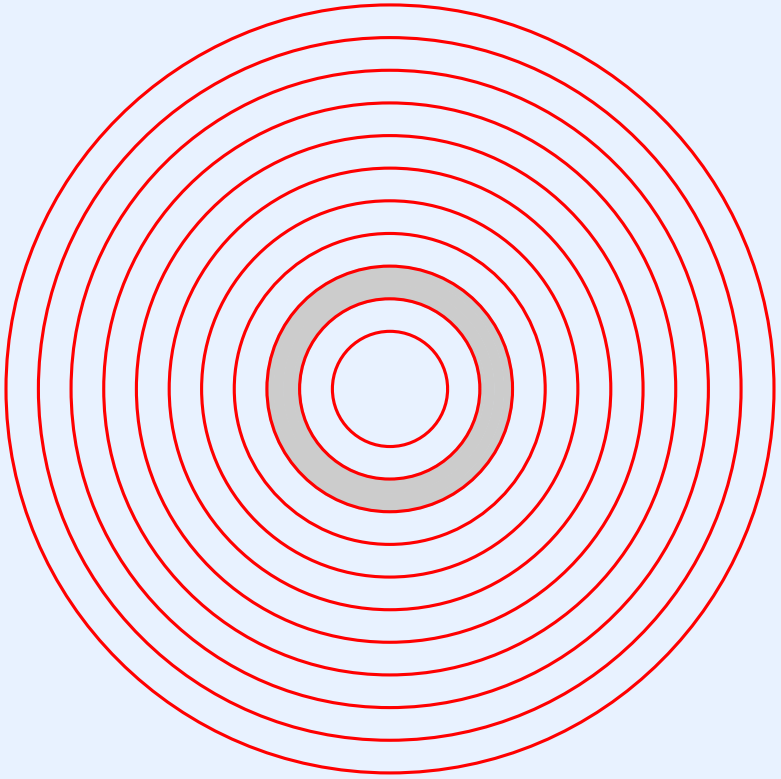
PART 4

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 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 Scheduling In The Hierarchy



Concurrent Processing

- Unit of computation
- Abstraction of a processor
  - Known only to operating system
  - Not known by hardware

The Operating System View

- All computation must be done by some process
  - No execution by the operating system
  - No execution “outside” of a process
- Key idea
  - A process must be running at all times

Concurrency Models

- Many variations have been used
  - *Job*
  - *Task*
  - *Thread*
  - *Process*
- Differences in
  - Address space and sharing
  - Coordination and communication mechanisms
  - Longevity
  - Dynamic or static definition

