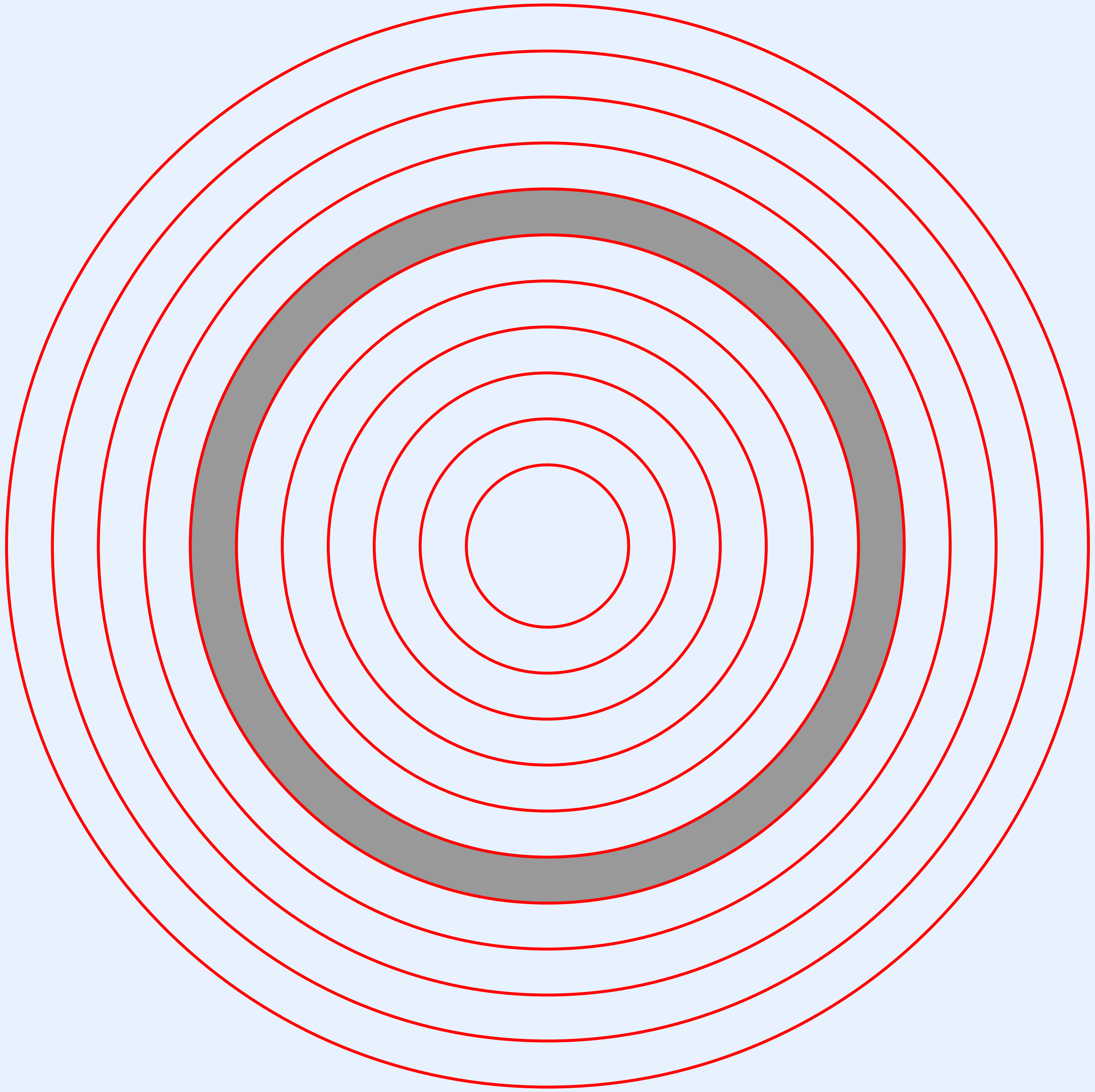
Module VIII

Device Management
Interrupts And Device Drivers

Copyright © 2019 by Douglas Comer and CRC Press. All rights reserved.

Location Of Device Management In The Hierarchy



Copyright © 2019 by Douglas Comer and CRC Press. All rights reserved.

Ancient History

- Each device had a unique hardware interface
- Code to communicate with device was built into applications
- An application polled the device; interrupts were not used
- Disadvantages
  - It was painful to create a program
  - A program could not use arbitrary devices (e.g., specific models of a printer and a disk were part of the program)

Copyright © 2019 by Douglas Comer and CRC Press. All rights reserved.

The Modern Approach

- A device manager is part of an operating system
- The operating system presents applications with a uniform interface to all devices (as much as possible)
- All I / O is *interrupt-driven*

Copyright © 2019 by Douglas Comer and CRC Press. All rights reserved.

A Device Manager In An Operating System

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

Copyright © 2019 by Douglas Comer and CRC Press. All rights reserved.

A Conceptual Note

One of the most intellectually difficult aspects of operating systems arises from the interaction between processes (an operating system 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.

Copyright © 2019 by Douglas Comer and CRC Press. All rights reserved.

Review Of I/O Using Interrupts

- The processor
  - Starts a device
  - Enables interrupts and continues with other computation
- The device
  - Performs the requested operation
  - Raises an interrupt on the bus
- Processor hardware
  - Checks for interrupts after each instruction is executed, and invokes an interrupt function if an interrupt is pending
  - Has a special instruction used to return from interrupt mode and resume normal processing

Copyright © 2019 by Douglas Comer and CRC Press. All rights reserved.

Processes And Interrupts

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

The current process executes interrupt code

Copyright © 2019 by Douglas Comer and CRC Press. All rights reserved.

Historic Interrupt Software

- A separate interrupt function was created for each device
  - Very low-level code
  - Handles many details
    - Saves / restores registers
    - Sets the interrupt mask
  - Finds the interrupting device on the bus
  - Interacts with the device to transfer data
  - Resets the device for the next interrupt
  - Returns from the interrupt to normal processing

Copyright © 2019 by Douglas Comer and CRC Press. All rights reserved.

