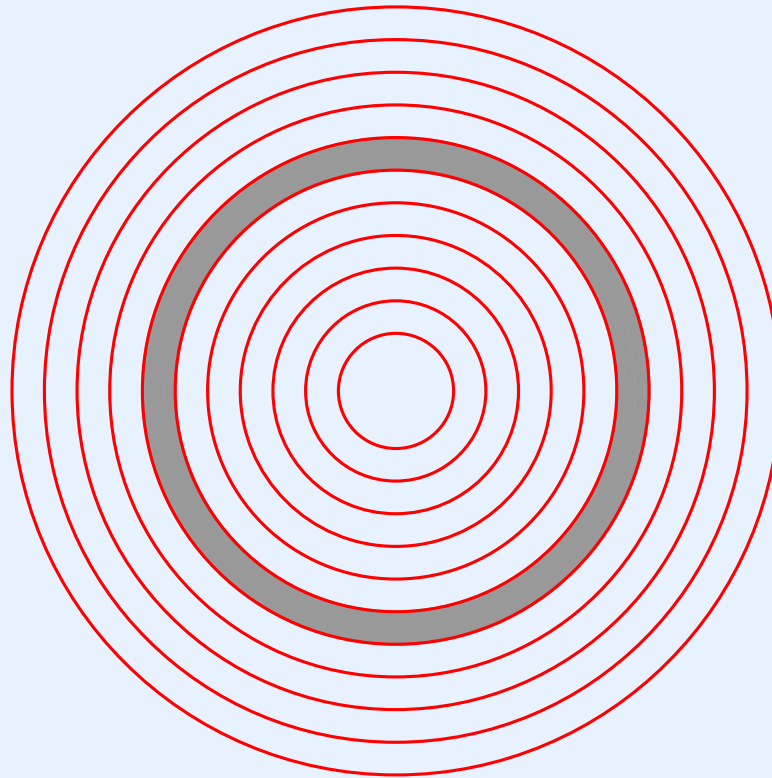


# **Module IX**

## **Device Management**

# Location Of Device Management In The Hierarchy



# Ancient History

- Each device had unique hardware interface
- Code to communicate with device was part of application
- Application polled the device; interrupts were not used
- Disadvantages
  - Generality and portability were limited
  - Painful to program

# Modern Approach

- Device manager is part of OS
- OS presents applications with uniform interface to all devices (as much as possible)
- All I/O is *interrupt-driven*

# Device Manager In An Operating System

- Manages peripheral resources
- Hides low-level hardware details
- Provides API to applications
- Synchronizes processes and I/O

# A Conceptual Note

**One of the most intellectually difficult aspects of operating systems arises from the interaction between processes (an OS abstraction) and devices (a hardware reality). Specifically, the connection between interrupts and scheduling can be tricky because an interrupt that occurs in one process can enable another.**

# Review Of Hardware Interrupts

- Processor
  - Starts a device
  - Enables interrupts
- Device
  - Performs the requested operation
  - Raises an interrupt on bus
- Processor hardware
  - Checks for interrupts after each instruction is executed
  - Invokes an interrupt function, if an interrupt is pending
  - Provides a mechanism for atomic return

# Processes And Interrupts

- Key ideas
  - Recall: at any time, a process is running
  - We think of an interrupt as a function call that occurs “between” two instructions
  - Processes are an OS abstraction, not part of the hardware
  - OS cannot afford to switch context whenever an interrupt occurs
- Consequence:

**The current process executes interrupt code**



# Historic Interrupt Software

- Separate interrupt function for each device
  - Low-level
  - Handles housekeeping details
    - \* Saves / restores registers
    - \* Sets interrupt mask
  - Finds interrupting device on the bus
  - Interacts with the device to transfer data
  - Resets the device for the next interrupt
  - Returns from interrupt

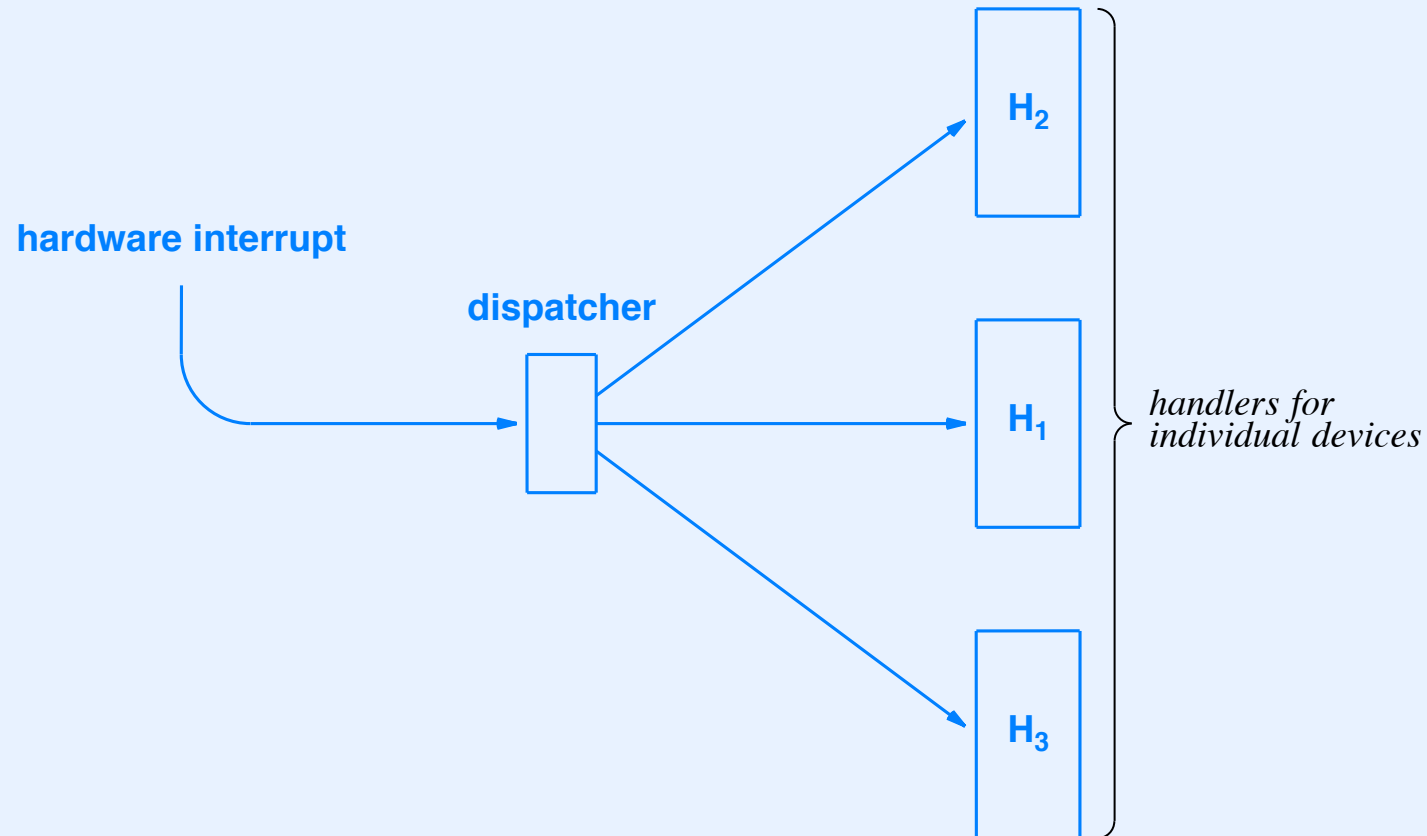
# Modern Interrupt Software (Two Pieces)