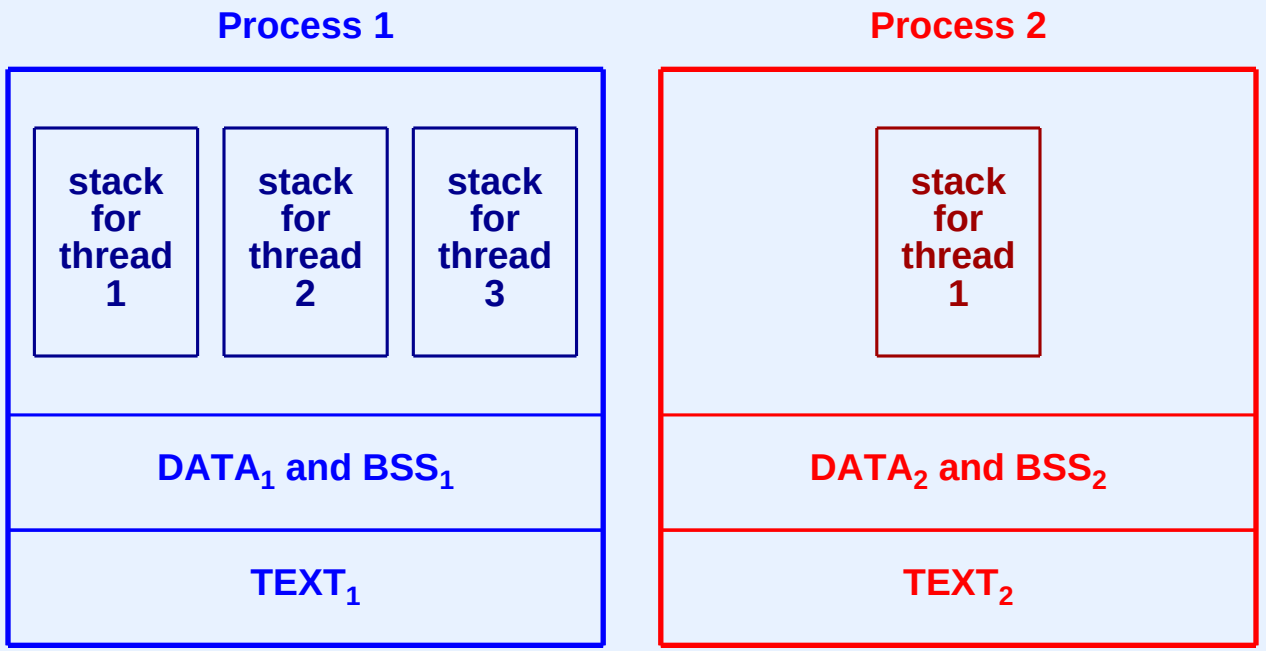
Thread Of Execution

- 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
  - Procedure calls

Process Abstraction

- Written with uppercase “P” to distinguish from generic notion
- Address space in which multiple threads can execute
- One data segment per Process
- One bss segment per Process
- Multiple threads per Process
- Each thread
  - Bound to a single Process
  - Cannot move to another Process

Illustration Of Two 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 Or Threads

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

Information Kept In A Process Table

- For each process
  - Unique *process identifier*
  - Owner (a user)
  - Scheduling priority
  - Location of code and data (stack)
  - Status of computation
  - Current program counter
  - Current values of registers

Information Kept In A Process Table (continued)

- If a 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 Model

- Simplest possible scheme
- Single-user system (no ownership)
- One global context
- One global address space
- No boundary between OS and applications
- Note Xinu “process” is technically a “thread”

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

- One “state” 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)

Example Declaration For Process States In Xinu

```
/* 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 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 computations eligible for execution
  - Select one
  - Switch CPU to selected process
- Three-level scheduling possible
  - Select User
  - Select Process owned by used
  - Select thread within Process

Implementation Of Scheduling

- Need a *scheduling policy* that specifies which process to select
- Build a scheduling function that
  - Selects a process according to the policy
  - Updates process table for current and selected process
  - Call *context switch* to switch from current to selected process

Scheduling Policy

- Fundamental part of OS
- Determines when process is selected for execution
- May depend on
  - User
  - How many processes a user owns
  - Whether each process contains multiple threads
  - Time a given process waits
  - Priority of process (or of threads)
- 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 chooses a process with highest priority
- Policy implemented by a system-wide invariant

The Xinu Scheduling Invariant

At any time, the CPU must run the 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 *ready list*
  - Order ready list by process priority
  - Selection of highest-priority process performed in constant time

Forcing Round-Robin Scheduling

- Operating system uses timer
- Whenever timer interrupts, if another equal-priority process is eligible for the CPU, switch to the other process
- We we will consider details later

High-Speed Scheduling

- Compare priority of current process to priority of first process on ready list
  - If current process priority higher, do nothing
  - Otherwise, call *context switch* to make process on ready list current

Xinu Scheduler Details

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

Scheduling And Equal Priority Processes

- Calling scheduler harmless if current process
  - Remains eligible, and
  - Has uniquely highest priority
- Scheduler causes context switch if current process
  - No longer eligible
  - Has priority less-than *or equal* to highest priority ready process
- Later, we will see why switching when a waiting process has equal priority is important

Example Scheduler Code (resched part 1)

```
/* except from resched.c - resched */

extern  void ctxsw(void *, void *);
/*------------------------------------------------------------------------
 *  resched  -  Reschedule processor to highest priority eligible process
 *------------------------------------------------------------------------
 */
int32   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   */

        ptold = &proctab[currpid]; /* current process' table entry */

        if (ptold->prstate == PR_CURR) {
                if (ptold->prprio > firstkey(readylist)) {
                        return OK;
                }

