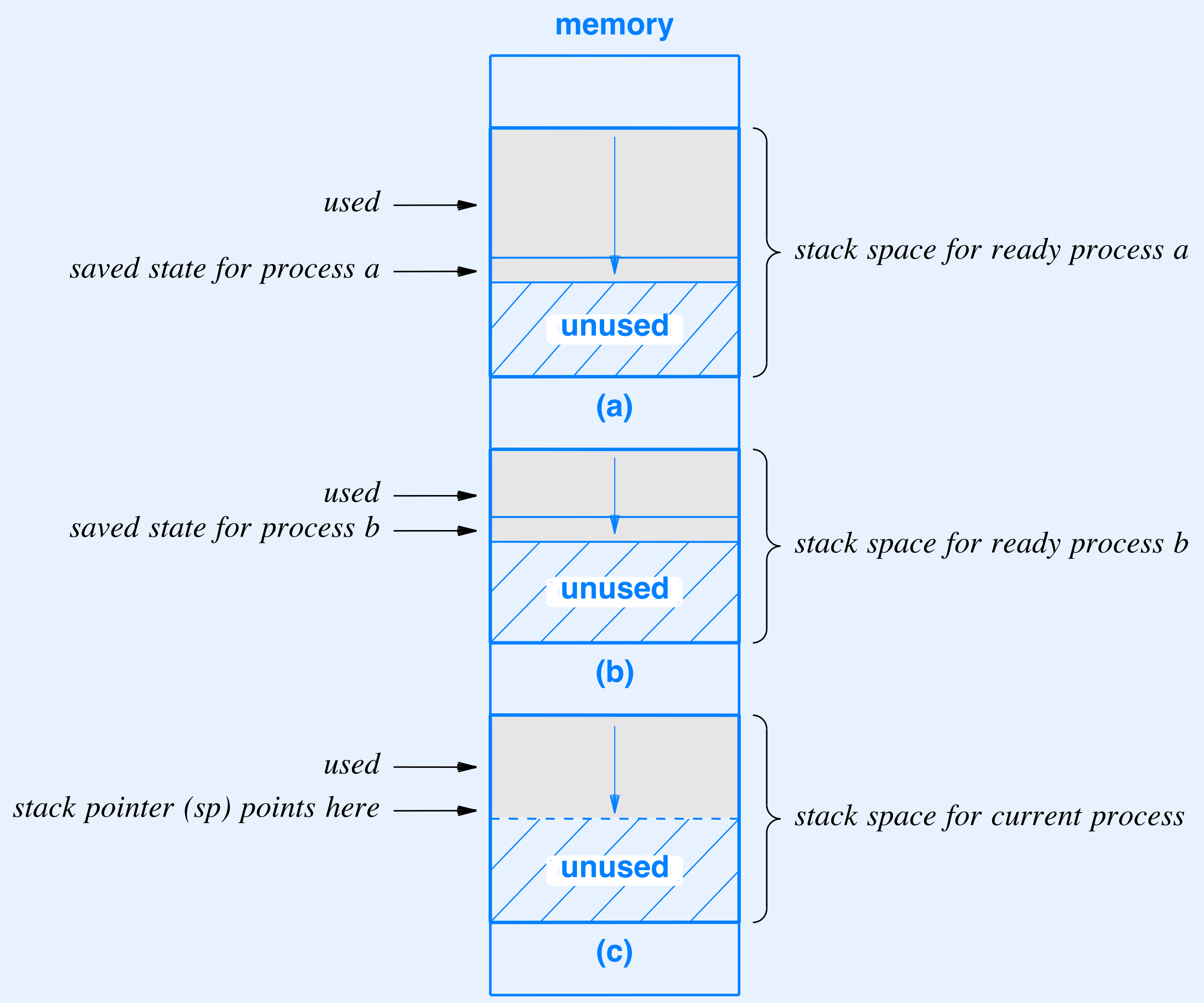
```
/*------------------------------------------------------------------------
 *  resched_cntl  -  Control whether rescheduling is deferred or allowed
 *------------------------------------------------------------------------
 */
status  resched_cntl(           /* Assumes interrupts are disabled      */
          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

Illustration Of State Saved On Process Stack

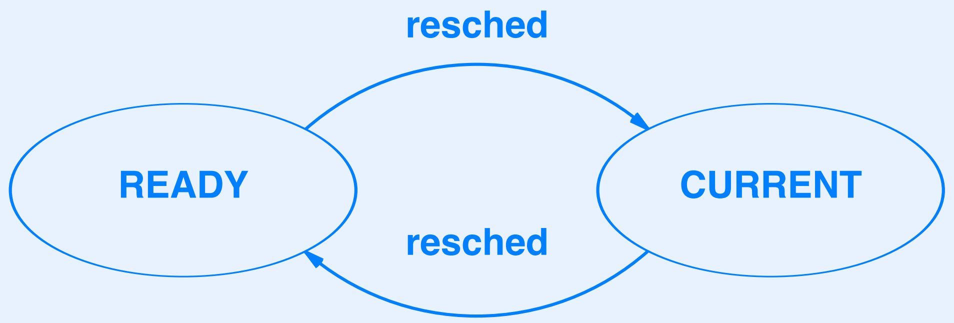

memory
used
saved state for process a
unused
stack space for ready process a
(a)
used
saved state for process b
unused
stack space for ready process b
(b)
used
stack pointer (sp) points here
unused
stack space for current process
(c)

- The stack of each ready process contains saved state

Process State Transitions

- Recall each process has a “state”
- State determines
  - Whether an operation is valid
  - Semantics of each operation
- 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

- Basic part of process manager
- Low-level (must manipulate underlying hardware)
- Usually written in assembly language
- Called by scheduler
- Moves processor from one process to another

Context Switch Operation

- Given a “new” process, *N*, and “old” process, *O*
- Save copy of all information pertinent to *O* on process O’s stack or the process table entry for process O
  - Contents of hardware registers
  - Program counter (instruction pointer)
  - Privilege level and hardware status
  - Memory map and address space
- Load saved information for *N*
- Resume execution of *N*

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         */
```

