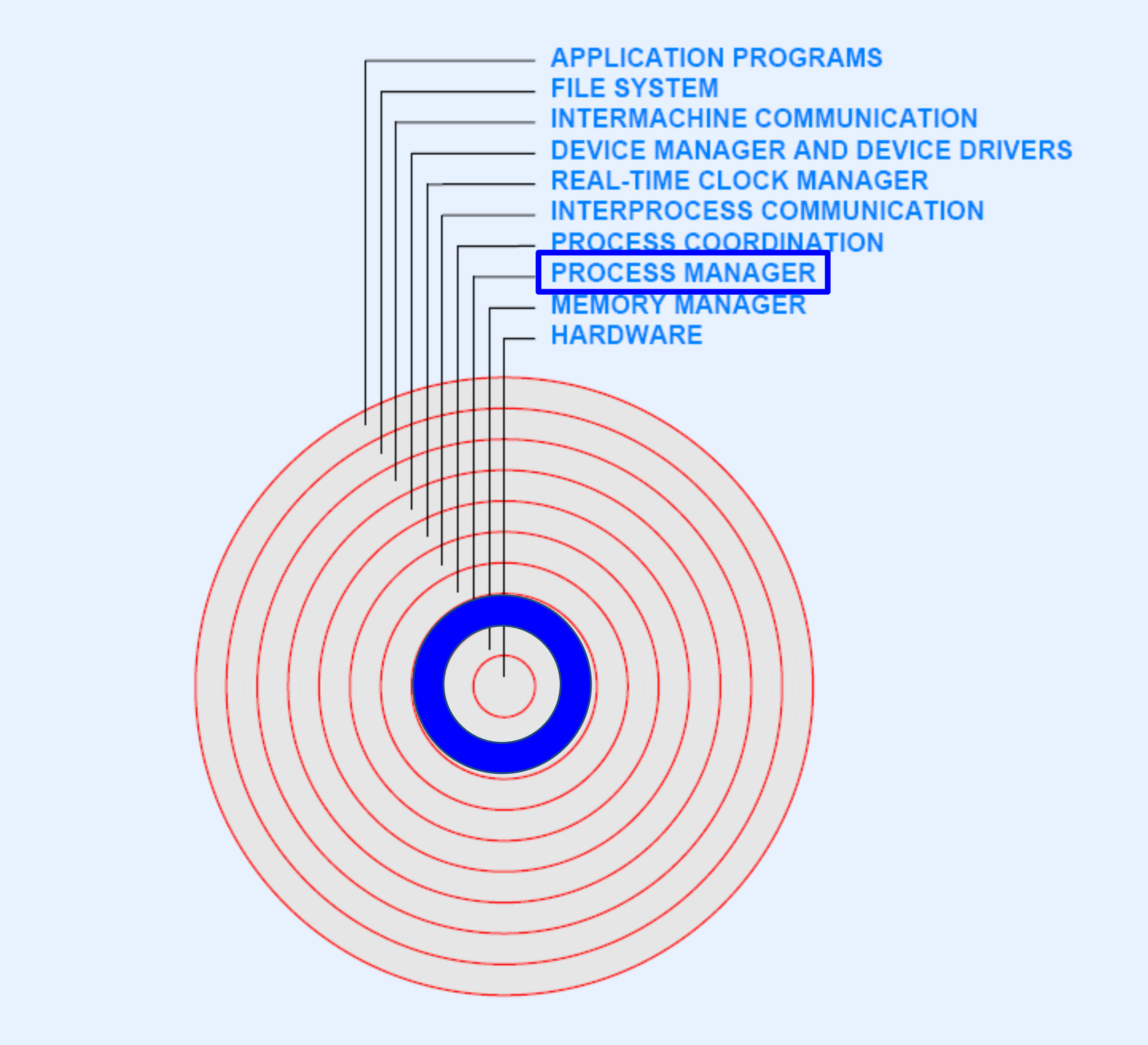
CSCI 8530
Advanced Operating Systems

Part 4

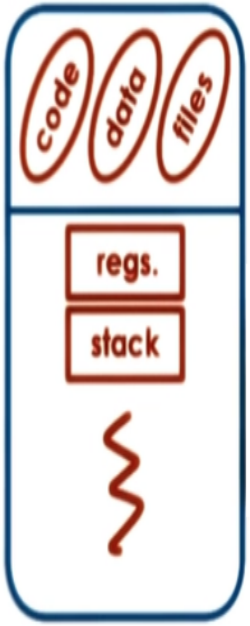
Process Management: Scheduling, Context Switching, Process Suspension, Process Resumption, and Process Creation

Location of Process Manager in the Hierarchy



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.
  - Shares all other resources