                /* old process will no longer remain current */

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

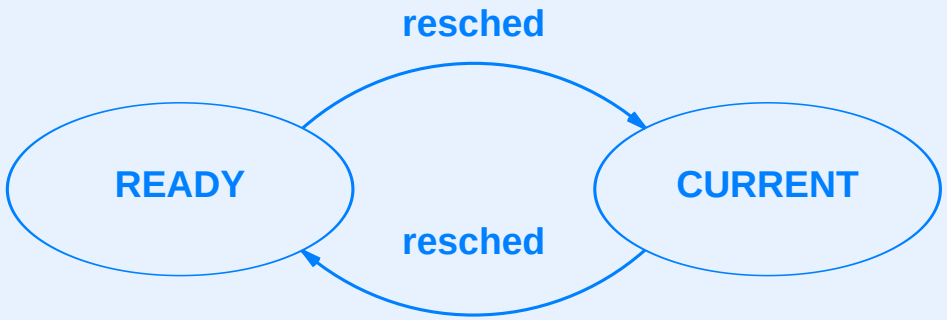
Example Scheduler Code (resched part 2)

```
        /* force context switch to highest priorty 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 OK;
}
```

Process State Transitions

- Recall each process has a “state”
- State determines
  - Whether an operation valid
  - Semantics of each operation
- Transition diagram documents valid operations

Illustration Of State Transition Between Current And Ready



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

Context Switch

- Basic facility in OS
  - Low-level (manipulates hardware state)
  - Written in assembly language
- Called by scheduler
- Moves CPU 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* in process table and / or on stack
  - Machine registers
  - Program counter
  - Privilege level
  - Memory maps
- Load information for *N*

Example Context Switch Code (MIPS part 1)

```
/* ctxsw.s - ctxsw */

        .align 4
        .globl  ctxsw

/*------------------------------------------------------------------------
 *  ctxsw  -  Switch from one process context to another
 *------------------------------------------------------------------------
 */
        .ent ctxsw
ctxsw:
        /* build context record on the current process' stack */

        addiu   sp, sp, -CONTEXT
        sw      ra, CONTEXT-4(sp)
        sw      ra, CONTEXT-8(sp)

        /* Save callee-save (non-volatile) registers */

        sw      s0, S0_CON(sp)
        sw      s1, S1_CON(sp)
        sw      s2, S2_CON(sp)
        sw      s3, S3_CON(sp)
        sw      s4, S4_CON(sp)
        sw      s5, S5_CON(sp)
        sw      s6, S6_CON(sp)
        sw      s7, S7_CON(sp)
```

Example Context Switch Code (MIPS part 2)

```
sw      s8, S8_CON(sp)
sw      s9, S9_CON(sp)

/* Save outgoing process' stack pointer */

sw      sp, 0(a0)

/* Load incoming process' stack pointer */

lw      sp, 0(a1)

/* At this point, we have switched from the run-time stack */
/*    of the outgoing process to the incoming process      */

/* Restore callee-save (non-volatile) registers from new stack  */

lw      s0, S0_CON(sp)
lw      s1, S1_CON(sp)
lw      s2, S2_CON(sp)
lw      s3, S3_CON(sp)
lw      s4, S4_CON(sp)
lw      s5, S5_CON(sp)
lw      s6, S6_CON(sp)
lw      s7, S7_CON(sp)
lw      s8, S8_CON(sp)
lw      s9, S9_CON(sp)
```

Example Context Switch Code (MIPS part 3)

```
        /* Restore argument registers for the new process */

        lw      a0, CONTEXT(sp)
        lw      a1, CONTEXT+4(sp)
        lw      a2, CONTEXT+8(sp)
        lw      a3, CONTEXT+12(sp)

        /* Remove context record from the new process' stack */

        lw      v0, CONTEXT-4(sp)
        lw      ra, CONTEXT-8(sp)
        addiu   sp, sp, CONTEXT

        /* If this is a newly created process, ensure */
        /*    it starts with interrupts enabled       */

        beq     v0, ra, ctxdone
        mfc0    v1, CP0_STATUS
        ori     v1, v1, STATUS_IE
        mtc0    v1, CP0_STATUS

ctxdone:
        jr      v0
        .end ctxsw
```

Example Context Switch Code (X86 part 1)

```
/* ctxsw.s - ctxsw */

                .text
                .globl  ctxsw
newmask:        .word   0

/* excerpt from ctxsw on an X86 architecture.  */
/* args: &oldsp, &oldmask, &newsp, &newmask    */

ctxsw:
                pushl   %ebp
                movl    %esp,%ebp

                pushl   12(%ebp)
                call    disable
                movl    20(%ebp),%eax
                movw    (%eax),%dx
                movw    %dx,newmask
                pushfl                  /* save flags */
                pushal                  /* save general regs */

                /* save segment registers here, if multiple allowed */

                movl    8(%ebp),%eax
                movl    %esp,(%eax)     /* save old SP */
```

Example Context Switch Code (X86 part 2)

```
/* restore new segment registers here, if multiple allowed */
popal                   /* restore general registers */
popfl                   /* restore flags */
pushl   $newmask
call    restore
leave
ret
```

Puzzle #1

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

Solution To Puzzle #1

- “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 #2

- Invariant says that at any time, one process must be executing
- All user processes may be idle (e.g., applications all wait for input)
- Which process executes?