- *Interrupt dispatcher*
  - Single function common to all interrupts
  - Performs housekeeping details
  - Finds interrupting device on the bus
  - Calls a device-specific function
- *Interrupt handler*
  - Separate code for each device
  - Invoked by the dispatcher
  - Performs all interaction with device

# Interrupt Dispatcher

- Low-level function
- Invoked by hardware when interrupt occurs
  - Is invoked with correct mode (i.e., with interrupts disabled)
  - Hardware has saved the instruction pointer
- Dispatcher
  - Saves other machine state as necessary
  - Identifies interrupting device
  - Establishes high-level runtime environment (needed for a C function)
  - Calls a device-specific *interrupt handler*

# Conceptual View Of Interrupt Dispatching



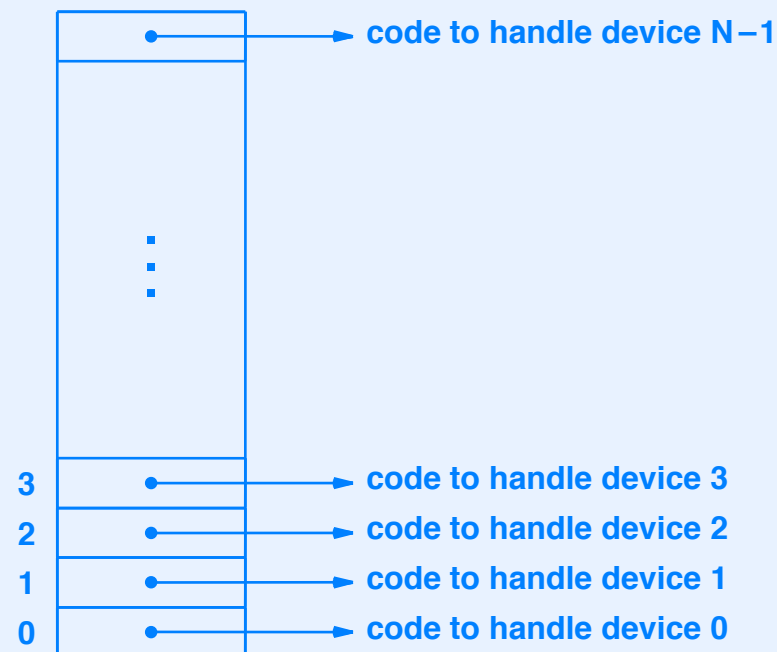
- Note: the dispatcher is typically written in assembly language

# Return From Interrupt

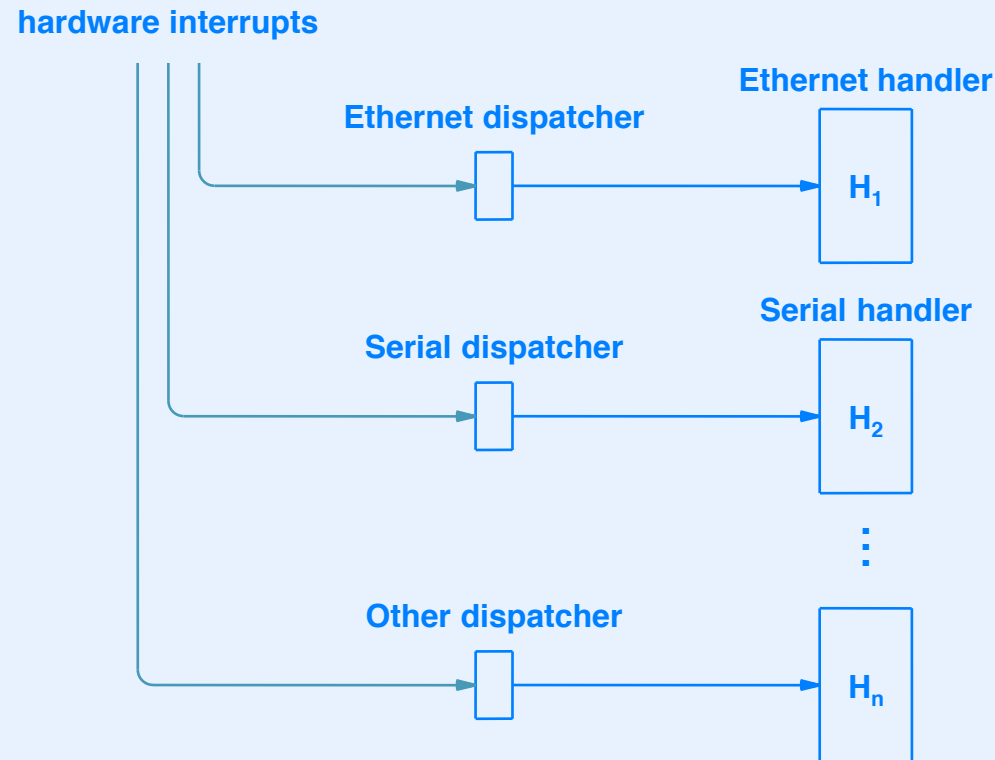
- Handler
  - Communicates with device
  - May restart next operation
  - Eventually returns to interrupt dispatcher
- Interrupt dispatcher
  - Executes special hardware instruction known as *return from interrupt*
- Return from interrupt instruction atomically
  - Resets instruction pointer to saved value
  - Enables interrupts

# Interrupt Mechanism: A Vector

- Each possible interrupt is assigned a unique IRQ
- Hardware uses IRQ as an index into an *interrupt vector* array
- Conceptual organization

