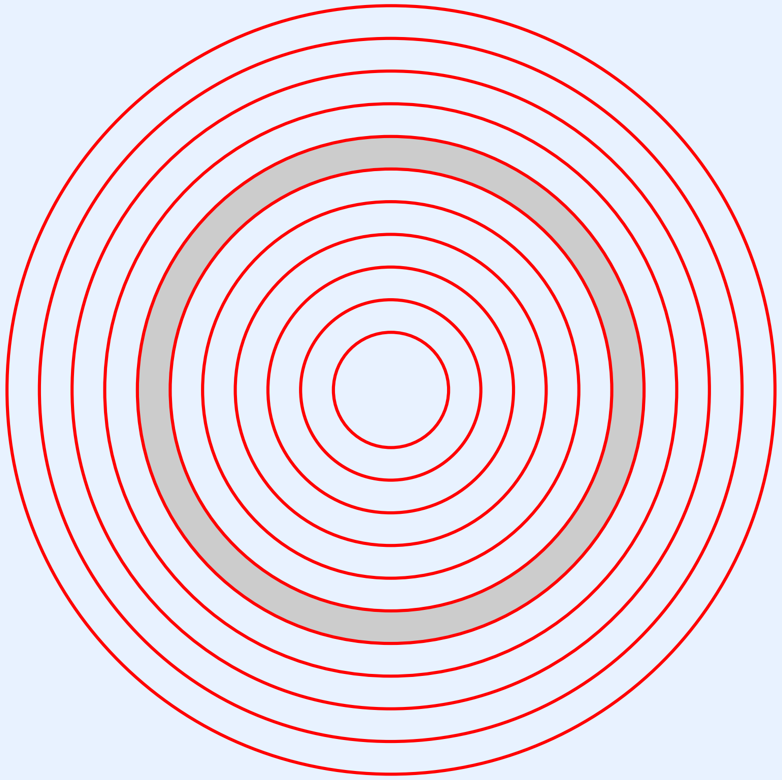
PART 8

Device Management

Location Of Device Management In The Hierarchy



Historical Approach

- Each device has unique hardware interface
- Code to communicate with device is part of application
- Application polls device, does not use interrupts
- Limits generality, portability

Modern Approach

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

Device Manager In An Operating System

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

A Conceptual Note

One of the most difficult aspects of operating systems arises from the interface 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

- CPU
  - Starts device
  - Enables interrupts
- Device
  - Performs requested operation
  - Raises interrupt on bus
- Processor hardware
  - Checks for interrupt
  - “Calls” interrupt procedure
  - Handles atomic return

Processes And Interrupts

- Key ideas
  - At any time, a process is running
  - An interrupt functions like 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 currently executing process executes interrupt code

Historic Interrupt Software

- Separate interrupt code for each device
  - Low-level
  - Handles housekeeping details
    - Saves / restores registers
    - Sets interrupt mask
  - Finds interrupting device on the bus
  - Handles interaction with device

Modern Interrupt Software

- Single function with common code
  - Known as *interrupt dispatcher*
  - Performs housekeeping details
  - Finds interrupting device on the bus
  - Calls device-specific function
- Separate routine for each device
  - Known as *interrupt handler*
  - Performs all interaction with device

Interrupt Dispatcher

- Low-level procedure
- Invoked by hardware when interrupt occurs
- Processor hardware
  - Saves instruction pointer when interrupt occurs
  - Sets processor mode (i.e., disables interrupts)
- Dispatcher
  - Saves other machine state as necessary
  - Identifies interrupting device
  - Establishes high-level run-time environment
  - Calls device-specific *interrupt handler*

Example

- Each physical device assigned unique interrupt vector in memory
- OS loads interrupt vector with address of dispatcher
- When interrupt occurs, processor
  - Disables further interrupts
  - Saves instruction pointer (for interrupt return)
  - Loads instruction pointer from interrupt vector
- Dispatcher
  - Identifies device
  - Calls interrupt handler for device

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*
- Hardware atomically
  - Resets instruction pointer to saved value
  - Enables interrupts

A Basic Rule For Interrupt Processing

- Facts
  - The CPU 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 interrupt occurs I/O 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

- 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 hander 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 re-establish the scheduling invariant
  - Arrange for the dispatcher to reestablish the scheduling invariant just before returning from the interrupt
- Either approach 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
  - Currently executing process executes interrupt code
- An important consequence
  - Because the null process may be running when an interrupt occurs, the null process *can* execute an interrupt handler

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}.