Modern Interrupt Software (Two Pieces)

- An *interrupt dispatcher*
  - Is a single function common to all interrupts
  - It handles low-level details, such as finding the interrupting device on the bus
  - It sets up the environment needed for a function call and calls a device-specific function
  - Some functionality may be incorporated into an *interrupt controller chip*
- An *interrupt handler*
  - One handler for each device
  - Is invoked by the dispatcher
  - Performs all interaction with a specific device

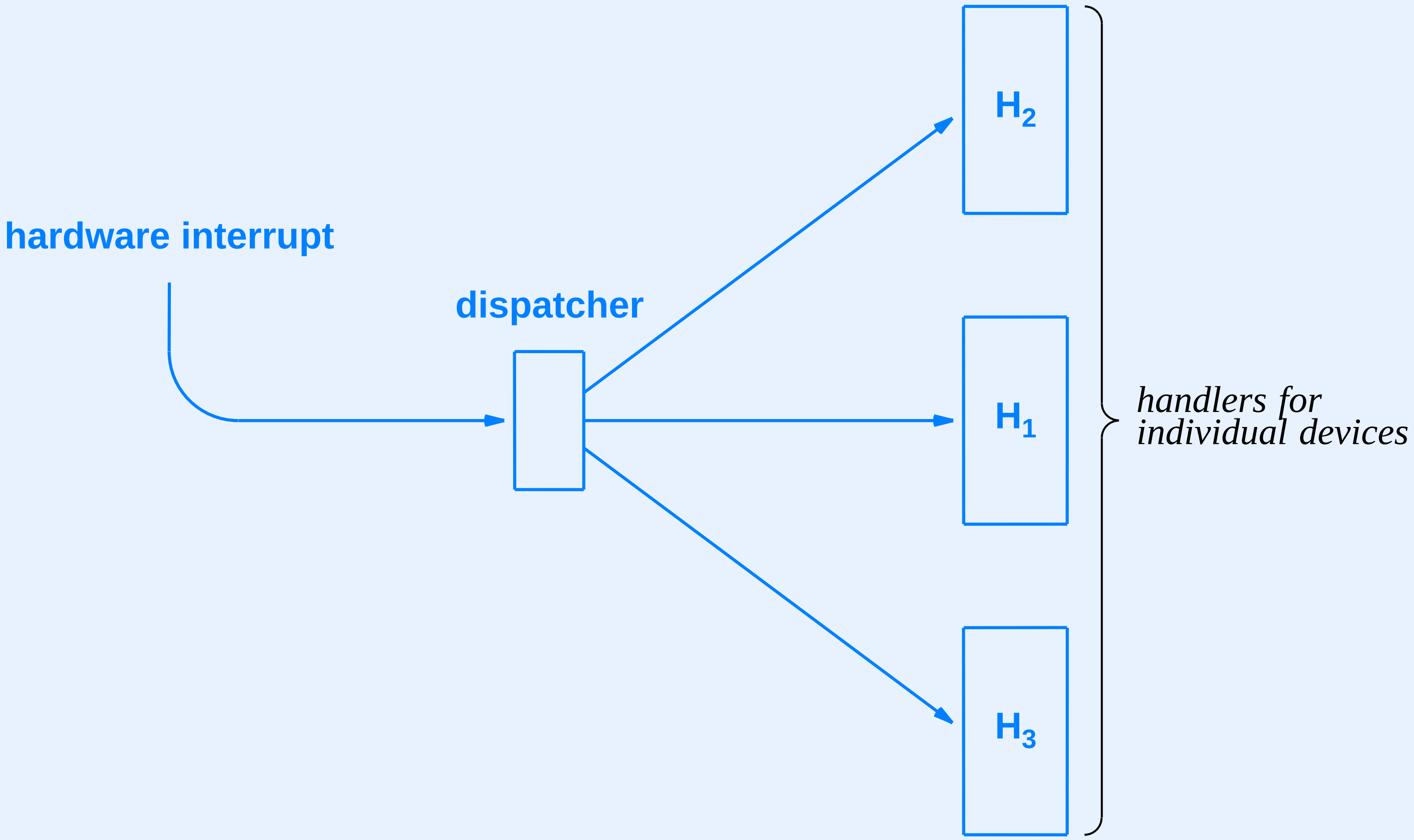
Copyright © 2019 by Douglas Comer and CRC Press. All rights reserved.

Interrupt Dispatcher

- A low-level piece of code
- Is invoked by the hardware when interrupt occurs
  - Runs in interrupt mode (i.e., with further interrupts disabled)
  - The hardware has saved the instruction pointer for a return
- The dispatcher
  - Saves other machine state as necessary
  - Identifies the interrupting device
  - Establishes the high-level runtime environment needed by a C function
  - Calls a device-specific *interrupt handler*

Copyright © 2019 by Douglas Comer and CRC Press. All rights reserved.

Conceptual View Of Interrupt Dispatching



- Note: the dispatcher is typically written in assembly language

Copyright © 2019 by Douglas Comer and CRC Press. All rights reserved.

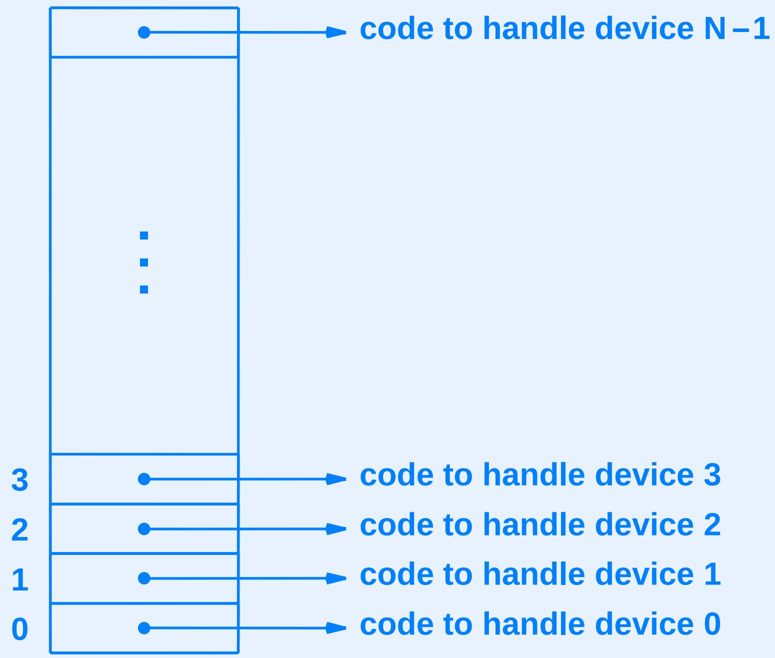
Return From Interrupt

- The interrupt handler
  - Communicates with the device
  - May restart the next operation on the device
  - Eventually returns to the interrupt dispatcher
- The interrupt dispatcher
  - Executes a special hardware instruction known as *return from interrupt*
- The *return from interrupt* instruction atomically
  - Resets the instruction pointer to the saved value
  - Enables interrupts

Copyright © 2019 by Douglas Comer and CRC Press. All rights reserved.