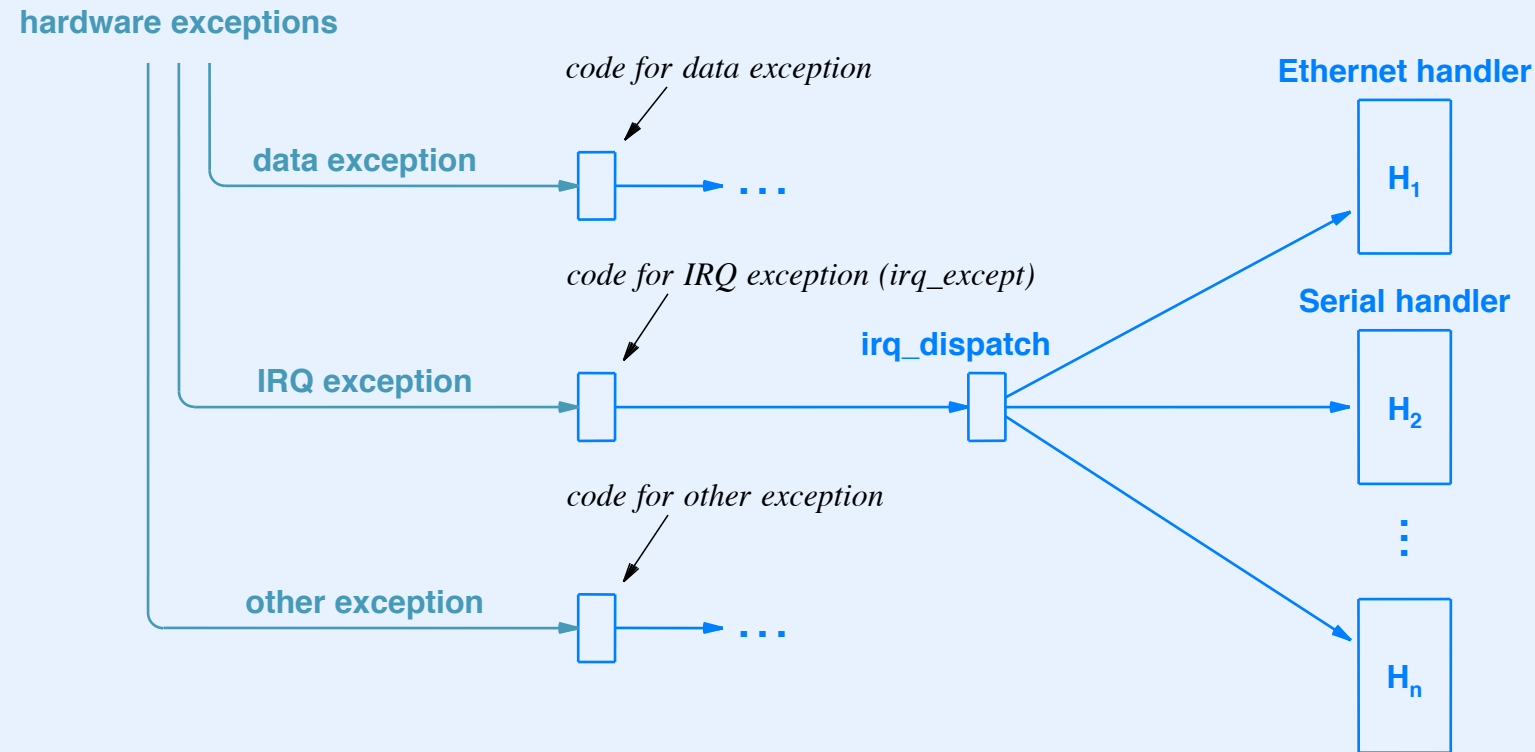


# Interrupt Code On A Galileo (x86)



- Operating system loads the interrupt controller with the address of a dispatcher for each device
- Controller invokes the correct dispatcher

# Interrupt Code On A BeagleBone Black (ARM)



- Two-level scheme where controller uses *IRQ exception* for all device interrupts
- Exception code invokes the IRQ dispatcher



# A Basic Rule For Interrupt Processing

- Facts
  - The processor disables interrupts before invoking the interrupt dispatcher
  - Interrupts remain disabled when the dispatcher calls a device-specific interrupt handler
- Rule
  - To prevent interference, an interrupt handler must keep interrupts disabled until it finishes touching global data structures, ensures all data structures are in a consistent state, and returns
- Note: we will consider a more subtle version later

# Interrupts And Processes

- When an interrupt occurs, I/O has completed
- Either
  - Data has arrived
  - Space has become available in an output buffer
- A process may have been blocked waiting
  - To read data
  - To write data
- The blocked process may have a higher priority than the currently executing process
- The scheduling invariant *must* be upheld

# A Question About Scheduling

- Suppose process  $X$  is executing when an interrupt occurs
- Process  $X$  remains executing when the interrupt dispatcher is invoked and when the dispatcher calls a handler
- Suppose data has arrived and a higher-priority process  $Y$  is waiting for the data
- If the handler merely returns from an interrupt, process  $X$  will continue to execute
- Should the interrupt handler call *resched*?
- If not, how is the scheduling invariant established?

## Possible Answers

- An OS may
  - Have an interrupt handler reestablish the scheduling invariant
  - Arrange for the dispatcher to reestablish the scheduling invariant just before returning from the interrupt
  - Postpone rescheduling until a later time (e.g., when the current process's time-slice expires)
- Any of the above works
- Placing a check in the dispatcher incurs more overhead

# Interrupts And The Null Process

- In the concurrent processing world
  - A process is always running
  - Interrupts can occur asynchronously
  - The currently executing process executes interrupt code
- An important consequence
  - The null process may be running when an interrupt occurs
  - If interrupted, the null process will execute the interrupt handler
- Keep in mind: the null process must always remain eligible to execute

# **A Restriction On Interrupt Handlers Imposed By The Null Process**

**Because an interrupt can occur while the null process is executing, an interrupt handler can only call functions that leave the executing process in the current or ready states. For example: an interrupt handler can call send or signal, but cannot call wait.**

# A Question About Scheduling And Interrupts

- Recall that
  - Interrupts are disabled when dispatcher calls device-specific interrupt handler
- To remain safe
  - A device-specific interrupt handler must keep further interrupts disabled until it completes changes to global data structures
- Can an interrupt handler call a function that calls *resched*?

# Rescheduling During Interrupt Processing

- Suppose
  - Interrupt handler calls *signal*
  - *Signal* calls *resched*
  - *Resched* switches to a new process
  - The new process executes with interrupts enabled
- Question
  - Will interrupts pile up indefinitely?