Rules For Interrupt Processing (Review)

- We assume

  Interrupts are disabled when dispatcher calls device-specific interrupt handler

- To remain safe

  Device-specific interrupt handler must keep further interrupts disabled until it completes changes to global data structures

- Can an interrupt handler call *resched*?

Rescheduling During Interrupt Processing

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

An Example

- Suppose
  - Process *T* is current
  - Interrupt occurs, so *T* executes interrupt handler
  - Interrupt handler calls *resched*, which switches to *S*
  - *S* runs 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
- Usually 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

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
  - Limited interaction with device hardware
  - Enqueues request
  - Starts device if idle
- Lower-half
  - Limited interaction with application
  - Talks to device to
    - Obtain incoming data
    - Start next operation
  - Starts process if any waiting

Coordination Of Processes Performing I/O

- Process may need to block when attempting to perform I/O
- Examples
  - Wait for incoming data to arrive
  - Wait for device to send outgoing data
- How should coordination be performed?

Answers

- Can use standard coordination mechanisms
  - Message passing
  - Semaphores
  - Suspend / resume
- Must be careful because lower-half cannot block

Using Message-Passing For Input Synchronization

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

Using Semaphores For Input Synchronization

- Shared buffer created with *N* slots
- Semaphore created with initial count *0*
- Upper-half
  - Calls *wait* on semaphore
  - Extracts next item from buffer and returns
  - Restarts device if necessary
- Lower-half
  - Places incoming item in buffer
  - Calls *signal* on the semaphore
- Note: semaphore counts items in the buffer

Semaphores And Output Synchronization

- A flawed approach
  - Semaphore counts items in buffer
  - Upper-half deposits an item and calls *signal*
  - Lower-half calls *wait* to wait for an item
- Why is the above flawed?

The lower-half cannot execute *wait*!

Using Semaphores For Output Synchronization

- Trick: semaphore 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
  - Starts next output operation
  - Signals semaphore to indicate space available

Device-Independent I / O

- Desires
  - Portability across machines
  - Generality sufficient for all devices
  - Elegance (not a potpourri of functions)
- Solution
  - Isolate application processes from device drivers
  - Use the *open-close-read-write* paradigm
  - Integrate with other operating system facilities

Approach To Device-Independent I / O

- Define a set of abstract operations
- Build a general-purpose mechanism
  - 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

Xinu Device-Independent I / O Primitives

init	– initialize device (once)
open	– make device ready
close	– terminate use of the device
read	– arbitrary data input
write	– arbitrary data output
putc	– single-character output
getc	– single-character input
seek	– position the device
control	– control device and / or driver

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

Implementation Of Device-Independent I/O In Xinu

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

Mapping Generic I/O Function To A Device-Specific Function

- Must be efficient
- Performed with a *device switch table*
  - Kernel data structure initialized when system loaded
  - One row per device
  - One column per operation
  - Entry points to a function
- Device ID is index into the table

Semantics Of Device-Independent I/O

- Each device-independent operation is generic
- A given operation may not make sense for a given device
  - Example: *seek* on keyboard or network
- However... all entries in device switch table must be valid
- Solution: special entries

Special Entries 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	
⋮				

- Row corresponds to device
- Column corresponds to operation

Replicated Devices

- Computer may contain multiple copies of a physical device
- Example: two serial lines
- Driver must know which physical copy to use
- Column in device table specifies *minor device number*
- Each instance of a replicated device is assigned a unique minor number (e.g., 0, 1, 2, ...)