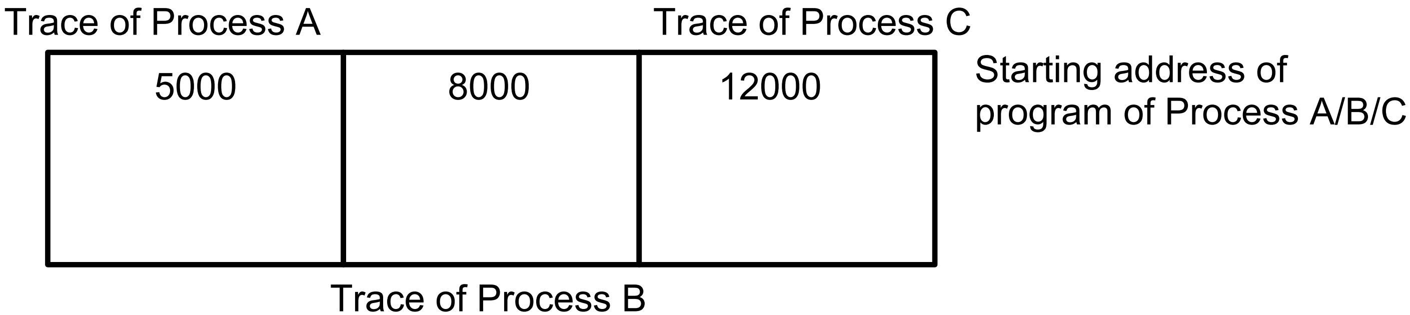
Process

Concurrent Processing

- A computing model in which multiple processors execute instructions simultaneously for better performance
- Process == Unit of computation
- Abstraction of a processor
  - Known only to operating system (processes)
  - Not known by hardware (instructions sets)

A Fundamental Principle

- All computation must be done by a process
  - No execution by the operating system itself
  - No execution “outside” of a process



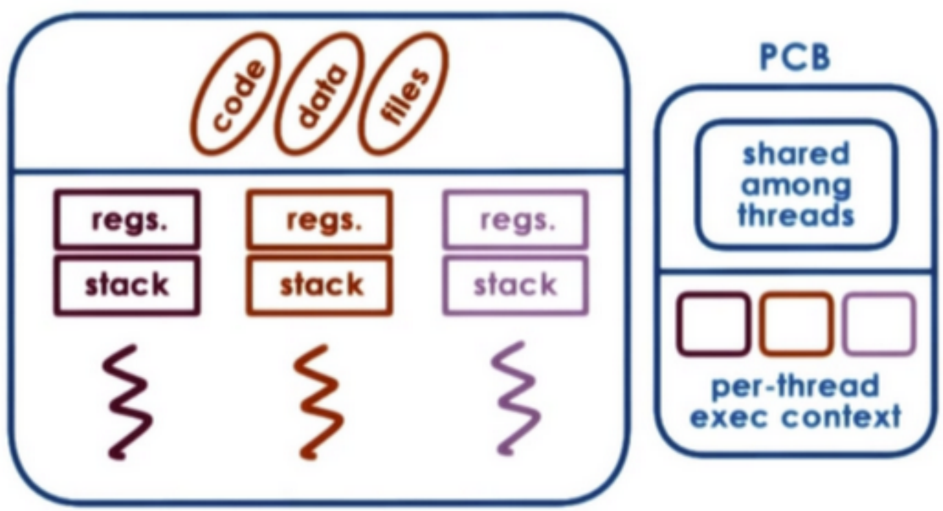
- Key consequence
  - At any time, a process *must* be running
  - Operating system cannot stop running a process unless it switches to another process

Concurrency Models

- Many variations have been used to a single computation
  - *Job; Process:* a single computation that is self-contained and isolated from other computations.
  - *Task:* a process that is declared statically
  - *Thread of control:* a type of concurrent process that shares an address space with other threads.
- 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



A Lightweight Process

- Creating a thread is cheaper than creating a process
- Communication between threads is easier than between processes
  - Processes must set up a shared resource or pass messages or signals
- Context switching between threads is cheaper (same address space)