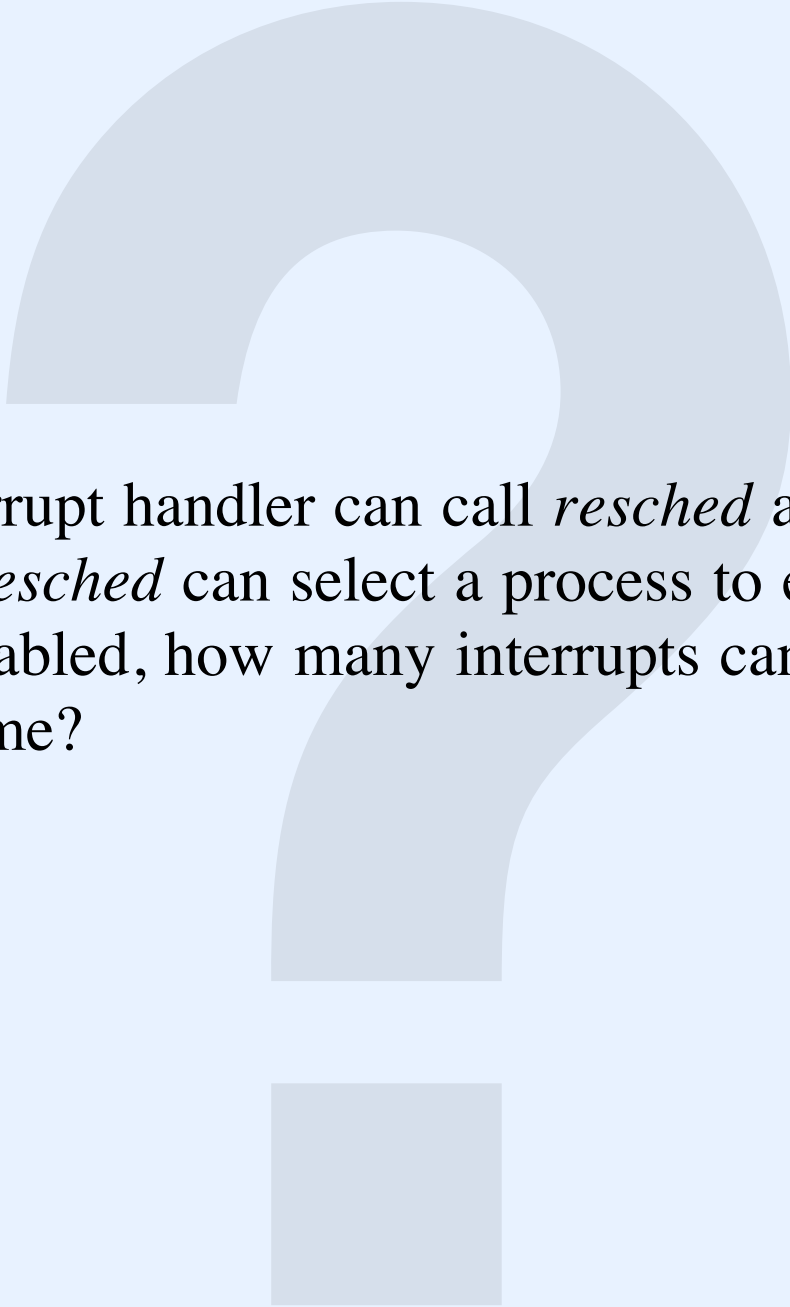


## An Example

- Let  $T$  be the current process
- When interrupt occurs,  $T$  executes an interrupt handler
- The interrupt handler calls *signal*
- *Signal* calls *resched*
- A context switch occurs and process  $S$  runs
- $S$  may run with interrupts enabled

# The Answer

**Rescheduling during interrupt processing is safe provided that each interrupt handler leaves global data in a valid state before rescheduling and no function enables interrupts unless it previously disabled them (i.e., uses disable/restore rather than enable).**



If an interrupt handler can call *resched* after restarting a device and *resched* can select a process to execute that has interrupts enabled, how many interrupts can be outstanding at a given time?

# **Device Driver Organization**

# Device Driver

- Set of functions that perform I/O on a given device
- Contains device-specific code
- Includes functions used to read or write data and control the device as well as interrupt handler code
- Code divided into two conceptual parts

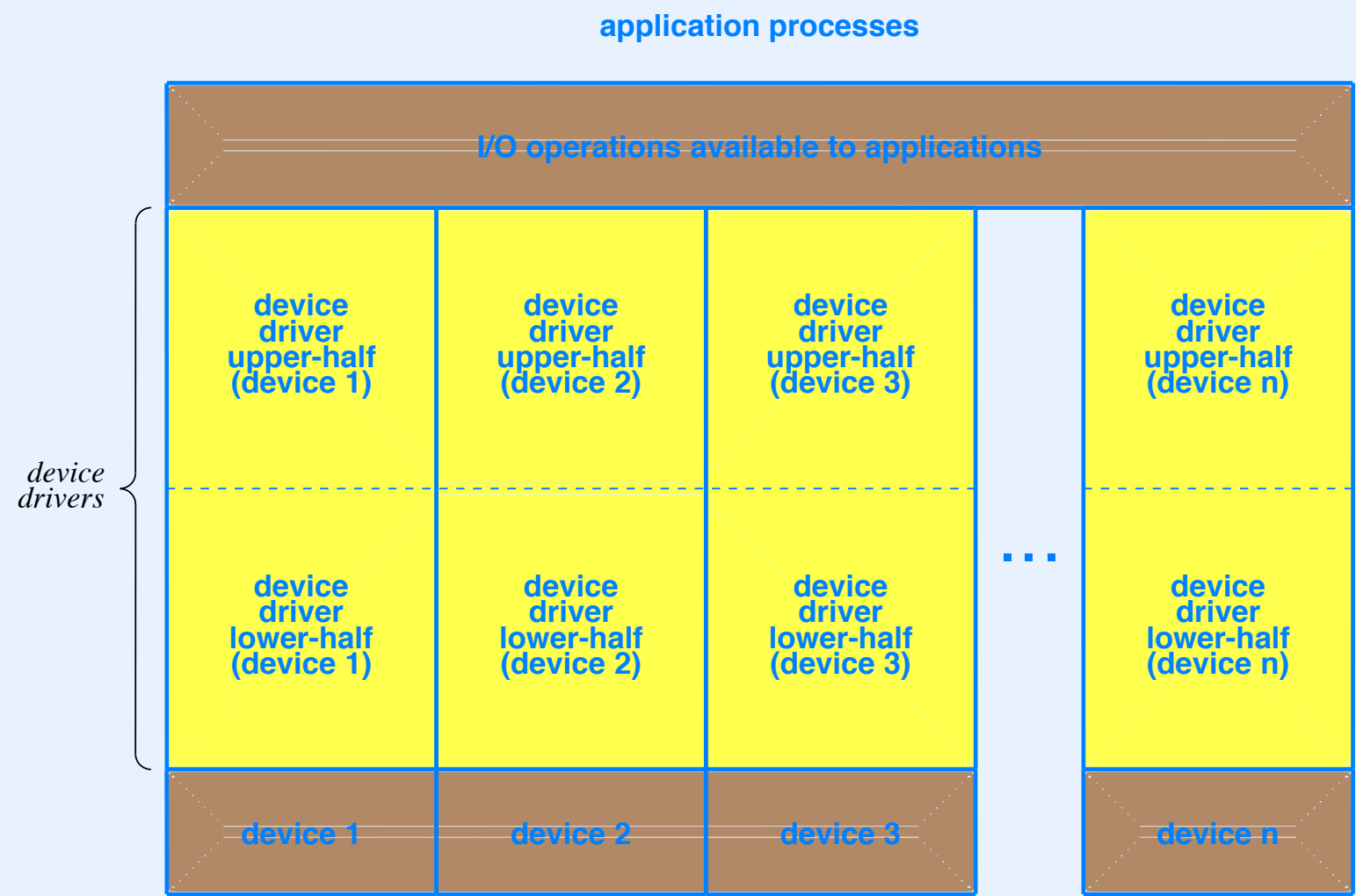
# Two Conceptual Parts Of A Device Driver