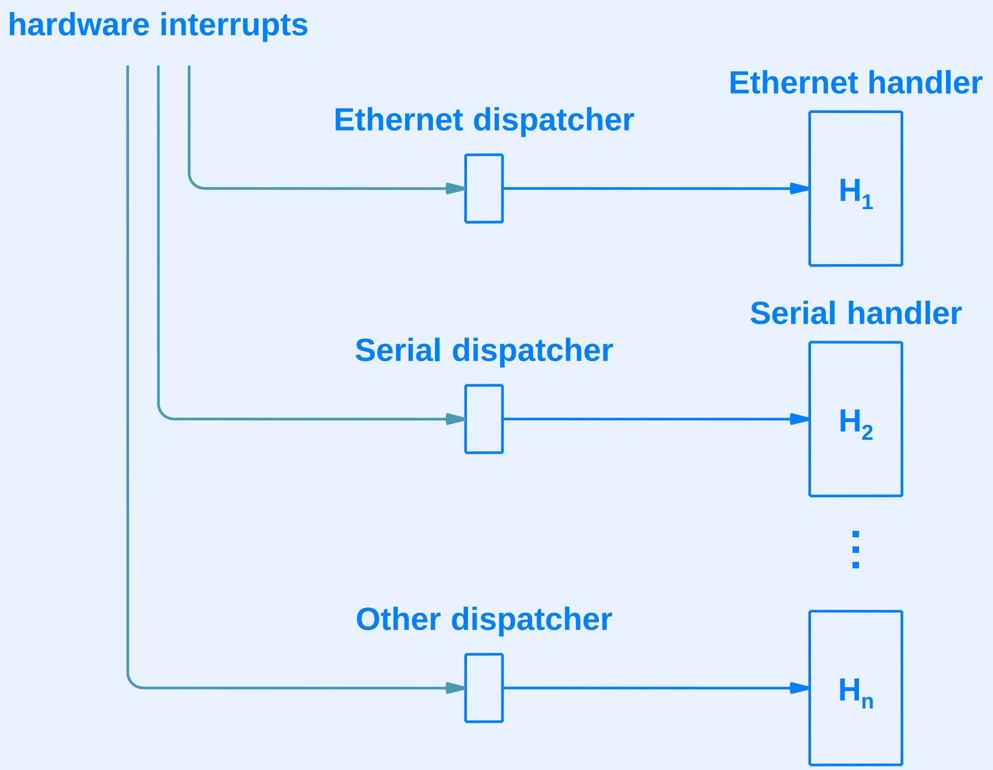
The Mechanism Used For Interrupts: A Vector

- Each possible interrupt is assigned a unique integer, sometimes called an *IRQ*
- The hardware uses the IRQ as an index into an *interrupt vector* array
- Conceptual organization of an interrupt vector



Copyright © 2019 by Douglas Comer and CRC Press. All rights reserved.

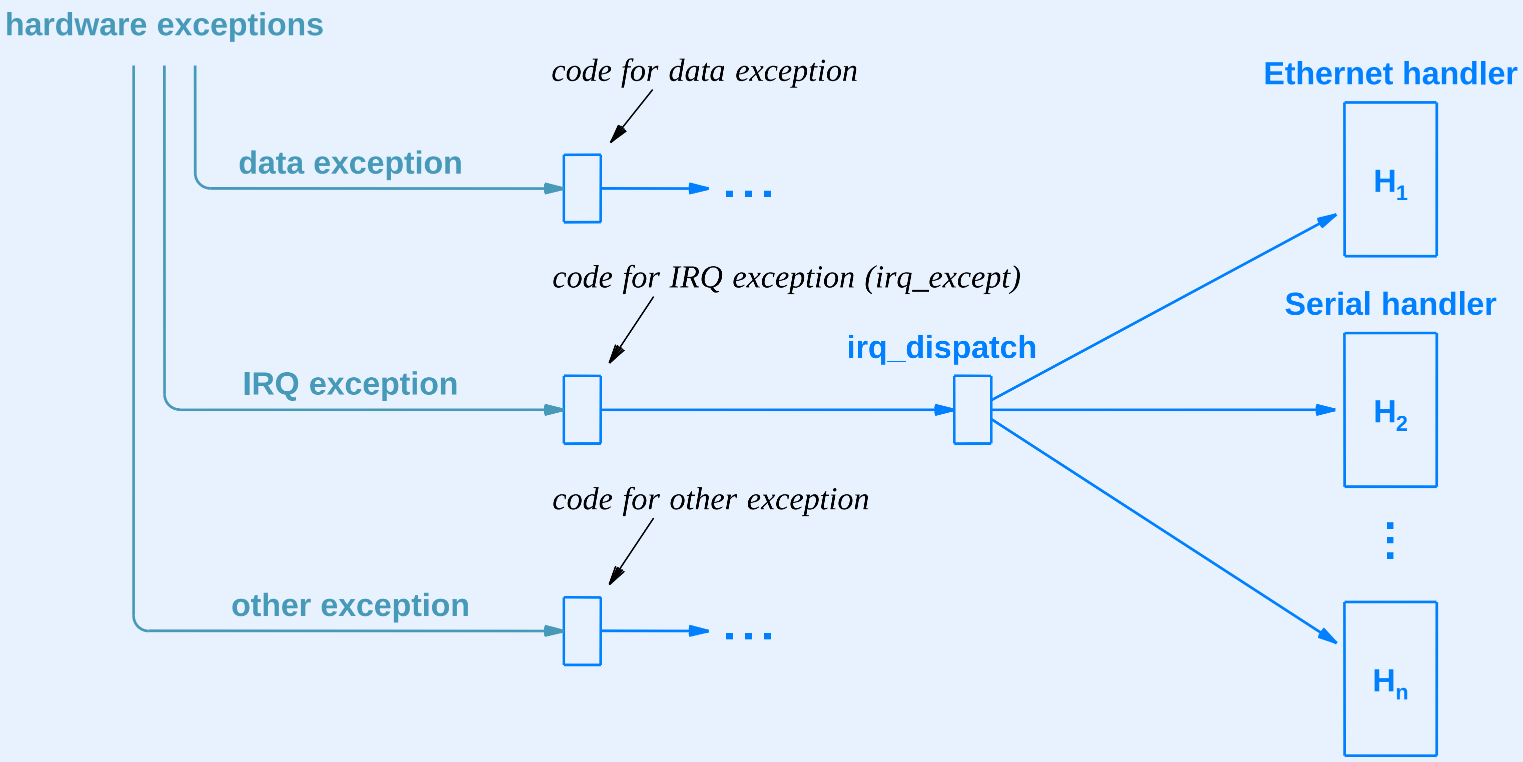
Interrupts On A Galileo (x86)



- The operating system preloads the interrupt controller with the address of a dispatcher for each device
- The controller invokes the correct dispatcher

Copyright © 2019 by Douglas Comer and CRC Press. All rights reserved.

Interrupts On A BeagleBone Black (ARM)



- Uses a two-level scheme where the controller hardware raises an *IRQ exception* for any device interrupt

- The IRQ exception code invokes the IRQ dispatcher, which calls the correct handler

Copyright © 2019 by Douglas Comer and CRC Press. All rights reserved.