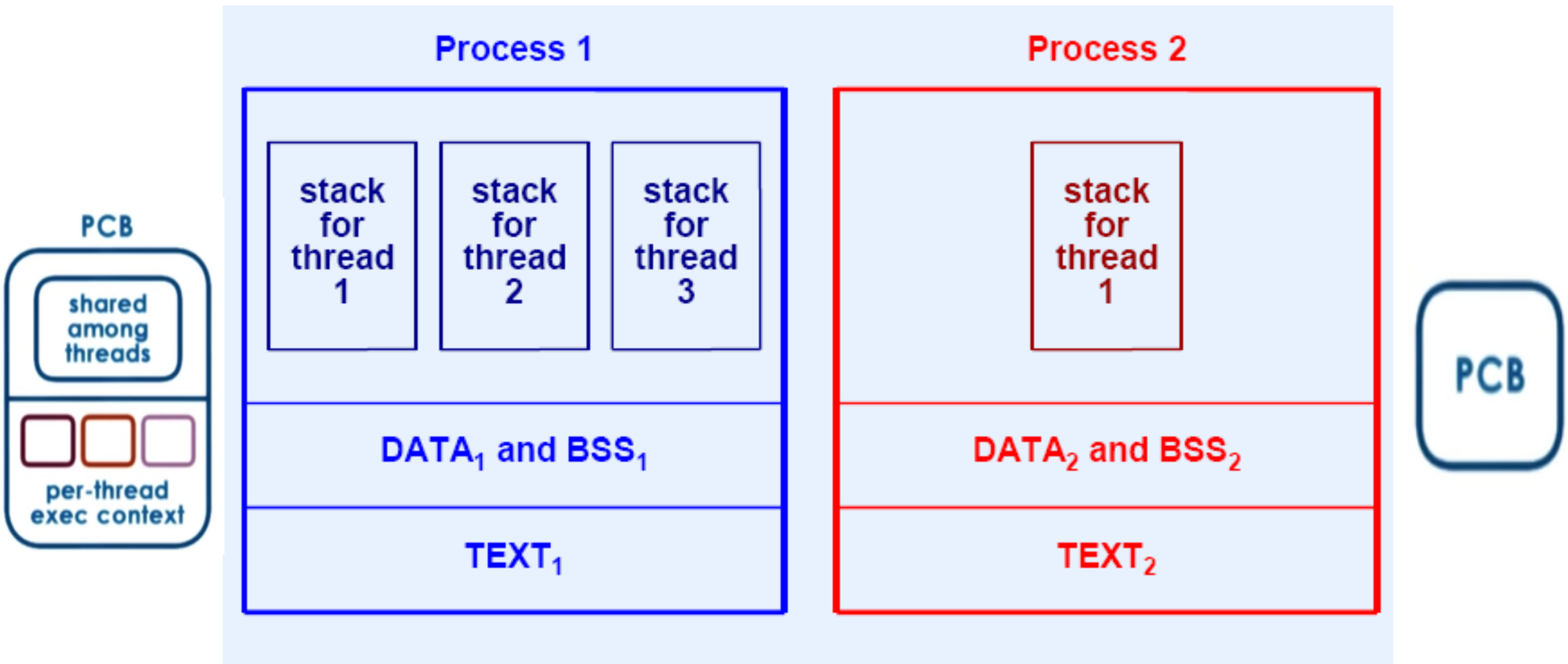
When To Use Threads

- Threads are appropriate tools to use when multiple independent execution paths (or computations) need access to the same address space and other resources.
- For example, consider a program such as a terminal emulator, like putty.
  - Wait for input from the keyboard
  - Wait for a network packet from a remote system

Heavyweight Process

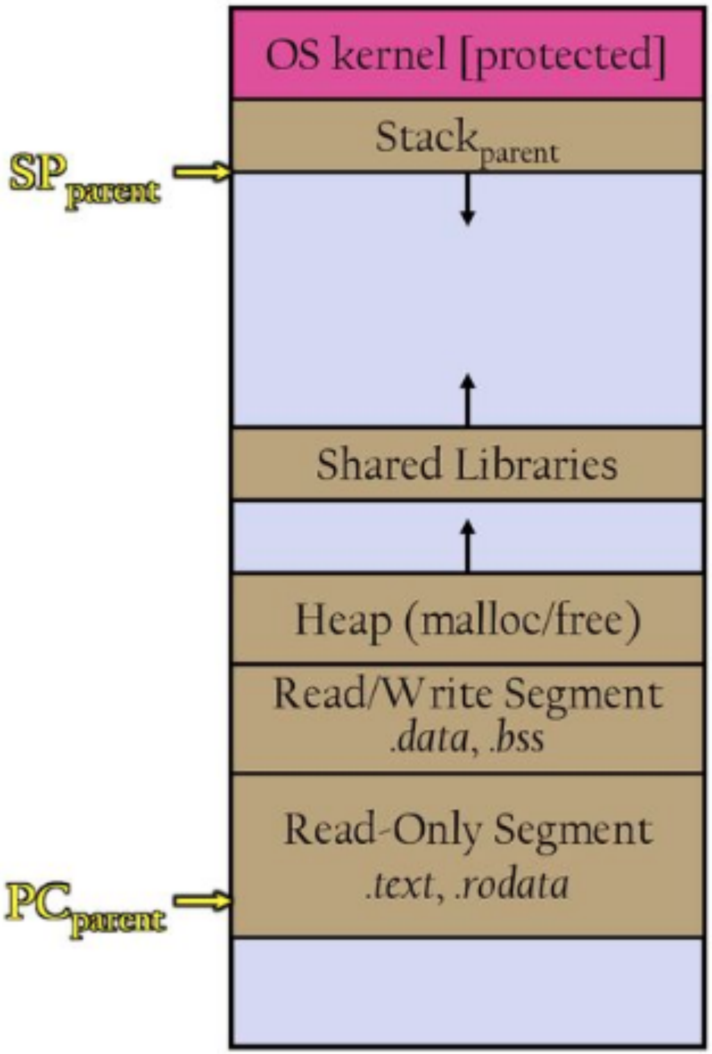
- Pioneered in Mach System (a two-level concurrent programming scheme) and adopted by Linux
- Also called *Process* (written with uppercase “P”)
- Create an address space in which multiple threads can execute
- One data segment per Process
- One bss segment per Process
- Multiple threads per Process
- 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