Device Names

- Previous examples have shown device names used in code
  - CONSOLE
  - SERIAL0
  - SERIAL1
  - ETHER
- Device switch table is an array
- How are constants such as *CONSOLE* defined?
- Answer: OS designer specifies during configuration
- More details later

Initializing The I / O Subsystem

- Required steps
  - Fill in interrupt vectors
  - Initialize data structures, such as shared buffers
  - Create semaphores used for coordination
  - Fill in mapping table used by interrupt dispatcher
- Method: at startup, OS calls *init* for each device
- We will see more later

Example Device Driver

- Device type *tty*
- Used for I / O to serial lines
- Separate input and output buffers
- Uses semaphores to synchronize
- Example code assumes dumb device that reads or writes a single character at a time

Tty Driver Functions

Upper-Half	Lower-Half
ttyInit	ttyInterrupt (interrupt handler)
ttyOpen	ttyInter_in (input interrupt)
ttyClose	ttyInter_out (output interrupt)
ttyRead	
ttyWrite	
ttyPutc	
ttyGetc	
ttyCntl	

Actions During Character Output

- Output semaphore counts spaces in buffer
- Upper-half
  - Waits (on semaphore) for buffer space
  - Deposits character in next buffer slot
  - Triggers device
- Lower-half
  - Extracts character from next filled slot and passes to device
  - Signals space semaphore

Tty Driver Complexity

- Modes (cooked, cbreak, and raw)
- CRLF mapping
- Input character echo
- Visualization of echoed control characters (e. g., ^A)
- Editing, including backspace over control characters
- Flow control (^S /^Q)
- Many features can be changed dynamically

Definitions Used By Tty Driver (part 1)

```
/* tty.h */

#define TY_OBMINSP              20      /* min space in buffer before   */
                                        /* processes awakened to write  */
#define TY_EBUFLEN              20      /* size of echo queue           */

/* size constants */

#ifndef Ntty
#define Ntty            1               /* number of serial tty lines   */
#endif
#ifndef TY_IBUFLEN
#define TY_IBUFLEN      128             /* num. chars in input queue    */
#endif
#ifndef TY_OBUFLEN
#define TY_OBUFLEN      64              /* num. chars in output queue   */
#endif

/* mode constants for input and output modes */

#define TY_IMRAW        'R'             /* raw mode => nothing done     */
#define TY_IMCOOKED     'C'             /* cooked mode => line editing  */
#define TY_IMCBREAK     'K'             /* honor echo, etc, no line edit*/
#define TY_OMRAW        'R'             /* raw mode => normal processing*/
```

Definitions Used By Tty Driver (part 2)

```
struct  ttycblk {                       /* tty line control block       */
        char    *tyihead;               /* next input char to read      */
        char    *tyitail;               /* next slot for arriving char  */
        char    tyibuff[TY_IBUFLEN];    /* input buffer (holds one line)*/
        sid32   tyisem;                 /* input semaphore              */
        char    *tyohead;               /* next output char to xmit     */
        char    *tyotail;               /* next slot for outgoing char  */
        char    tyobuff[TY_OBUFLEN];    /* output buffer                */
        sid32   tyosem;                 /* output semaphore             */
        char    *tyehead;               /* next echo char to xmit       */
        char    *tyetail;               /* next slot to deposit echo ch */
        char    tyebuff[TY_EBUFLEN];    /* echo buffer                  */
        char    tyimode;                /* input mode raw/cbreak/cooked */
        bool8   tyiecho;                /* is input echoed?             */
        bool8   tyieback;               /* do erasing backspace on echo?*/
        bool8   tyevis;                 /* echo control chars as ^X ?   */
        bool8   tyecrlf;                /* echo CR-LF for newline?      */
        bool8   tyicrlf;                /* map '\r' to '\n' on input?   */
        bool8   tyierase;               /* honor erase character?       */
        char    tyierasec;              /* erase character (backspace)  */
        bool8   tyikill;                /* honor line kill character?   */
        char    tyikillc;               /* line kill character          */
        int32   tyicursor;              /* current cursor position      */
```