Solution To Puzzle #2

- OS needs an extra process
  - Called *NULL process*
  - Never terminates
  - 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)

Null Process Code

- Typical null process

```
while(1)
   ;
```

- May not be optimal

Puzzle #3

- Null process must always remain ready to execute
- Null process should avoid using bus because doing so "steals" cycles from I/O activity
- Instructions reside in memory, so merely fetching instructions use the bus
- How can a null process avoid using the bus?

Two Solutions To Puzzle #3

- Solution #1
  - Halt the CPU until interrupt occurs
  - Special hardware instruction required
- Solution #2
  - Install an instruction cache
  - Processor fetches instructions from cache when possible
  - Avoids using bus when executing tight loop

More Process Management

Process Manipulation

- A process does not exist forever and does not perform computation continuously
- Need to invent ways to control processes
- Example operations
  - Suspension
  - Resumption
  - Creation
  - Termination
- State variable in process table records activity

Process Suspension

- Temporarily “stop” a process
- Prohibit from using the CPU
- To allow later resumption
  - Process table entry retained
  - Complete state of computation saved

Example Suspension Code (suspend part 1)

```
/* excerpt from suspend.c - suspend */

/*------------------------------------------------------------------------
 *  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 (suspend 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 reschedule */
        }
        prio = prptr->prprio;
        restore(mask);
        return prio;
}
```

Process Resumption

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

Example Resumption Code

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

Example Code Make A Process Ready (part 1)

```
/* 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  */
          bool8         resch           /* reschedule afterward?        */

        )
{
        register struct procent *prptr;

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

Example Code Make A Process Ready (part 2)

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

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

        if (resch == RESCHED_YES) {
                resched();
        }
        return(OK);
}
```