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 of the rule later

Copyright © 2019 by Douglas Comer and CRC Press. All rights reserved.

Interrupts And Processes

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

Copyright © 2019 by Douglas Comer and CRC Press. All rights reserved.

The Scheduling Invariant

- Suppose process *X* is executing when an interrupt occurs
- We said that 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, process *Y*, is waiting for the data
- If the hander merely returns from the interrupt, process *X* will continue to execute
- To maintain the scheduling invariant, the handler must call *resched*

Copyright © 2019 by Douglas Comer and CRC Press. All rights reserved.

Interrupts And The Null Process

- In the concurrent processing world
  - A process is always running
  - An interrupt can occur at any time
  - The currently executing process executes interrupt code
- An important consequence: the null process may be running when an interrupt occurs, which means the null process will execute the interrupt handler
- We know that the null process must always remain eligible to execute

Copyright © 2019 by Douglas Comer and CRC Press. All rights reserved.

A Restriction On Interrupt Handlers Imposed By The Null Process

Because an interrupt can occur while the null process is executing, an interrupt hander 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**.**

Copyright © 2019 by Douglas Comer and CRC Press. All rights reserved.

A Question About Scheduling And Interrupts

- Recall that
  - The hardware disables further interrupts before invoking a dispatcher
  - Interrupts remain disabled when the dispatcher calls a 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
- What happens if an interrupt calls a function that calls *resched* and the new process has interrupts enabled?

Copyright © 2019 by Douglas Comer and CRC Press. All rights reserved.

An Example Of Rescheduling During Interrupt Processing

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

Copyright © 2019 by Douglas Comer and CRC Press. All rights reserved.

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

Copyright © 2019 by Douglas Comer and CRC Press. All rights reserved.

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**).**

Copyright © 2019 by Douglas Comer and CRC Press. All rights reserved.

Device Drivers

Definition Of A Device Driver

- A *device driver* consists of a set of functions that perform I/O operations on a given device
- The code is device-specific
- The set includes
  - An interrupt handler function
  - Functions to control the device
  - Functions to read and write data
- The code is divided into two conceptual parts

Copyright © 2019 by Douglas Comer and CRC Press. All rights reserved.

The Two Conceptual Parts Of A Device Driver

- The upper-half
  - Functions that are executed by an application
  - The functions usually perform data transfer (*read* or *write*)
  - The code copies data between the user and kernel address spaces
- The lower-half
  - Is invoked by the hardware when an interrupt occurs
  - Consists of a device-specific interrupt handler
  - May also include dispatcher code, depending on the architecture
  - Executed by whatever process is executing
  - May restart the device for the next operation

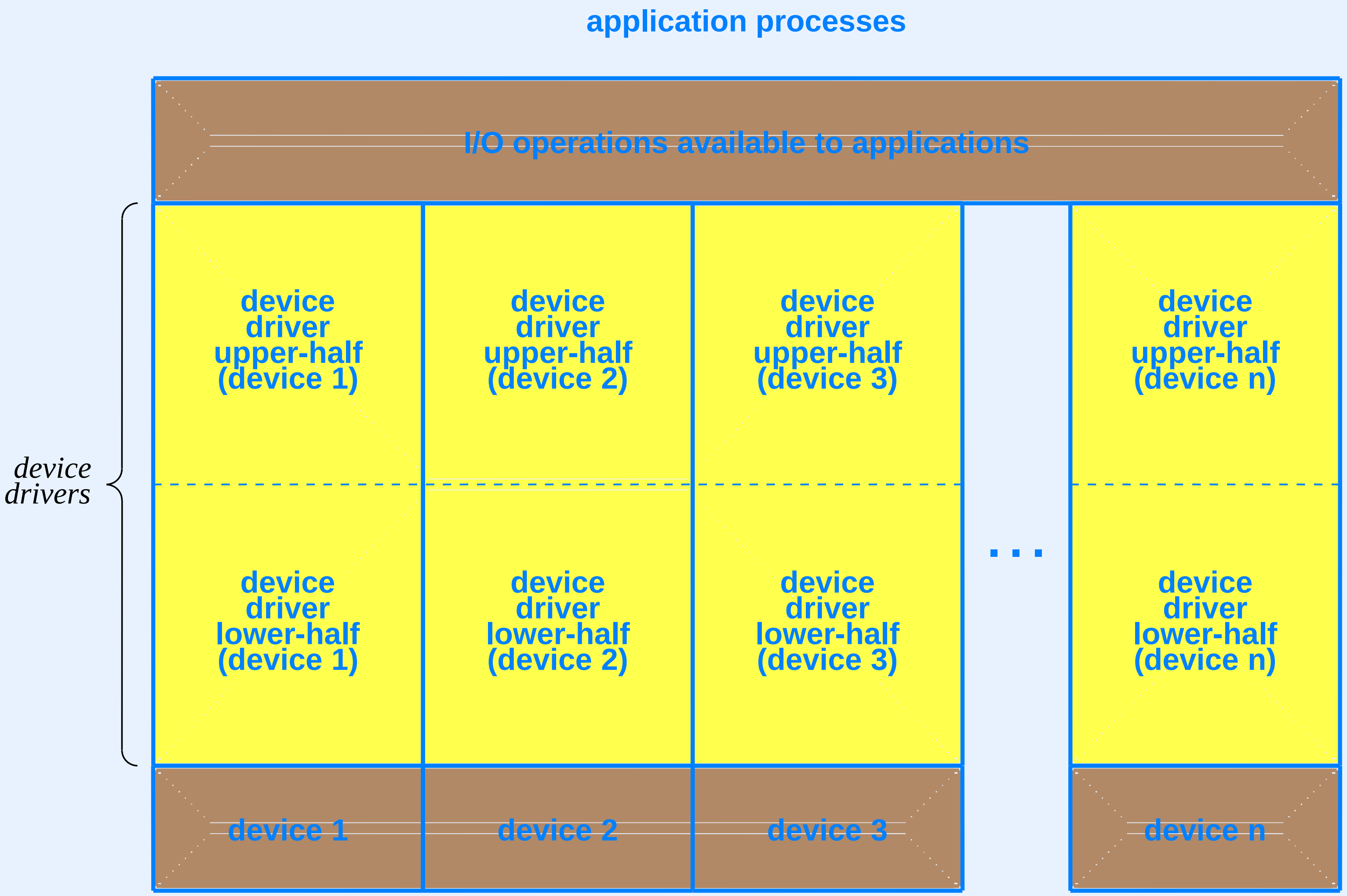
Copyright © 2019 by Douglas Comer and CRC Press. All rights reserved.