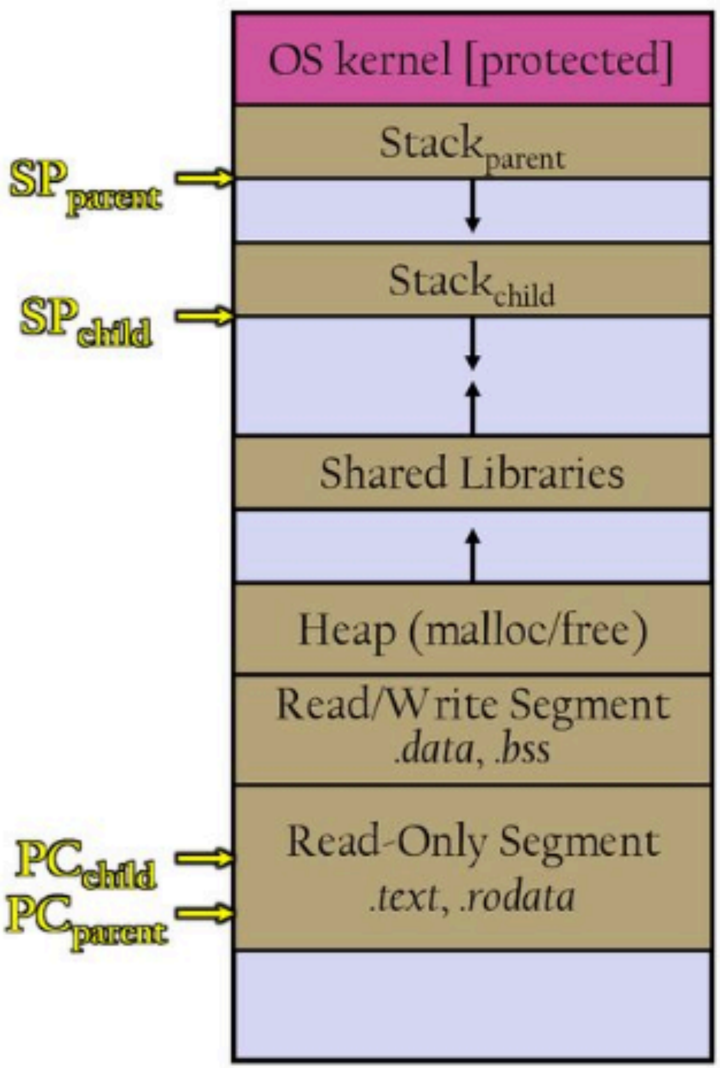
Threads Implementation



Before creating a thread

- One thread of execution running in the address space
- That main thread invokes a function to create a new thread
  - phtread_create()

Threads Implementation



After creating a thread

- Two threads of execution running in the address
  - Extra stack created
  - Child thread maintains separate values of its SP and PC
- Both threads share the other segments
  - They can cooperatively modify shared data

Terminology

- For an embedded environment, Xinu permits processes to share an address space
  - Xinu processes follow a thread model
- For this course, assume generic use ("process") unless used in context of specific OS

Maintaining Processes

- Process
  - An "instantiation" of a program
  - 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 (1/2)

For each process

- Identification information:
  - Unique *process identifier*
  - Owner (a user)
- Control information:
  - Scheduling priority
- An address space
  - Location of code and data (stack)
- Other execution contexts – threads
  - Status of computation
  - Current program counter
  - Current values of registers

Information Kept in a Process Table (2/2)

- If a heavyweight process contains multiple threads, keep for each thread
  - Owning process
  - Thread's scheduling priority
  - Location of stack
  - Status of 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

(*proctab*)

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' **stack pointer** when the process is not executing
prstkbase	The **address of the base** of the process' stack
prstklen	A limit on the **maximum size** that the process' stack can grow
prname	A **name** assigned to the process that humans use to identify the process' purpose

- Each entry in *proctab* is defined to be a struct of type *procent*.
- The declaration of struct *procent* can be found in file **process.h**

struct *procent* in *Process.h*

The declaration of struct *procent* in file *process.h* along with other declarations related to processes.

```
/* Definition of the process table (multiple of 32 bits) */

struct procent {                /* Entry in the process table */
        uint16 prstate;         /* Process state: PR_CURR, etc. */
        pri16 prprio;           /* Process priority */
        char *prstkptr;         /* Saved stack pointer */
        char *prstkbase;        /* Base of run time stack */
        uint32 prstklen;        /* Stack length in bytes */
        char prname[PNMLEN];    /* Process name */
        sid32 prsem;            /* Semaphore on which process waits */
        pid32 prparent;         /* ID of the creating process */
        umsg32 prmsg;           /* Message sent to this process */
        bool8 prhasmsg;         /* Nonzero iff msg is valid */
        int16 prdesc[NDESC];    /* Device descriptors for process */
};
```

Process States (1/2)

- 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 (2/2)

- 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

File *process.h* contains the definitions

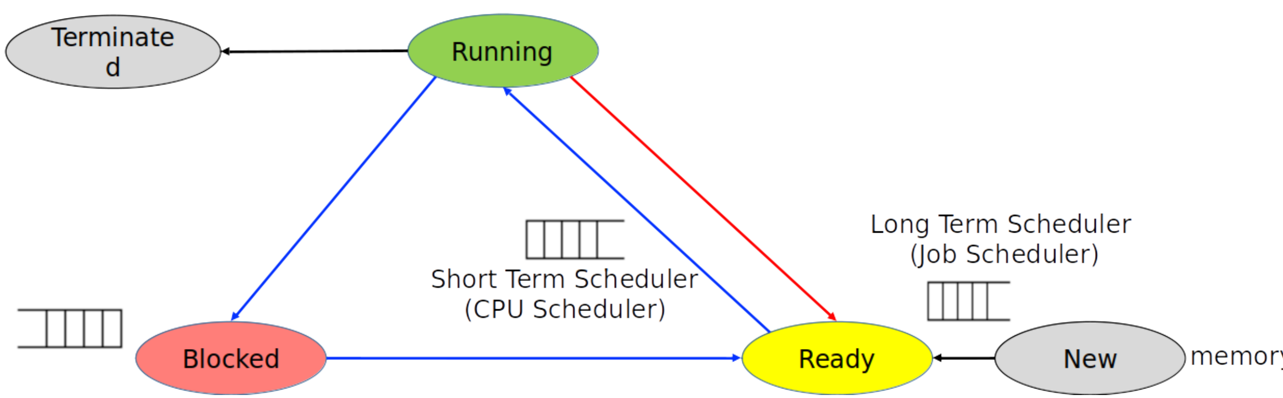
```
/* Process state constants */

#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 */
```

- Xinu uses field *prstate* in the process table to record state information for each process
- The system defines 0-7 valid states and a symbolic constant for each.
- States are defined as needed when a system is constructed
- Xinu always keeps the code and data for all processes in memory.