- Upper half
  - Functions executed by an application
  - Used to request I/O
  - May copy data between user and kernel address spaces
- Lower half
  - Device-specific interrupt handler
  - Invoked by interrupt when operation completes
  - Executed by whatever process is executing
  - May restart the device for next operation

# Division Of Duties In A Driver

- Upper-half
  - Minimal interaction with device hardware
  - Enqueues request
  - Starts device if idle
- Lower-half
  - Minimal interaction with application
  - Talks to the device
    - \* Obtains incoming data
    - \* Starts next operation
  - Starts process if any waiting

# Conceptual Organization Of Device Software





# Coordination Of Processes Performing I/O

- Process may need to block when attempting to perform I/O
- Examples
  - Application waits for incoming data to arrive
  - Application blocks until the device is ready to send outgoing data
- How should coordination be performed?

# Answer

- No need to invent new mechanisms; standard process coordination mechanisms suffice
  - Message passing
  - Semaphores
  - Suspend / resume
- However: a programmer must exercise caution because a lower-half function must *not* block the calling process

# Using Message Passing For Input Synchronization

- During input operation (*read*)
- Upper-half
  - Starts the device
  - Places the current process ID in a data structure associated with device
  - Calls *receive* to block
- Lower-half
  - Is invoked during the interrupt when input completes
  - Calls *send* to send a message to the blocked process

# Using Semaphores For Input Synchronization

- A shared buffer is created with  $N$  slots
- A semaphore is created with initial count  $0$
- Upper-half
  - Calls *wait* on the semaphore
  - Extracts the next item from buffer and returns
  - Can restart the device if the device is idle
- Lower-half
  - Places the incoming item in the buffer
  - Calls *signal* on the semaphore
- Note: the semaphore counts items in the buffer

# Semaphores And Output Synchronization

# Semaphores And Output Synchronization

- A flawed approach
  - A semaphore counts items in the output buffer
  - The upper-half deposits an item and calls *signal*
  - The lower-half calls *wait* to block until an item is present

# Semaphores And Output Synchronization

- A flawed approach
  - A semaphore counts items in the output buffer
  - The upper-half deposits an item and calls *signal*
  - The lower-half calls *wait* to block until an item is present
- Why is the above flawed?

# Semaphores And Output Synchronization

- A flawed approach
  - A semaphore counts items in the output buffer
  - The upper-half deposits an item and calls *signal*
  - The lower-half calls *wait* to block until an item is present
- Why is the above flawed?

**Lower-half functions cannot execute *wait*!**



# Using Semaphores For Output Synchronization

- Trick: semaphore count interpreted as “space available”
  - Initialized to buffer size
- Upper-half
  - Calls *wait* on the semaphore
  - Deposits a data item in the buffer
  - Starts the device, if necessary
- Lower-half
  - Is invoked when an output operation completes
  - Starts next output operation, if the buffer is not empty
  - Signals the semaphore to indicate that space is available

# **Device-independent I/O**

# Application Interface To Devices

- Desires
  - Portability across machines
  - Generality sufficient for all devices
  - Elegance (minimalism rather than a potpourri of functions)
- Solution
  - Isolate application processes from device drivers
  - Use a common paradigm across all devices
  - Integrate with other operating system facilities

# Approach To Device-independent I/O

- Define a set of abstract operations
- Build a general-purpose mechanism
  - Use generic operations (e.g., *read*)
  - Include parameters to specify device instance
  - Arrange an efficient way to map generic operation onto code for a specific device
- Notes
  - The set of generic operations form an abstract data type
  - The upper-half of each driver must define functions that apply each generic operation to the device

# Device-Independent I/O Primitives In Xinu

init      – initialize device (invoked once, at system startup)  
open      – make the device ready  
close     – terminate use of the device  
read      – input arbitrary data from the device  
write     – output arbitrary data to the device  
getc      – input a single character from the device  
putc      – output a single character to the device  
seek      – position the device  
control   – control the device and / or the driver

- Note: some abstract functions may not apply to a given device

# Implementation Of Device-Independent I/O In Xinu

- Application process
  - Makes calls to device-independent functions (e.g., *read*)
  - Supplies the device ID as parameter (e.g., ETHER)
- OS
  - Maps the device ID to an actual device
  - Invokes appropriate device-specific function (e.g., *ethread* to read from an Ethernet)

# Mapping Generic I/O Function To A Device-specific Function

- Mapping must be efficient
- Is performed with a *device switch table*
  - Kernel data structure initialized when system loaded
  - One row per device
  - One column per operation
  - Each entry in the table points to a function
- The device ID is used as an index into the table

# Semantics Of Device-independent I/O



# Semantics Of Device-independent I/ O

- Each device-independent operation is generic

# Semantics Of Device-independent I/O

- Each device-independent operation is generic
- An operation may not make sense for a given device
  - *Seek* on keyboard, network, or display screen
  - *Close* on a mouse

# Semantics Of Device-independent I/O

- Each device-independent operation is generic
- An operation may not make sense for a given device
  - *Seek* on keyboard, network, or display screen
  - *Close* on a mouse
- However...