Example 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

- Intel is a CISC architecture with powerful instructions
- ARM is a RISC architecture where each instruction performs one basic operation
- Why is the Intel context switch code longer?

Solution to Puzzle #1

- It’s a trick question
- The ARM assembler uses shorthand
  - Programmer writes *push {r0-r11, lr}*
  - Assembler generates thirteen *push* instructions
- If a programmer wrote one line of code for each instruction, the ARM code would be much longer than the Intel code

Puzzle #2

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

Solution To Puzzle #2

- Both the *old* and *new* process do
- Old process
  - Executes first half of context switch
  - Is suspended
- New process
  - Continues executing where previously suspended
  - Usually runs second half of context switch

Puzzle #3

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

Solution To Puzzle #3

- Operating system needs an extra process
  - Called the *NULL process*
  - Never terminates
  - Always remains eligible to execute
  - Cannot make a system call that takes it out of ready or current state
  - Typically an infinite loop

Null Process

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

Code For A Null Process

- Easiest way to code a null process

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

- May not be optimal because fetch-execute takes bus cycles that compete with I/O devices
- Two ways to optimize
  - Some processors offer a special *pause* instruction that stops the processor until an interrupt occurs
  - Instruction cache can eliminate bus accesses

More Process Management

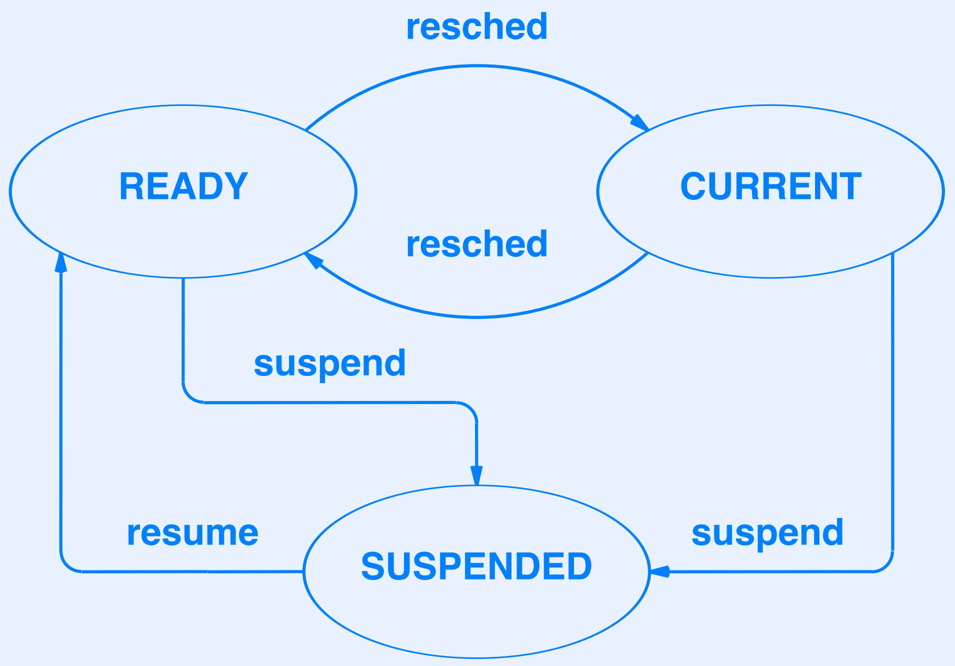
Process Manipulation

- Need to invent ways to control processes
- Example operations
  - Suspension
  - Resumption
  - Creation
  - Termination
- Recall: state variable in process table records activity