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

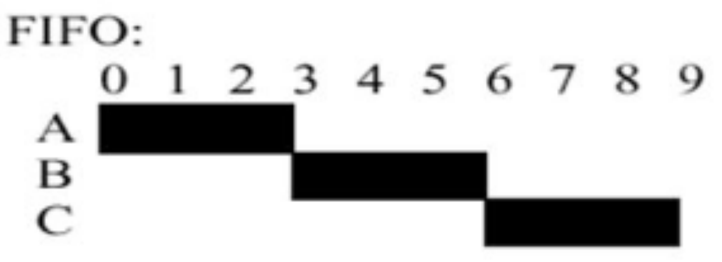
- In Xinu, function *resched* makes the selection according to the following well-known scheduling policy:

> *At any time, the highest priority eligible process is executing. Among processes with equal priority scheduling is round-robin.*

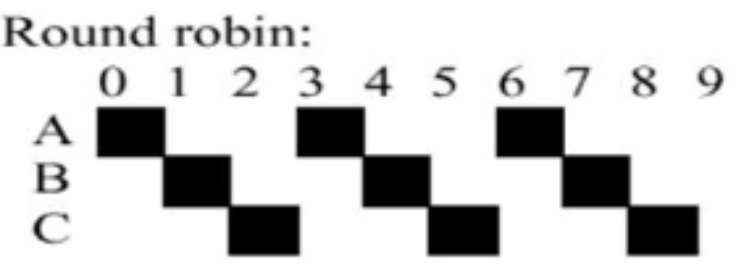
- *Ready* takes an argument that specifies a process ID and executes this process
- Each OS function should maintain a *scheduling invariant*:
  - Ensure that the highest priority process is executing when the function returns
- If a function changes the state of processes, the function must call *resched* to reestablish the invariant.

Round-Robin Scheduling

- Each time a process can use at most a specified amount of time called its *quantum*
- 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



Process A finishes after 3 slices, B 6, and C9.
The average is (3+6+9)/3 = 6 slices



Process A finishes after 7 slices, B 8, and C9.
The average is (7+8+9)/3 = 8 slices

Implementation of Scheduling in Xinu

- A scheduler consists of a function that a running process calls to willingly give up the processor
- Every process: *prprio* field of the process's entry
  - A user assigns a priority to each process to control
- 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 in Xinu

- To provide fast access to the current process, its ID is stored in global integer variable *currpid*.
- 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 (PR_READY) and perform a *context switch* to switch the processor to the process
    - In such situations, the scheduler must change the state of the current process to PR_READY and insert the process onto the ready list

Deferred Rescheduling

- Motivation: some OS functions move multiple processes onto the ready list at the same time
  - Rescheduling can result in incomplete and incorrect operation
- The solution for multiple processes consists of temporarily suspending the scheduling policy.
  - A call to *resched_cntl(DEFER_START)* suspends rescheduling
  - A call to *resched_cntl(DEFER_STOP)* resumes normal scheduling
  - Each request deferral, a global counter, Defer.ndefers
- Main purpose: allow device driver can make multiple processes ready before allowing any of them to run

Xinu Scheduler Details

`/* resched.c */`

- *Resched* checks global variable Defer.ndefers to see whether rescheduling is deferred
- If deferred, resched sets global variable Defer.attempt
- Once it passes the test for deferral, *resched* examines the state of the current process *prstate*
  - If the state contains PR_CURR and the current process is the highest priority → *resched* returns
  - If the state specifies PR_CURR but the current process is not the highest priority → *resched* adds the current process to the ready list
    - Perform a context switch

Example Scheduler Code (1/3)

```
/* 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 (2/3)