# Semantics Of Device-independent I/O

- Each device-independent operation is generic
- An operation may not make sense for a given device
  - *Seek* on keyboard, network, or display screen
  - *Close* on a mouse
- However... all entries in device switch table must be valid

# Semantics Of Device-independent I/O

- Each device-independent operation is generic
- An operation may not make sense for a given device
  - *Seek* on keyboard, network, or display screen
  - *Close* on a mouse
- However... all entries in device switch table must be valid
- Our solution: create functions that can be used to fill in meaningless entries

# Special Entries Used In The Device Switch Table

- *ionull*
  - Used for innocuous operation
  - Returns *OK*
- *ioerr*
  - Used for incorrect operation
  - Returns *SYSERR*

# Illustration Of Device Switch Table

*device* ↓

*operation* →

	open	read	write	
CONSOLE	&ttyopen	&ttyread	&ttywrite	
SERIAL0	&ionull	&comread	&comwrite	
SERIAL1	&ionull	&comread	&comwrite	...
ETHER	&ethopen	&ethread	&ethwrite	

⋮

- Each row corresponds to device
- Each column corresponds to operation
- Each entry specifies the address of a function to invoke

# Replicated Devices

- Computer may contain multiple copies of a physical device
- Example: two serial lines or two Ethernet NICs
- Goal
  - Have a single copy of the device driver functions
  - Allow a function to be used with any copy of the device



# Parameterized Device Drivers

- A driver must
  - Know which physical copy of a device to use
  - Keep the information for one copy separate from the information for other copies
- Technique
  - Assign each instance of a replicated device a unique minor number (e.g., 0, 1, 2, ...) known as a *minor device number*
  - Store the minor device number in the device switch table (each column includes a minor device number)

# Device Names

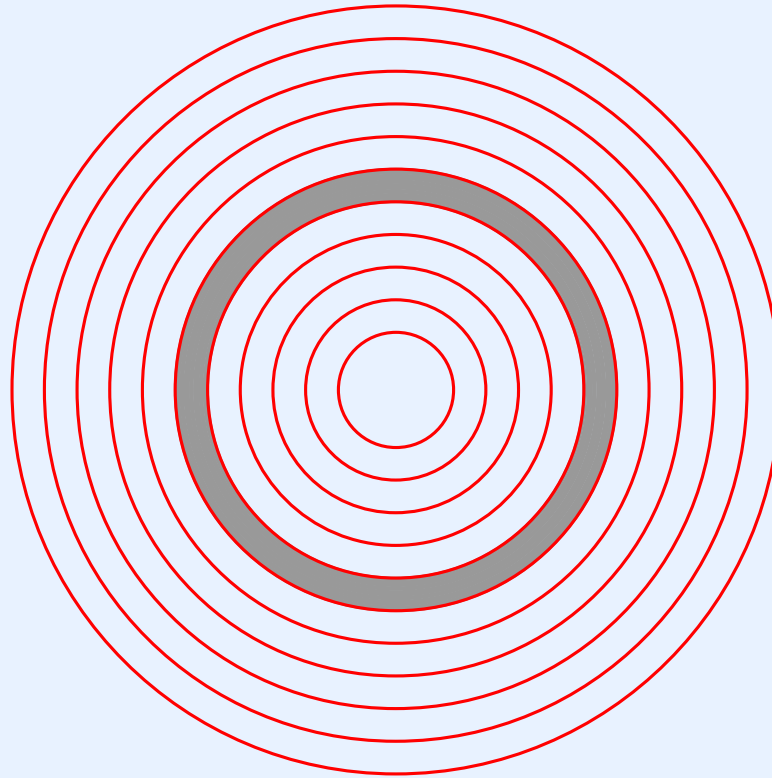
- Previous examples have shown device names used in code
  - CONSOLE
  - SERIAL0
  - SERIAL1
  - ETHER
- The device switch table is an array, and each device name is an index
- How are unique values assigned to names, such as *CONSOLE*?
- Answer: OS designer specifies names during configuration
- More details later

# Initializing The I/O Subsystem

- Required steps
  - Fill in the device switch table
  - Fill in interrupt vectors
  - Initialize data structures, such as shared buffers
  - Create semaphores used for coordination
- Xinu approach
  - Device switch table configured at compile time
  - At startup, Xinu calls *init* for each device
- We will see more later

# **Time Management**

# Location Of Clock Management In The Hierarchy



# Clock Hardware

- Processor clock
  - Controls processor rate
  - Often cited as processor speed
  - Usually not visible to the OS
- Time-of-day clock
  - Can be set or read by the processor
  - Operates independently from processor

# **Module X**

## **Clock And Timer Management**

# Clock Hardware

## (continued)

- Real-time clock
  - Pulses regularly
  - Called *programmable* if rate can be controlled by OS
  - Interrupts the processor on each pulse
  - Does *not* count pulses; an interrupt can be lost
  - Consequence: an OS cannot leave interrupts disabled for more than one clock tick or the real-time clock will be inaccurate
  - Note: some real-time clock hardware has a small counter for missed interrupts, which allows OS to correct counts later if only a few are missed



# Clock Hardware

## (continued)

- Interval timer
  - Hardware device that operates asynchronously
  - Timeout value  $T$  is set by processor, and device interrupts  $T$  time units later
  - Processor may be able to find the time remaining

# Timed Events

- *Hard real-time* system requires each event to occur at an exact time
- *Soft real-time* system requires an event to occur on or after the specified time
- In theory, an OS may need to handle many event types
- In practice, only two basic event types suffice

# Two Principle Types Of Timed Events

- *Preemption event*
  - Implements *timeslicing* by switching the processor among ready processes
  - Guarantees a given process cannot run forever
  - A preemption event is scheduled during a context switch
  - Cancellation is important (few processes exhaust their timeslice)
- *Sleep event*
  - Scheduled by a process
  - Delays the calling process a specified time

# Time-slicing Trade-Off

- How large should a timeslice be?

# Time-slicing Trade-Off

- How large should a timeslice be?
- Small granularity
  - Advantage: guarantees fairness because processes proceed at approximately equal rate
  - Disadvantage: increased rescheduling overhead

# Time-slicing Trade-Off

- How large should a timeslice be?
- Small granularity
  - Advantage: guarantees fairness because processes proceed at approximately equal rate
  - Disadvantage: increased rescheduling overhead
- Large granularity
  - Advantage: lower rescheduling overhead
  - Disadvantage: unfair because one process may be far ahead of another

# Timeslicing And Conventional Applications

**Most applications are I/O bound, which means the application is likely to perform an operation that takes the process out of the current state before its timeslice expires.**

# Hardware Pulses And Clock Ticks

- Real-time clock hardware
  - May have fixed *pulse rate*
  - Can run off processor clock
  - Rate may not divide a second by power of ten
- Software
  - Written to use *abstract tick rate*, typically a power of 10
  - Can differ from pulse rate
- Clock interrupt handler matches the two rates