Process Suspension

- Temporarily “stop” a process
- Prohibit it from using the processor
- To allow later resumption
  - Process table entry retained
  - Complete state of computation saved
- OS sets process table entry to indicate process is suspended

State Transitions For Suspension And Resumption



- Ether current or ready process can be suspended
- Only a suspended process can be resumed
- System calls *suspend* and *resume* handle transitions

A Note About System Calls

- OS contains many functions
- Some functions correspond to system calls and others are internal
- We use the type *syscall* to distinguish

Example Suspension Code (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;
        }
```

Example Suspension Code (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

- Resume execution of previously suspended process
- Method
  - Make process eligible for processor
  - Re-establish scheduling invariant
- Note: resumption does *not* guarantee instantaneous execution

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;
}
```

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  );
}
```

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;
}
```

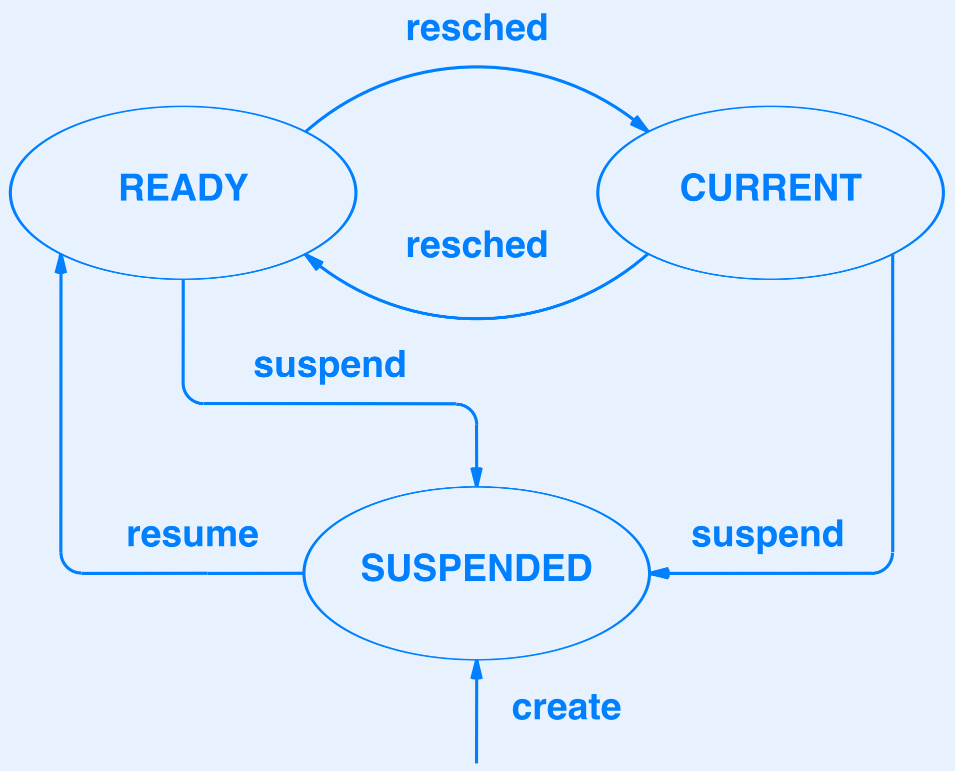
Process Termination

- Final and permanent
- Record of the process is expunged
- Process table entry becomes available for reuse
- Known as *process exit* if initiated by the thread itself
- We will see more about termination later

Process Creation

- Processes are dynamic — process *creation* refers to starting a new process
- Performed by *create* procedure in Xinu
- Method
  - Find free entry in process table
  - Fill in entry
  - Place new process in *suspended* state
- We will see more about creation later

Illustration Of State Transitions For Additional Process Management Functions



- Note that both current and ready processes can be suspended

At one time, process scheduling was the primary research topic in operating systems. Why did the topic fade? Was the problem completely solved?

Summary

- Process management is a fundamental part of OS
- Information about processes 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
  - Key part of the process manager
  - Chooses next process to execute
  - Implements a scheduling policy
  - Changes information in the process table
  - Calls context switch to change from one process to another
  - Usually optimized for high speed

Summary
(continued)

- Context switch
  - Low-level piece of a process manager
  - Moves processor from one process to another
- Processes can be suspended, resumed, created, and terminated
- At any time a process must be executing
- Special process known as *null process* remains ready to run at all times

Questions?