```
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;
}
```

If context switching occurs, *resched* chooses to switch to another process, the original process will be stopped in the call to ctxsw (assembly language function).

Example Scheduler Code (3/3)

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

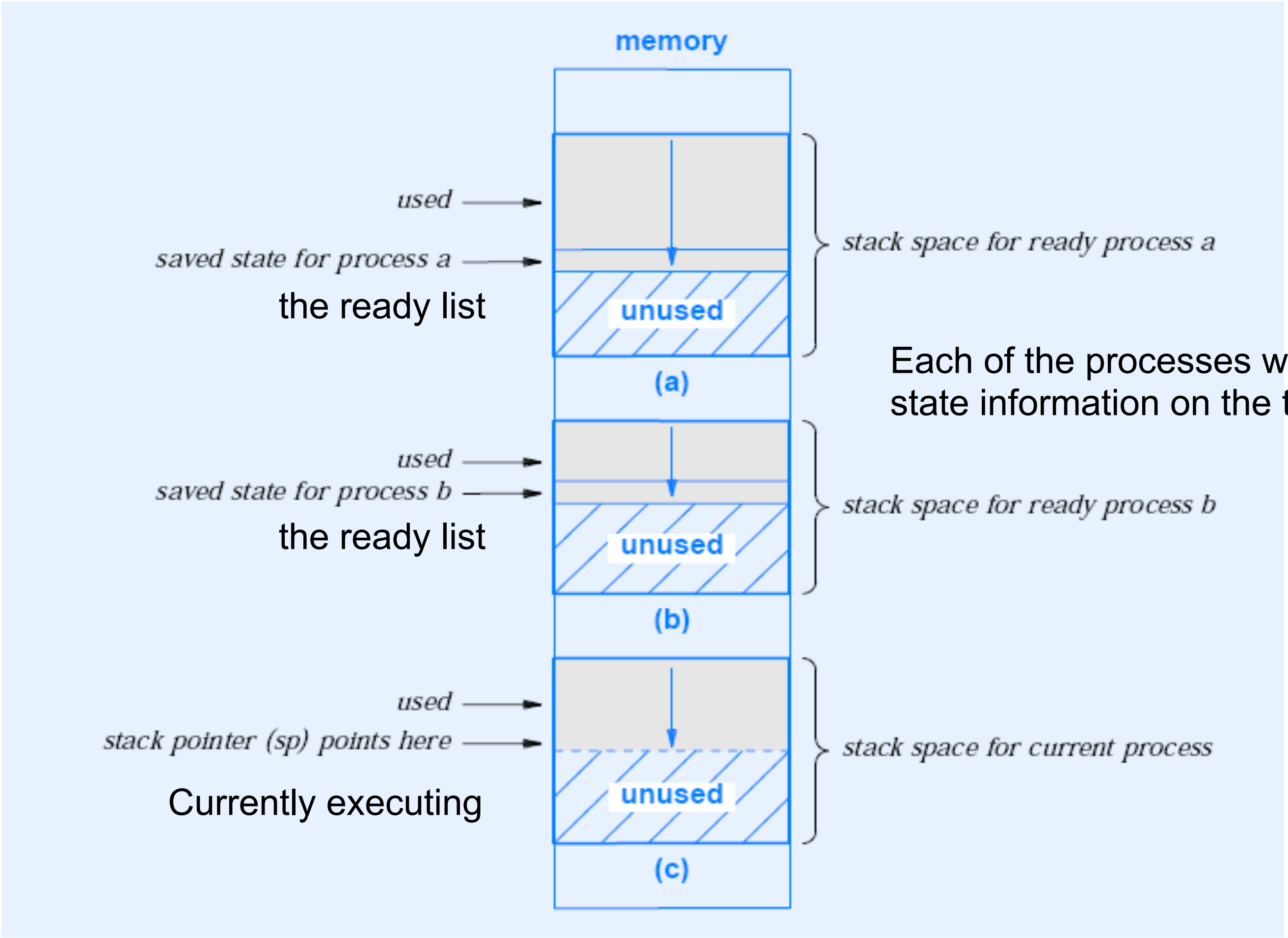
See whether rescheduling is deferred.

Indicate if an attempt was made during the deferral period and returns to the caller.

Implementation Of Context Switching in Xinu

- *resched* calls an assembly language function, *ctxsw*, to switch context from one process to another
- Reset the program counter (i.e., jumping to the location in the new process at which execution should resume)
- The text segment for the new process will be present in memory
- The RISC architecture contains a pair of instructions that are used in context switching:
  - Store processor state information in successive memory locations
  - Load processor state from successive memory locations

Illustration of State Saved on Process stack



- 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

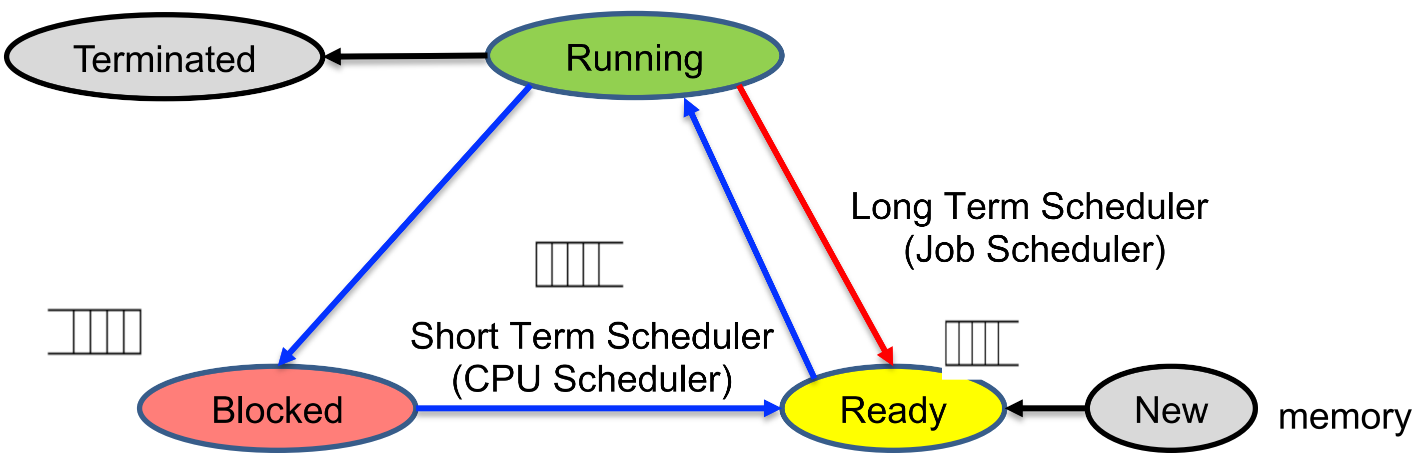
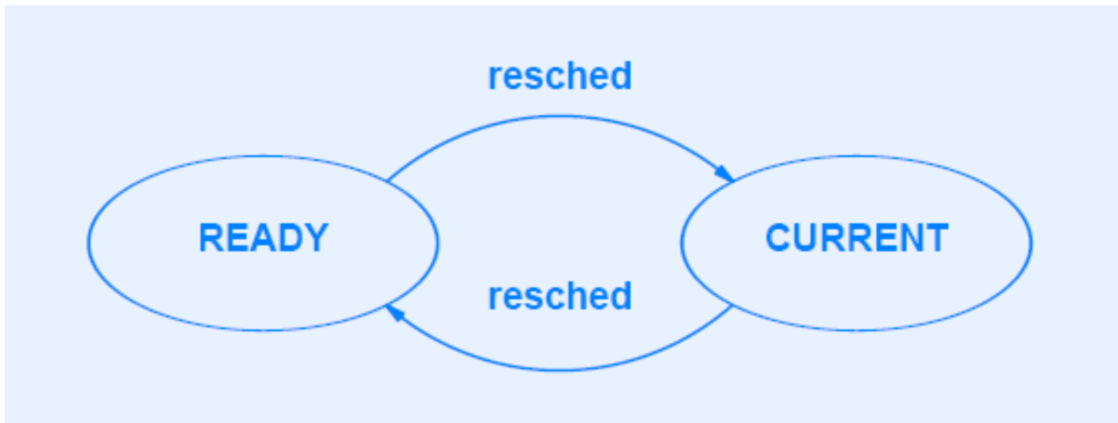


Illustration of Transitions between the Current and Ready States in Xinu



- The scheduler's only function is to switch the processor among the set of processes that are current or ready
- Single function (*resched*) moves a process in either direction between the two states

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
  - 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 (1/2)

/* 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     */
```