# Managing Real-time Clock Events

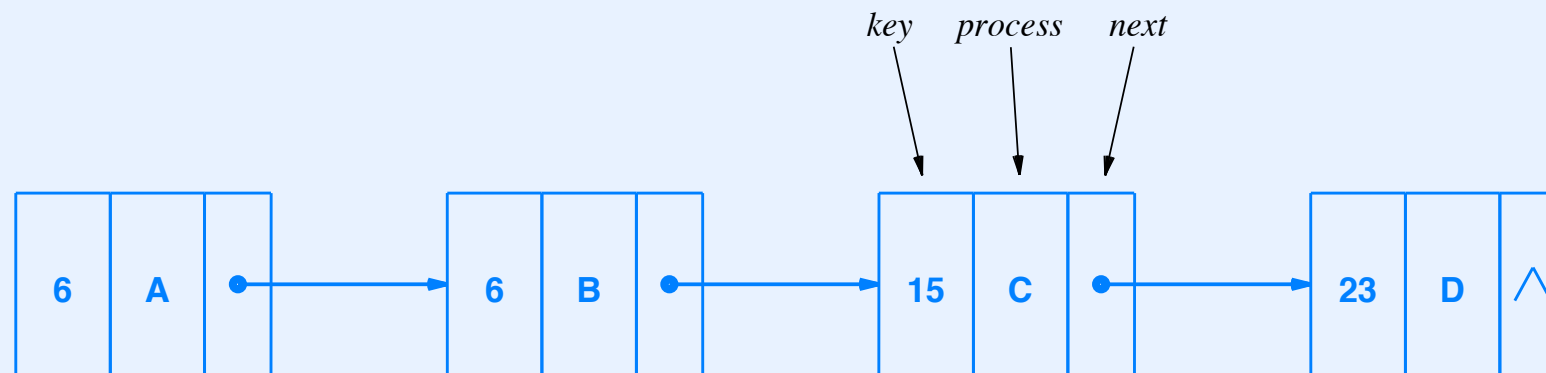
- Must be efficient
  - Clock interrupts occur frequently and continuously
  - Set of timed events examined on each clock interrupt
  - Should avoid searching a list
- Mechanism
  - Keep timed events on a linked list
  - One item appears on the list for each outstanding event
  - Called *event queue*

# Delta List

- Data structure used for timed events
- Items on the list are ordered by time of occurrence
- For efficiency, list stores *relative* times
- The key in an item stores the difference (*delta*) between the time for the event and time for the previous event
- The key in first event stores the delta from “now”

# Delta List Example

- Assume events for processes *A* through *D* will occur 6, 12, 27, and 50 ticks from now
- The delta keys are 6, 6, 15, and 23



# Real-time Clock Processing In Xinu

- Clock interrupt handler
  - Decrements preemption counter
  - Reschedules if timeslice has expired
  - Processes the event list
- Sleep queue
  - Delta list of delayed processes
  - Each node on the list corresponds to a sleeping process
- Global variable *sleepq* contains the ID of the sleep queue

# Keys On The Xinu Sleep Queue

- Processes on *sleepq* are ordered by time at which they will awaken
- Each key tells the number of clock ticks that the process must delay beyond the preceding one on the list
- Relationship must be maintained whenever an item is inserted or deleted

# Real-time Delay And Clock Resolution

- Process calls *sleep* to delay
- Question: what resolution should be used for sleep?
  - Humans typically think in seconds or minutes
  - Real-time applications may need millisecond accuracy or higher (if available)

# Xinu Sleep Primitives And Resolution

- A set of functions accommodates a range of possible resolutions

sleep        – delay given in seconds

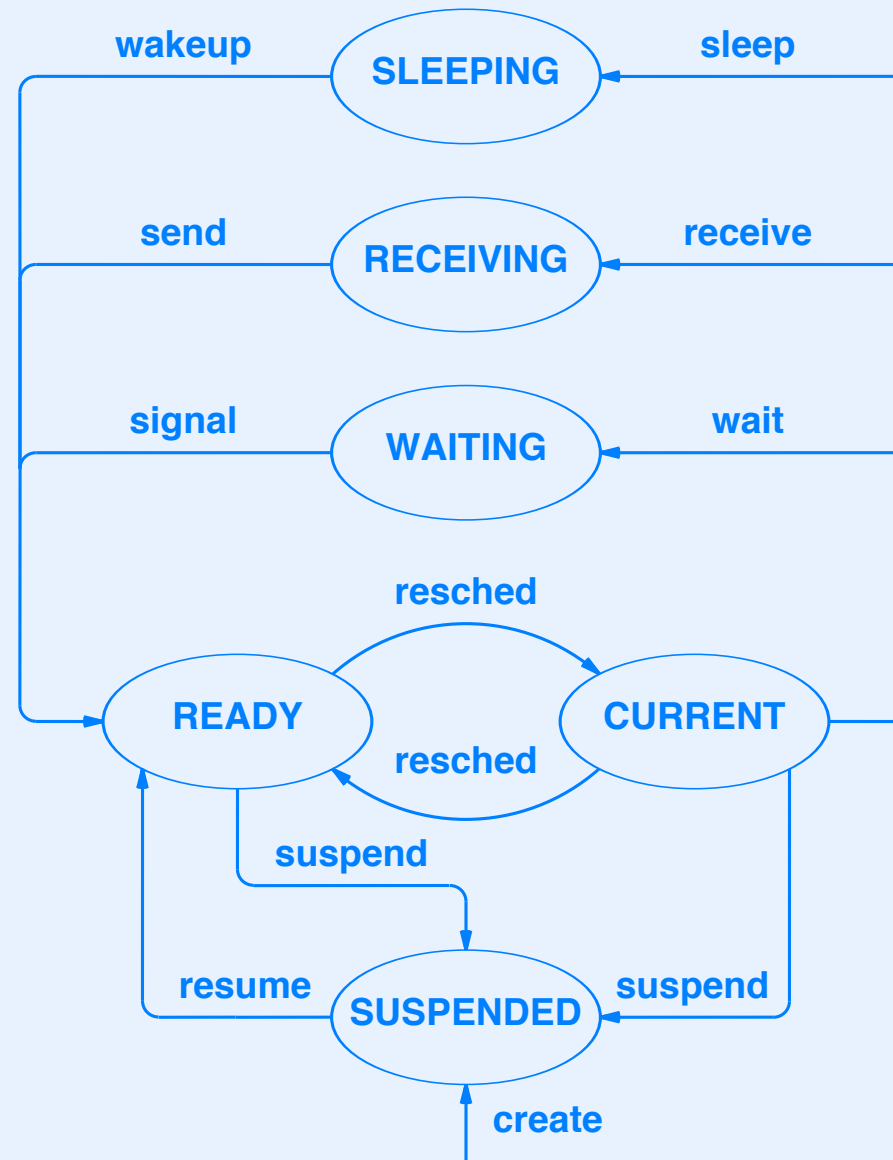
sleep10     – delay given in tenths of seconds

sleep100    – delay given in hundredths of seconds

sleepms     – delay given in milliseconds

- Note: some hardware does not support all resolutions

# New Process State For Sleeping Processes





# Xinu Sleep Function (part 1)

```
/* sleep.c - sleep sleeps */

#include <xinu.h>

#define MAXSECONDS      2147483          /* Max seconds per 32-bit msec */

/*-----
 * sleep - Delay the calling process n seconds
 *-----
 */
syscall sleep(
    int32 delay          /* Time to delay in seconds */
)
{
    if ( (delay < 0) || (delay > MAXSECONDS) ) {
        return SYSERR;
    }
    sleeps(1000*delay);
    return OK;
}
```