Division Of Duties In A Driver

- The upper-half functions
  - Have minimal interaction with device hardware
  - Enqueue a request, and may start the device
- The lower-half functions
  - Have minimal interaction with application
  - Interact with the device to
    - Obtain incoming data
    - Start output
  - Reschedule if a process is waiting for the device

Copyright © 2019 by Douglas Comer and CRC Press. All rights reserved.

Conceptual Organization Of Device Software


application processes
I/O operations available to applications
device drivers
device driver upper-half (device 1)
device driver upper-half (device 2)
device driver upper-half (device 3)
device driver upper-half (device n)
device driver lower-half (device 1)
device driver lower-half (device 2)
device driver lower-half (device 3)
. . .
device driver lower-half (device n)
device 1
device 2
device 3
device n

Copyright © 2019 by Douglas Comer and CRC Press. All rights reserved.

Synchronous Interface I/O

- Most systems provide a *synchronous* I/O interface to applications
- For input, the calling process is blocked until data arrives
- For output, the calling process is blocked until the device driver has buffer space to store the outgoing data

Copyright © 2019 by Douglas Comer and CRC Press. All rights reserved.

Coordination Of Processes Performing I/O

- A device driver must be able to block and later unblock application processes
- Good news: there is no need to invent new coordination mechanisms because standard process coordination mechanisms suffice
  - Message passing
  - Semaphores
  - Suspend / resume
- We will see examples later

Copyright © 2019 by Douglas Comer and CRC Press. All rights reserved.

Summary Of Interrupts And Device Drivers

- The *device manager* in an operating system handles I/O
- Device-independent routines
  - Provide uniform interface
  - Define generic operations that must be mapped to device-specific functions
- Interrupt code
  - Consists of single dispatcher and handler for each device
  - Is executed by whatever process was running when interrupt occurred
- To accommodate null process, interrupt handler must leave executing process in *current* or *ready* states

Copyright © 2019 by Douglas Comer and CRC Press. All rights reserved.

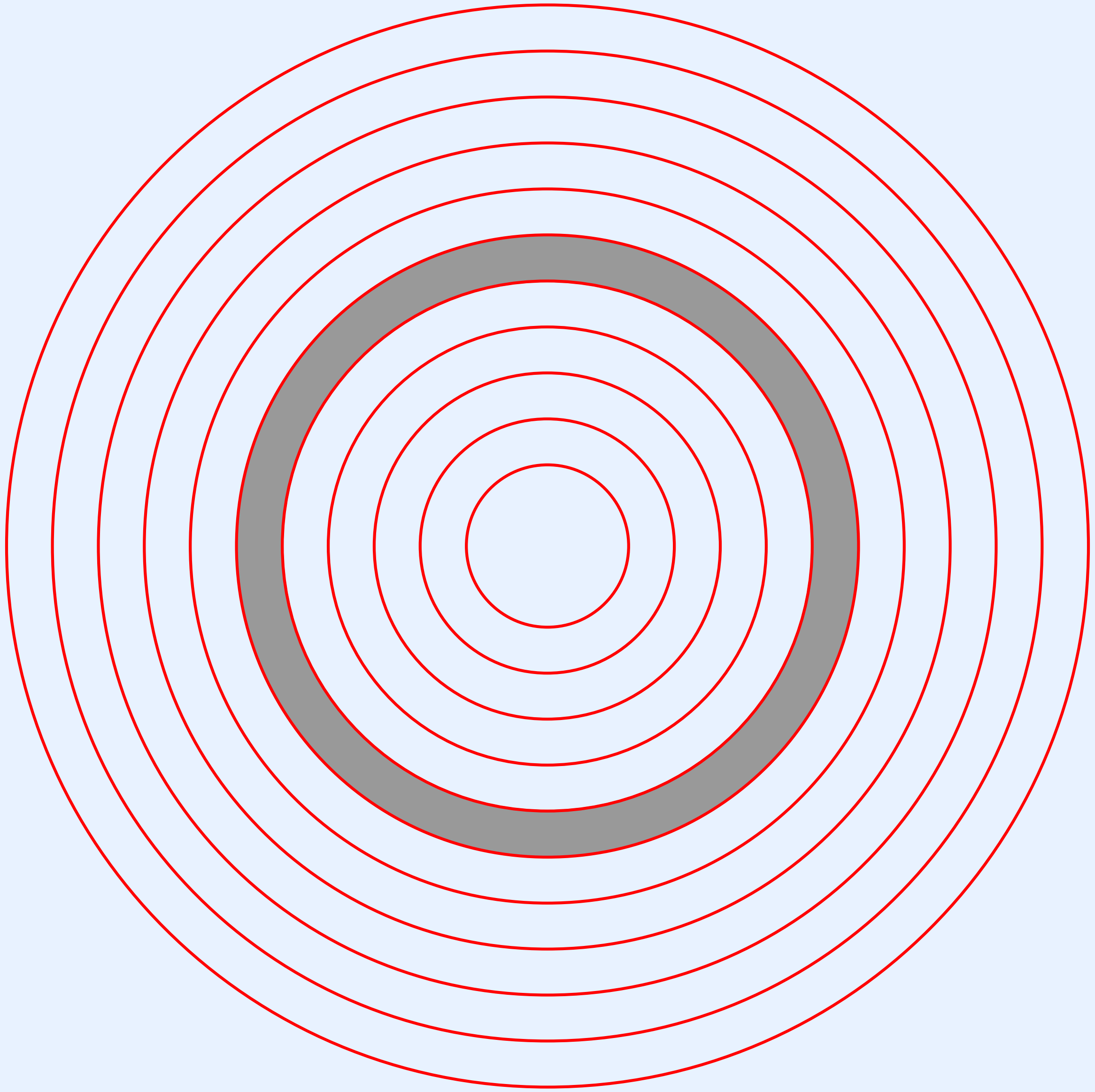
Summary

- Rescheduling during interrupt is safe provided
  - Global data structures valid
  - No process explicitly enables interrupts
- Device driver functions
  - Are divided into upper-half and lower-half
  - Can use existing primitives to block and unblock processes

Copyright © 2019 by Douglas Comer and CRC Press. All rights reserved.

Clocks And Clock Management

Location Of Clock Management In The Hierarchy



Copyright © 2019 by Douglas Comer and CRC Press. All rights reserved.

Various Types Of Clock Hardware Exist

- Processor clock (rate at which instructions execute)
- Real-time clock
  - Pulses regularly
  - Interrupts the processor on each pulse
  - Called *programmable* if rate can be controlled by OS
- Interval timer
  - The processor sets a timeout and the device interrupts after the specified time
  - Can be used to pulse regularly
  - May have an automatic restart mechanism

Copyright © 2019 by Douglas Comer and CRC Press. All rights reserved.