Saving values for the old process on the current stack

The flags contain the current processor status.

A single machine instruction to restore all the registers from the saved values

Example Context Switch Code - Intel (2/2)

```
/* 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            */
```

A single machine instruction to restore all the registers from the saved values

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		*/
```

The code uses assembler *directives* that the assembler interprets and uses to generate multiple instructions.

The co-processor stores the internal hardware status register.

Opposite order pop

Moves the status from the co-processor status register to a specified general-purpose register.

Question 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 Question 1

- 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

push r0	pop lr
push r1	pop r11
push r2	pop r10
push r3	pop r9
push r4	pop r8
...	pop r7
push lr	...
	pop r0

Question 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 (old or new) executes the context switch code?

Solution to Question 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

Question 3

- Our invariant says that at any time, one process must be executing
- All user processes may be idle (e.g., applications all wait for input)
- What happens if all processes wait for I/O?

Because an operating system can only switch the processor from one process to another, at least one process must remain ready to execute at all times.

- Which process executes?

Solution to Question 3

- To ensure that at least one process always executes, Operating system needs an extra process
  - Called the *NULL process*: process ID zero and priority zero
  - Never terminates
  - Typically an infinite loop
  - Cannot make a system call that takes it out of ready or current state

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

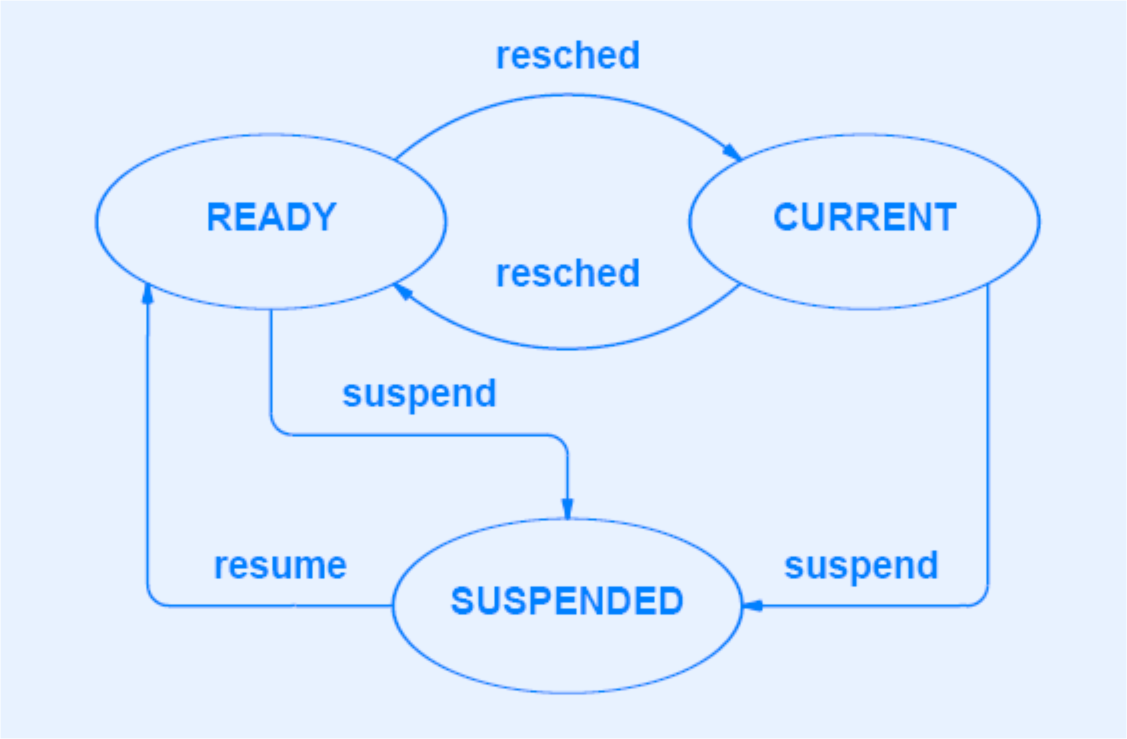
Process Manipulation

- 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 system calls from other functions in the OS
- Another way to look at interrupt handling focuses on an invariant that a system function must maintain:

An operating system function always returns to its caller with the same interrupt status as when it was called.

Template For System Calls

syscall *function_name*(*args*) {

intmask mask; /*Save interrupt mask*/

mask = disable( ); /* Disable interrupts at start of function*/

if (*args are incorrect*) {

restore(mask); /* Restore interrupts before error return*/

return(SYSERR);

}

Report status:
Indicate that an error occurred during processing

... *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*);

}

Report status:
Indicate that the operation is successfu

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;
     }
```