## Xinu Sleep Function (part 2)

```
/*-----  
 *  sleepms  -  Delay the calling process n milliseconds  
 *-----  
 */  
syscall sleepms(  
    int32 delay                /* Time to delay in msec.          */  
)  
{  
    intmask mask;              /* Saved interrupt mask          */  
  
    if (delay < 0) {  
        return SYSERR;  
    }  
  
    if (delay == 0) {  
        yield();  
        return OK;  
    }  
}
```

## Xinu Sleep Function (part 3)

```
/* Delay calling process */

mask = disable();
if (insertd(currpid, sleepq, delay) == SYSERR) {
    restore(mask);
    return SYSERR;
}

proctab[currpid].prstate = PR_SLEEP;
resched();
restore(mask);
return OK;
}
```

# Inserting An Item On Sleepq

- Current process calls *sleepms* or *sleep* to request delay
- *Sleepms*
  - Underlying function that takes action
  - Inserts current process on *sleepq*
  - Calls *resched* to allow other processes to execute
- Method
  - Walk through *sleepq* (with interrupts disabled)
  - Find place to insert the process
  - Adjust remaining keys as necessary

# Xinu Insertd (part 1)

```
/* insertd.c - insertd */

#include <xinu.h>

/*-----
 * insertd - Insert a process in delta list using delay as the key
 *-----
 */
status insertd(                /* Assumes interrupts disabled */
    pid32      pid,            /* ID of process to insert */
    qid16      q,              /* ID of queue to use */
    int32      key             /* Delay from "now" (in ms.) */
)
{
    int32      next;            /* Runs through the delta list */
    int32      prev;            /* Follows next through the list*/

    if (isbadqid(q) || isbadpid(pid)) {
        return SYSERR;
    }
}
```

## Xinu Insertd (part 2)

```
prev = queuehead(q);
next = queuestab[queuehead(q)].qnext;
while ((next != queuestab[q]) && (queuestab[next].qkey <= key)) {
    key -= queuestab[next].qkey;
    prev = next;
    next = queuestab[next].qnext;
}

/* Insert new node between prev and next nodes */

queuestab[pid].qnext = next;
queuestab[pid].qprev = prev;
queuestab[pid].qkey = key;
queuestab[prev].qnext = pid;
queuestab[next].qprev = pid;
if (next != queuestab[q]) {
    queuestab[next].qkey -= key;
}

return OK;
}
```

# Invariant During Sleepq Insertion

**At any time during the search, both `key` and `queuetab[next].qkey` specify a delay relative to the time at which the predecessor of the “next” process awakens.**

# Testing A Clock

- On some systems, clock hardware is optional
- If optional, OS can test for presence of a clock
  - Initialize clock interrupt vector
  - Enable interrupts
  - Loop “long enough”
  - If interrupt occurs, declare clock present
  - Otherwise, declare no clock present and disable *sleep*



# Clock Interrupt Handler

- May need to be optimized
- Handles preemption
  - Decrements the preemption counter
  - Calls *resched* if the counter reaches zero
- Handles sleeping processes
  - Decrements the key of the first process on the sleep queue
  - Calls *wakeup* if the counter reaches zero

# Clock Interrupt Handler

## (continued)

- Process priority and sleep
  - More than one process may awaken at a given time
  - Processes that awaken at the same time may not be in order by priority
  - If a process starts running immediately, a process of higher priority may be blocked, even if its sleep time has expired
- Consequence: *wakeup* should awaken *all* processes that have zero time remaining before allowing any of them to run

# Example Clock Interrupt Handler (part 1)

```
/* clkhandler.c - clkhandler */

#include <xinu.h>

/*-----
 * clkhandler - high level clock interrupt handler
 *-----
 */
void clkhandler()
{
    static uint32 count1000 = 1000;          /* Count to 1000 ms */

    /* Decrement the ms counter, and see if a second has passed */
    if((--count1000) <= 0) {
        /* One second has passed, so increment seconds count */
        clktime++;

        /* Reset the local ms counter for the next second */
        count1000 = 1000;
    }
}
```

## Example Clock Interrupt Handler (part 2)

```
/* Handle sleeping processes if any exist */

if(!isempty(sleepq)) {

    /* Decrement the delay for the first process on the      */
    /*   sleep queue, and awaken if the count reaches zero    */

    if((--queuetab[firstid(sleepq)].qkey) <= 0) {
        wakeup();
    }
}

/* Decrement the preemption counter, and reschedule when the */
/*   remaining time reaches zero                               */

if((--preempt) <= 0) {
    preempt = QUANTUM;
    resched();
}
}
```

# Clock Initialization (Part 1)

```
/* clkinit.c - clkinit (x86) */

#include <xinu.h>

uint32  clktime;           /* Seconds since boot          */
uint32  ctr1000 = 0;       /* Milliseconds since boot     */
qid16   sleepq;           /* Queue of sleeping processes */
uint32  preempt;          /* Preemption counter          */

/*-----
 * clkinit - Initialize the clock and sleep queue at startup (x86)
 *-----
 */
void    clkinit(void)
{
    uint16  intv;           /* Clock rate in KHz          */

    /* Allocate a queue to hold the delta list of sleeping processes*/

    sleepq = newqueue();

    /* Initialize the preemption count */

    preempt = QUANTUM;
```

## Clock Initialization (Part 2)

```
/* Initialize the time since boot to zero */

clktime = 0;

/* Set interrupt vector for the clock to invoke clkdisp */

set_evec(IRQBASE, (uint32)clkdisp);

/* Set the hardware clock: timer 0, 16-bit counter, rate */
/*   generator mode, and counter runs in binary           */

outb(CLKCNTRL, 0x34);

/* Set the clock rate to 1.190 Mhz; this is 1 ms interrupt rate */

intv = 1193;    /* Using 1193 instead of 1190 to fix clock skew */

/* Must write LSB first, then MSB */

outb(CLOCK0, (char) (0xff & intv) );
outb(CLOCK0, (char) (0xff & (intv>>8)));

return;

}
```

# Summary

- Computer can contain several types of hardware clocks
  - Processor
  - Time of day
  - Real-time
  - Interval timer
- Real-time clock or interval timer used for
  - Preemption
  - Process delay
- OS may need to convert hardware pulse rate to appropriate tick rate

## Summary (continued)

- Delta list provides elegant data structure to store a set of sleeping processes
- Only the key of the first item on the list needs to be updated on each clock tick
- Multiple processes may awaken at the same time; rescheduling is deferred until all have been made ready
- *Recvtime* allows a process to wait a specified time for a message to arrive





**Questions?**