- Note: ready assumes that interrupts are disabled

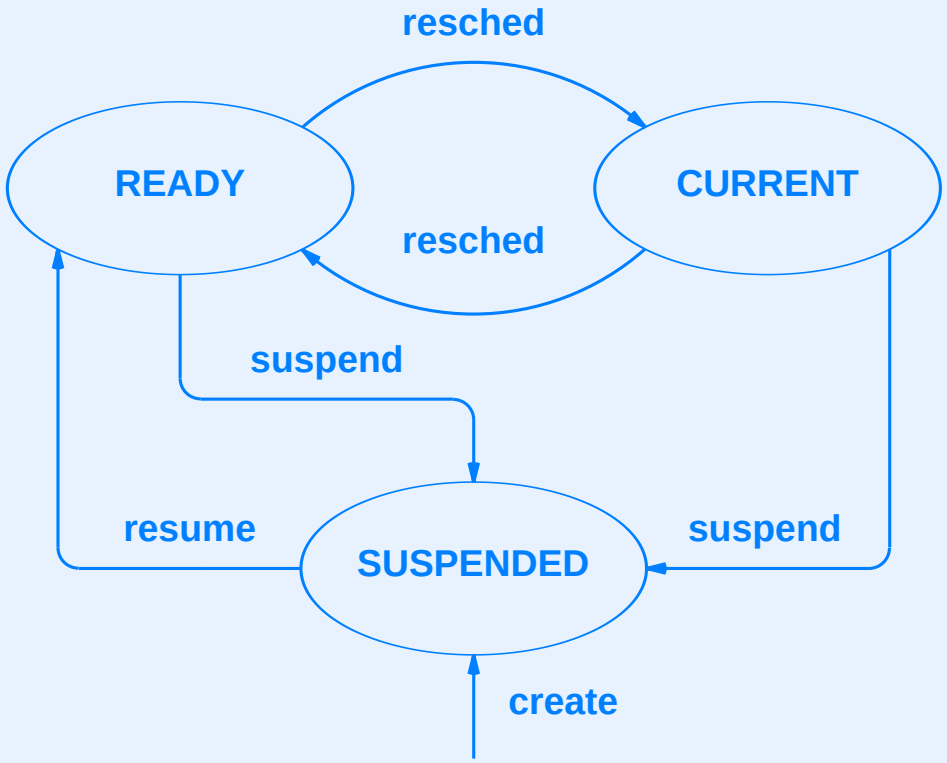
Process Termination

- 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

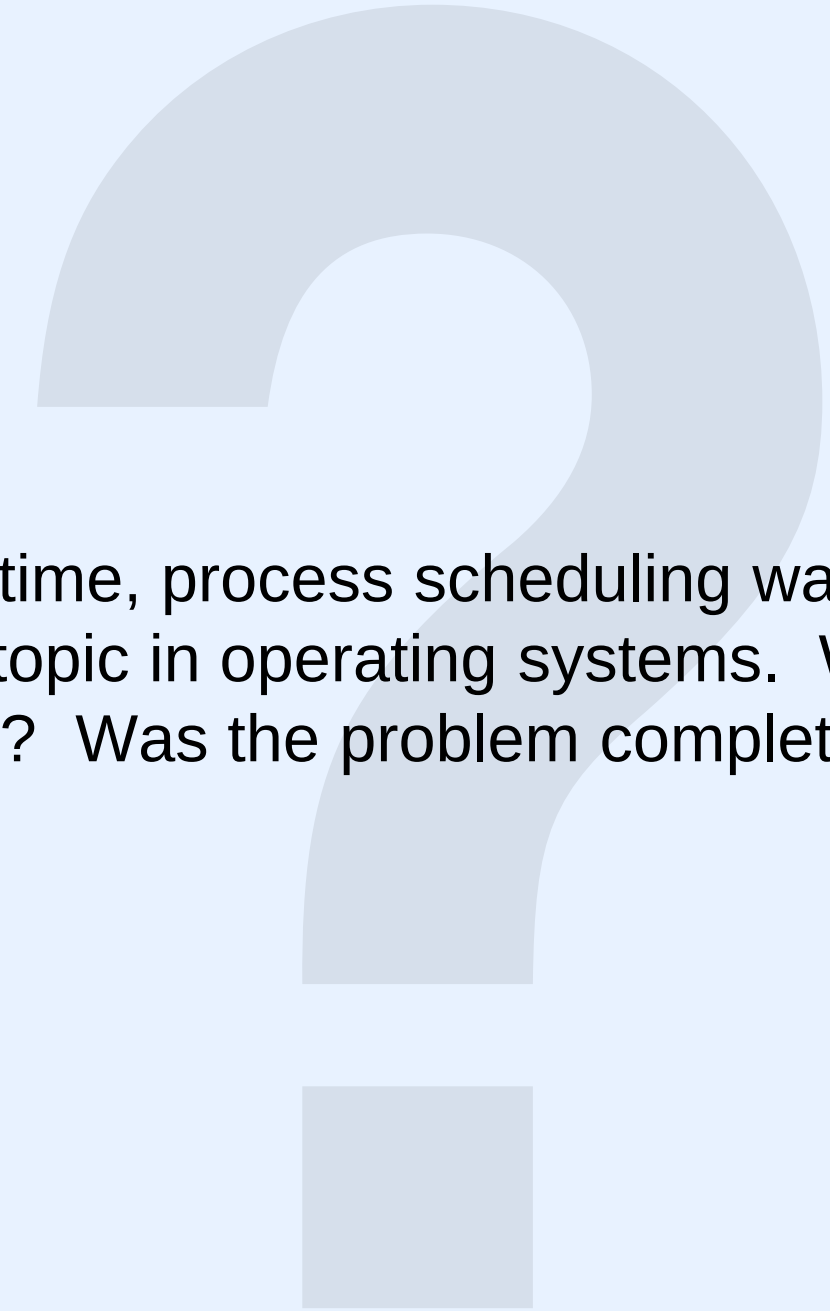
- Processes are dynamic — process *creation* refers to starting a new process
- Performed by *create* procedure in Xinu
- Method
  - Find free entry in process table
  - Fill in entry
  - Place new process in *suspended* state
- We will see more about creation later

Illustration Of State Transitions For Additional Process Management Functions



System Calls

- Define interface from applications to OS
- Define OS characteristics
- Conceptually like procedure calls
- Transfer to kernel address space
- Note: for 503 version of Xinu
  - System calls *are* procedure calls
  - *syscall* clarifies intent



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 or 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
- At any time a process must be executing
- Processes can be suspended, resumed, created, and terminated
- Special process known as *null process* remains ready to run at all times