Timed Events

- Two types of timed events are important to an operating system
- A *preemption event*
  - Known as *timeslicing*
  - Guarantees that a given process cannot run forever
  - Switches the processor to another process
- A *sleep event*
  - Is requested by a process to delay for a specified time
  - The process resumes execution after the time passes

Copyright © 2019 by Douglas Comer and CRC Press. All rights reserved.

A Note About Timeslicing

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.

Copyright © 2019 by Douglas Comer and CRC Press. All rights reserved.

Managing Timed Events

- The code must be efficient because
  - Clock interrupts occur frequently and continuously
  - More than one event may occur at a given time
  - The clock interrupt code should avoid searching a list
- An efficient mechanism
  - All timed events are kept on a list
  - The list is known as an *event queue*

Copyright © 2019 by Douglas Comer and CRC Press. All rights reserved.

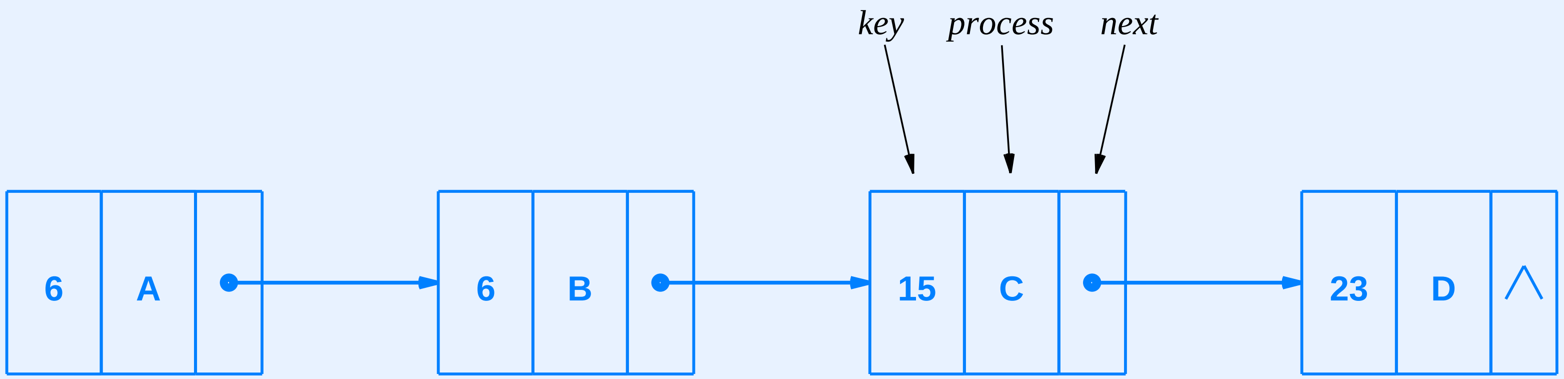
The Delta List

- A data structure used for timed events
- Items on a delta list are ordered by the time they will occur
- Trick to make processing efficient: use *relative* times
- Implementation: 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”

Copyright © 2019 by Douglas Comer and CRC Press. All rights reserved.

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*



Copyright © 2019 by Douglas Comer and CRC Press. All rights reserved.

Real-time Clock Processing In Xinu

- The clock interrupt handler
  - Decrements the preemption counter and calls *resched* if the timeslice has expired
  - Processes the sleep queue
- The sleep queue
  - Is a delta list
  - Each item on the list is a sleeping process
- Global variable *sleepq* contains the ID of the sleep queue

Copyright © 2019 by Douglas Comer and CRC Press. All rights reserved.

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
- The relationship must be maintained whenever an item is inserted or deleted

Copyright © 2019 by Douglas Comer and CRC Press. All rights reserved.

Sleep Timer Resolution

- A process calls *sleep* to delay
- Question: what resolution should be used for sleep?
  - Humans typically think in seconds or minutes
  - Some applications may need millisecond accuracy (or more, if available)
- The tradeoff: using a high resolution, such as microseconds, means long delays will overflow a 32-bit integer

Copyright © 2019 by Douglas Comer and CRC Press. All rights reserved.

Xinu Sleep Primitives

- Xinu offers a set of functions to accommodate a range of possible resolutions

  sleep – the delay is given in seconds

  sleep10 – the delay is given in tenths of seconds

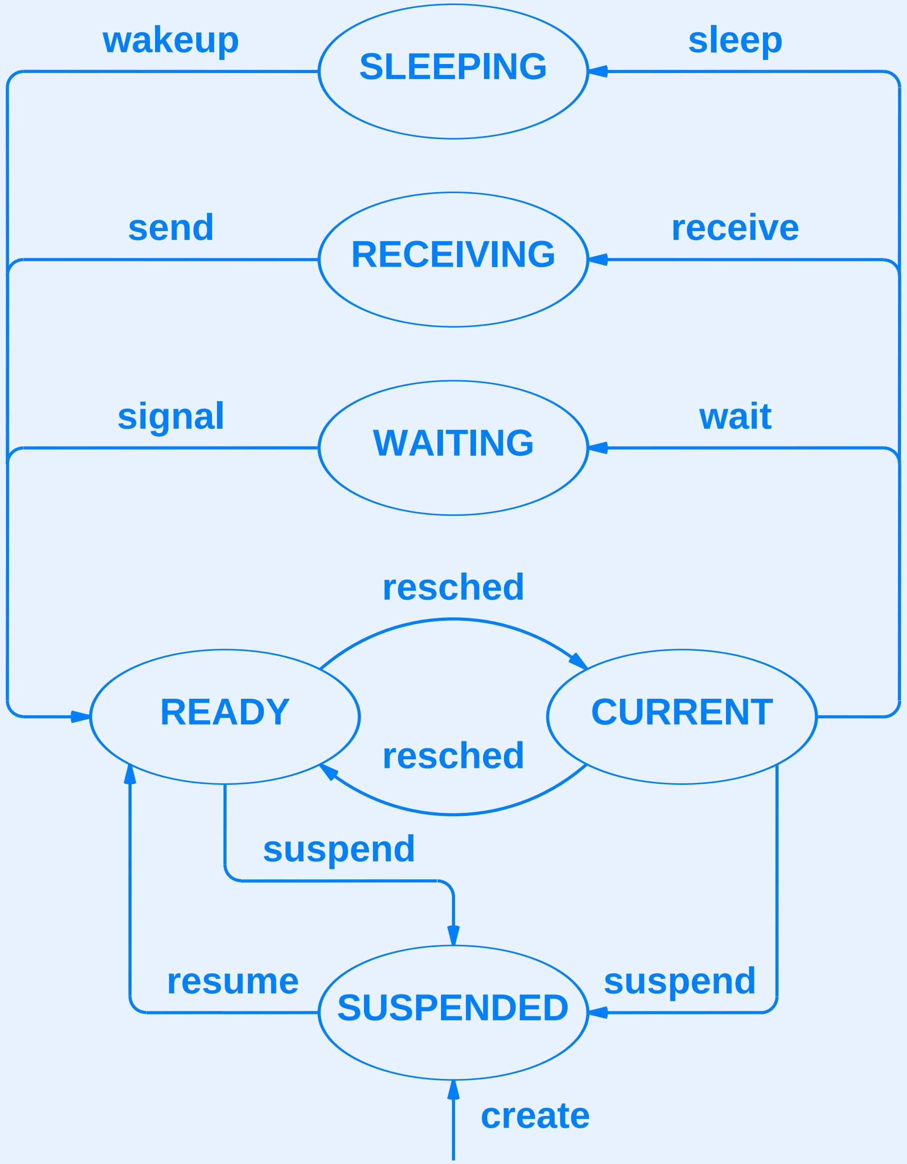
  sleep100 – the delay is given in hundredths of seconds