The function disables interrupts when it is invoked.

Verify that it is a valid process ID (ready/current)

Restores interrupts

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;
}
```

Suspend sets the state of the current process to the desired next state.

The key idea is that when a process suspends itself, the process remains executing until the call to *resched* selects another process and switches context.

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) || (pid == NULLPROC)) {
                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;
}
```

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;               /* Ptr to process' table entry */

          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;
}
```

If a function changes the state of processes, the function must call *resched()* to reestablish the invariant

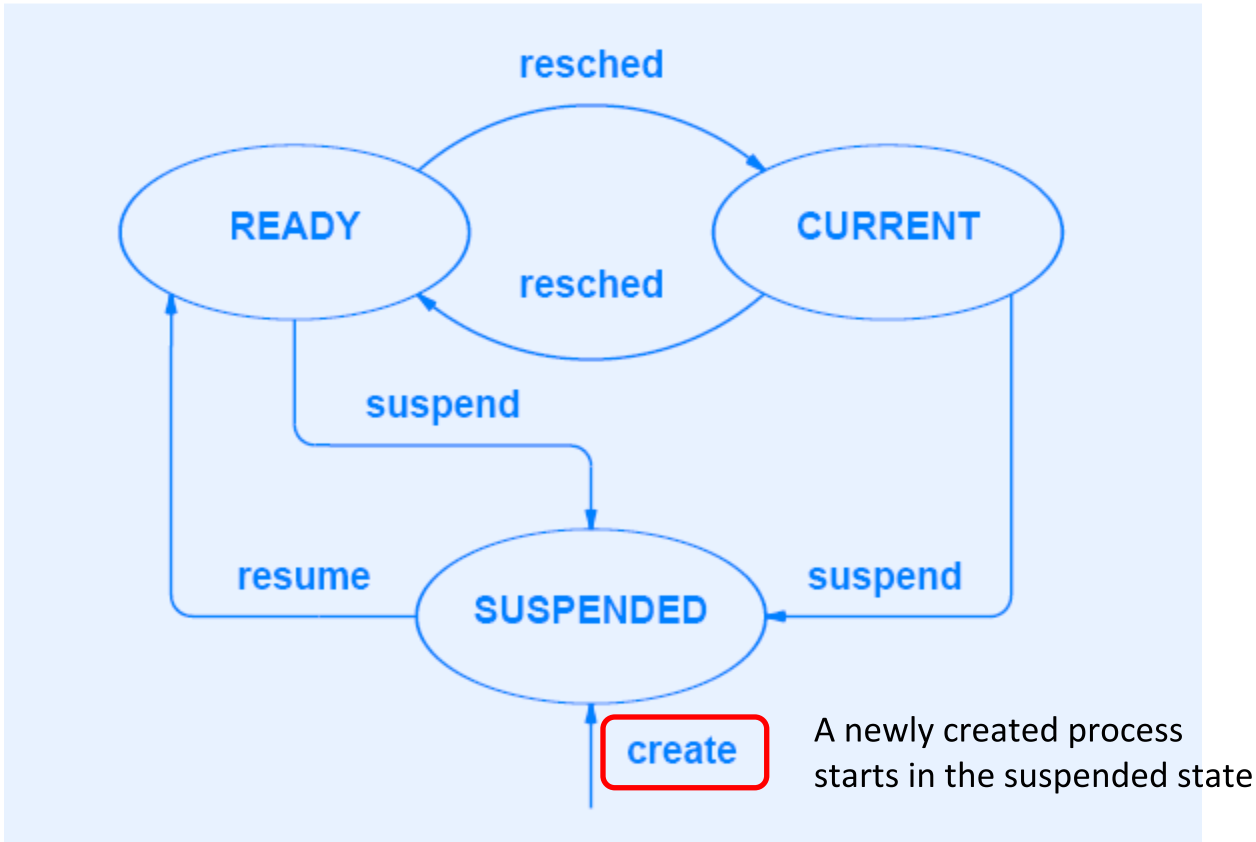
Process Termination

- A system call, *kill*, implements process termination by completely removing a process from the system.
- Record of the process is expunged
- Process table entry becomes available for reuse
- Known as *process exit* if initiated by the thread itself

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
    - *Creation* uses function *newpid* to search the table
  - Fill in entry
    - *Creation* uses function *getstk* to allocate space for the new process's stack
  - Place new process in *suspended* state

Illustration of State Transitions for Additional Process Management Functions



- Note that both current and ready processes can be suspended

Other Process Scheduling Algorithms

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

Summary (1/3)

- *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 (2/3)

- 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 (3/3)

- 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