  sleepms – the delay is given in milliseconds

- The smallest resolution is milliseconds because the clock operates at a rate of one millisecond per tick

Copyright © 2019 by Douglas Comer and CRC Press. All rights reserved.

A New Process State For Sleeping Processes



Copyright © 2019 by Douglas Comer and CRC Press. All rights reserved.

Xinu Sleep Function (Part 1)

```
/* sleep.c - sleep sleepms */

#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;
        }
        sleepms(1000*delay);
        return OK;
}
```

Copyright © 2019 by Douglas Comer and CRC Press. All rights reserved.

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;
        }
```

Copyright © 2019 by Douglas Comer and CRC Press. All rights reserved.

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;
}
```

Copyright © 2019 by Douglas Comer and CRC Press. All rights reserved.

Inserting An Item On Sleepq

- The current process calls *sleepms* or *sleep* to request a delay
- *Sleepms*
  - The 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 the place to insert the process
  - Adjust remaining keys as necessary

Copyright © 2019 by Douglas Comer and CRC Press. All rights reserved.

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;
        }
```

Copyright © 2019 by Douglas Comer and CRC Press. All rights reserved.

Xinu Insertd (Part 2)

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

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

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

        return OK;
}
```

Copyright © 2019 by Douglas Comer and CRC Press. All rights reserved.

The Invariant Used 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.**

Copyright © 2019 by Douglas Comer and CRC Press. All rights reserved.

A Clock Interrupt Handler

- Updates the time-of-day (which counts seconds)
- Handles sleeping processes
  - Decrements the key of the first process on the sleep queue
  - Calls *wakeup* if the counter reaches zero
- Handles preemption
  - Decrements the preemption counter
  - Calls *resched* if the counter reaches zero

Copyright © 2019 by Douglas Comer and CRC Press. All rights reserved.

A Clock Interrupt Handler
(continued)

- When sleeping processes awaken
  - More than one process may awaken at a given time
  - The processes may not have the same priority
  - If the clock interrupt handler starts a process running, a higher priority process may remain on the sleep queue, even if its time has expired
- Solution: *wakeup* awakens *all* processes that have zero time remaining before allowing any of them to run

Copyright © 2019 by Douglas Comer and CRC Press. All rights reserved.

Xinu Wakeup

```
/* wakeup.c - wakeup */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  wakeup  -  Called by clock interrupt handler to awaken processes
 *------------------------------------------------------------------------
 */
void    wakeup(void)
{
        /* Awaken all processes that have no more time to sleep */

        resched_cntl(DEFER_START);
        while (nonempty(sleepq) && (firstkey(sleepq) <= 0)) {
                ready(dequeue(sleepq));
        }

        resched_cntl(DEFER_STOP);
        return;
}
```

- Note that rescheduling is deferred until all processes are awakened

Copyright © 2019 by Douglas Comer and CRC Press. All rights reserved.

Timed Message Reception

- Many operating system components offer a “timeout” on operations
- Timeout is especially useful in building communication protocols
- A Xinu example: receive with timeout
  - Operates like *receive*, but includes a timeout argument
  - If a message arrives before the timer expires, the message is returned
  - If the timer expires before a message arrives, the value *TIMEOUT* is returned
  - Implemented with *recvtime*
- Recvtime uses the same queue and wakeup mechanism as sleeping processes

Copyright © 2019 by Douglas Comer and CRC Press. All rights reserved.

Xinu Recvtime (Part 1)

```
/* recvtime.c - recvtime */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  recvtime  -  Wait specified time to receive a message and return
 *------------------------------------------------------------------------
 */
umsg32  recvtime(
          int32         maxwait         /* Ticks to wait before timeout */
        )
{
        intmask mask;                   /* Saved interrupt mask         */
        struct  procent *prptr;         /* Tbl entry of current process */
        umsg32  msg;                    /* Message to return            */

        if (maxwait < 0) {
                return SYSERR;
        }
        mask = disable();
```

Copyright © 2019 by Douglas Comer and CRC Press. All rights reserved.

Xinu Recvtime (Part 2)

```
        /* Schedule wakeup and place process in timed-receive state */

        prptr = &proctab[currpid];
        if (prptr->prhasmsg == FALSE) { /* Delay if no message waiting  */
                if (insertd(currpid,sleepq,maxwait) == SYSERR) {
                        restore(mask);
                        return SYSERR;
                }
                prptr->prstate = PR_RECTIM;
                resched();
        }

        /* Either message arrived or timer expired */

        if (prptr->prhasmsg) {
                msg = prptr->prmsg;     /* Retrieve message             */
                prptr->prhasmsg = FALSE;/* Reset message indicator      */
        } else {
                msg = TIMEOUT;
        }
        restore(mask);
        return msg;
}
```

Copyright © 2019 by Douglas Comer and CRC Press. All rights reserved.

ARM Clock Definitions (Part 1)

```
/* clock.h */

extern  uint32  clktime;        /* current time in secs since boot      */

extern  qid16   sleepq;         /* queue for sleeping processes         */
extern  int32   slnonempty;     /* nonzero if sleepq is nonempty        */
extern  int32   *sltop;         /* ptr to key in first item on sleepq   */
extern  uint32  preempt;        /* preemption counter                   */

struct am335x_timer1ms {
        uint32  tidr;           /* Identification register      */
        uint32  res1[3];        /* Reserved                     */
        uint32  tiocp_cfg;      /* OCP Interface register       */
        uint32  tistat;         /* Status register              */
        uint32  tisr;           /* Interrupt status register    */
        uint32  tier;           /* Interrupt enable register    */
        uint32  twer;           /* Wakeup enable register       */
        uint32  tclr;           /* Optional features            */
        uint32  tcrr;           /* Internal counter value       */
        uint32  tldr;           /* Timer load value             */
        uint32  ttgr;           /* Trigger register             */
        uint32  twps;           /* Write posting register       */
        uint32  tmar;           /* Match register               */
```

Copyright © 2019 by Douglas Comer and CRC Press. All rights reserved.

ARM Clock Definitions (Part 2)

```
        uint32  tcar1;          /* Capture register 1           */
        uint32  tsicr;          /* Synchronous interface control*/
        uint32  tcar2;          /* Capture register 2           */
        uint32  tpir;           /* Positive increment register  */
        uint32  tnir;           /* Negative increment register  */
        uint32  tcvr;           /* 1ms control register         */
        uint32  tocr;           /* Overflow mask register       */
        uint32  towr;           /* no. of overflows             */
};

#define AM335X_TIMER1MS_ADDR                    0x44E31000
#define AM335X_TIMER1MS_IRQ                     67

#define AM335X_TIMER1MS_TIOCP_CFG_SOFTRESET     0x00000002
#define AM335X_TIMER1MS_TISTAT_RESETDONE        0x00000001

#define AM335X_TIMER1MS_TISR_MAT_IT_FLAG        0x00000001
#define AM335X_TIMER1MS_TISR_OVF_IT_FLAG        0x00000002
#define AM335X_TIMER1MS_TISR_TCAR_IT_FLAG       0x00000004

#define AM335X_TIMER1MS_TIER_MAT_IT_ENA         0x00000001
#define AM335X_TIMER1MS_TIER_OVF_IT_ENA         0x00000002
#define AM335X_TIMER1MS_TIER_TCAR_IT_ENA        0x00000004
```

Copyright © 2019 by Douglas Comer and CRC Press. All rights reserved.

ARM Clock Definitions (Part 3)

```
#define AM335X_TIMER1MS_TCLR_ST                 0x00000001
#define AM335X_TIMER1MS_TCLR_AR                 0x00000002

#define AM335X_TIMER1MS_CLKCTRL_ADDR            0x44E004C4
#define AM335X_TIMER1MS_CLKCTRL_EN              0x00000002
```

Copyright © 2019 by Douglas Comer and CRC Press. All rights reserved.

Example Clock Interrupt Handler (Part 1)

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

#include <xinu.h>

/*-----------------------------------------------------------------------
 * clkhandler - high level clock interrupt handler
 *-----------------------------------------------------------------------
 */
void    clkhandler()
{

        static uint32 count1000 = 1000; /* variable to count 1000ms */
        volatile struct am335x_timer1ms *csrptr =
                        (struct am335x_timer1ms *)0x44E31000;
                        /* Set csrptr to address of timer CSR       */

        /* If there is no interrupt, return */

        if((csrptr->tisr & AM335X_TIMER1MS_TISR_OVF_IT_FLAG) == 0) {
                return;
        }
```

Copyright © 2019 by Douglas Comer and CRC Press. All rights reserved.

Example Clock Interrupt Handler (Part 2)

```
/* Acknowledge the interrupt */

csrptr->tisr = AM335X_TIMER1MS_TISR_OVF_IT_FLAG;

/* Decrement 1000ms counter */

count1000--;

/* After 1 sec, increment clktime */

if(count1000 == 0) {
        clktime++;
        count1000 = 1000;
}
```

Copyright © 2019 by Douglas Comer and CRC Press. All rights reserved.

Example Clock Interrupt Handler (Part 3)

```
        /* check if sleep queue is empty */

        if(!isempty(sleepq)) {

                /* sleepq nonempty, decrement the key of */
                /* topmost process on sleepq             */

                if((--queuetab[firstid(sleepq)].qkey) == 0) {

                        wakeup();
                }
        }

        /* Decrement the preemption counter */
        /* Reschedule if necessary          */

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

Copyright © 2019 by Douglas Comer and CRC Press. All rights reserved.

Clock Initialization (Part 1)

```
/* clkinit.c - clkinit (BeagleBone Black) */

#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
 *------------------------------------------------------------------------
 */
void    clkinit(void)
{
        volatile struct am335x_timer1ms *csrptr =
        (volatile struct am335x_timer1ms *)AM335X_TIMER1MS_ADDR;
                        /* Pointer to timer CSR in BBoneBlack   */
        volatile uint32 *clkctrl =
                        (volatile uint32 *)AM335X_TIMER1MS_CLKCTRL_ADDR;

        *clkctrl = AM335X_TIMER1MS_CLKCTRL_EN;
        while((*clkctrl) != 0x2) /* Do nothing */ ;
```

Copyright © 2019 by Douglas Comer and CRC Press. All rights reserved.

Clock Initialization (Part 2)

```
/* Reset the timer module */

csrptr->tiocp_cfg |= AM335X_TIMER1MS_TIOCP_CFG_SOFTRESET;

/* Wait until the reset is complete */

while((csrptr->tistat & AM335X_TIMER1MS_TISTAT_RESETDONE) == 0)
        /* Do nothing */ ;

/* Set interrupt vector for clock to invoke clkint */

set_evec(AM335X_TIMER1MS_IRQ, (uint32)clkhandler);

sleepq = newqueue();    /* Allocate a queue to hold the delta   */
                        /*   list of sleeping processes         */

preempt = QUANTUM;      /* Set the preemption time              */

clktime = 0;            /* Start counting seconds               */

/* The following values are calculated for a    */
/*   timer that generates 1ms tick rate         */

csrptr->tpir = 1000000;
csrptr->tnir = 0;
csrptr->tldr = 0xFFFFFFFF - 26000;
```

Copyright © 2019 by Douglas Comer and CRC Press. All rights reserved.

Clock Initialization (Part 3)

```
        csrptr->tpir = 1000000;
        csrptr->tnir = 0;
        csrptr->tldr = 0xFFFFFFFF - 26000;

        /* Set the timer to auto reload */

        csrptr->tclr = AM335X_TIMER1MS_TCLR_AR;

        /* Start the timer */

        csrptr->tclr |= AM335X_TIMER1MS_TCLR_ST;

        /* Enable overflow interrupt which will generate */
        /*   an interrupt every 1 ms                     */

        csrptr->tier = AM335X_TIMER1MS_TIER_OVF_IT_ENA;

        /* Kickstart the timer */

        csrptr->ttgr = 1;

        return;
}
```

Copyright © 2019 by Douglas Comer and CRC Press. All rights reserved.

Summary

- Two types of timed events are especially important in an operating system
  - Preemption
  - Process delay (sleep)
- A delta list provides an elegant and efficient data structure to store a set of sleeping processes
- If multiple processes awaken at the same time, rescheduling must be deferred until all have been made ready
- *Recvtime* allows a process to wait a specified time for a message to arrive

Copyright © 2019 by Douglas Comer and CRC Press. All rights reserved.

Questions?