Definitions Used By Tty Driver (part 3)

```
        bool8   tyoflow;                /* honor ostop/ostart?          */
        bool8   tyoheld;                /* output currently being held? */
        char    tyostop;                /* character that stops output  */
        char    tyostart;               /* character that starts output */
        bool8   tyocrlf;                /* output CR/LF for LF ?        */
        char    tyifullc;               /* char to send when input full */
};
extern  struct  ttycblk ttytab[];

/* characters with meaning to the tty driver */

#define TY_BACKSP       '\b'
#define TY_BELL         '\07'
#define TY_BLANK        ' '
#define TY_NEWLINE      '\n'
#define TY_RETURN       '\r'
#define TY_STOPCH       '\023'          /* Control-S stops output       */
#define TY_STRTCH       '\021'          /* Control-Q restarts output    */
#define TY_KILLCH       '\025'          /* Control-U is line kill       */
#define TY_UPARROW      '^'
#define TY_FULLCH       TY_BELL         /* char to echo when buffer full*/
```

Definitions Used By Tty Driver (part 4)

```
/* tty control function codes */

#define TC_NEXTC        3               /* look ahead 1 character       */
#define TC_MODER        4               /* set input mode to raw        */
#define TC_MODEC        5               /* set input mode to cooked     */
#define TC_MODEK        6               /* set input mode to cbreak     */
#define TC_ICHARS       8               /* return number of input chars */
#define TC_ECHO         9               /* turn on echo                 */
#define TC_NOECHO       10              /* turn off echo                */
```

Xinu TtyPutc (part 1)

```
/* ttyPutc.c - ttyPutc */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  ttyPutc - write one character to a tty device (interrupts disabled)
 *------------------------------------------------------------------------
 */
devcall ttyPutc(
        struct  dentry  *devptr,        /* entry in device switch table */
        char    ch                      /* character to write           */
        )
{
        struct  ttycblk *typtr;         /* pointer to tty control block */

        typtr = &ttytab[devptr->dvminor];

        /* Handle output CRLF by sending CR first */

        if ( ch==TY_NEWLINE && typtr->tyocrlf ) {
                ttyPutc(devptr, TY_RETURN);
        }
```

Xinu TtyPutc (part 2)

```
	wait(typtr->tyosem);		/* wait for space in queue */
	*typtr->tyotail++ = ch;

	/* wrap around to beginning of buffer, if needed */

	if (typtr->tyotail >= &typtr->tyobuff[TY_OBUFLEN]) {
		typtr->tyotail = typtr->tyobuff;
	}

	/* start otuput in case device is idle */

	ttyKickOut(typtr, (struct uart_csreg *)devptr->dvcsr);

	return OK;
}
```

Xinu TtyGetc (part 1)

```
/* ttyGetc - ttyGetc */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  ttyGetc - read one character from a tty device (interrupts disabled)
 *------------------------------------------------------------------------
 */
devcall ttyGetc(
          struct dentry *devptr         /* entry in device switch table */
        )
{
        char    ch;
        struct  ttycblk *typtr;         /* pointer to ttytab entry */

        typtr = &ttytab[devptr->dvminor];

        /* wait for a character in the buffer */

        wait(typtr->tyisem);
        ch = *typtr->tyihead++;                 /* extract character    */
```

Xinu TtyGetc (part 1)

```
        /* wrap around to beginning of buffer, if needed */

        if (typtr->tyihead >= &typtr->tyibuff[TY_IBUFLEN]) {
                typtr->tyihead = typtr->tyibuff;
        }

        return (devcall)ch;
}
```

Xinu TtyWrite

/* ttyWrite.c - ttyWrite, writcopy */

```
#include <xinu.h>

/*------------------------------------------------------------------------
 *  ttyWrite - write character(s) to a tty device (interrupts disabled)
 *------------------------------------------------------------------------
 */
devcall ttyWrite(
          struct dentry *devptr,        /* entry in device switch table */
          char  *buff,                  /* buffer of characters         */
          int32 count                   /* count of character to write  */
        )
{
        if (count < 0) {
                return SYSERR;
        } else if (count == 0){
                return OK;
        }
        for (; count>0 ; count--) {
                ttyPutc(devptr, *buff++);
        }
        return OK;
}
```

Xinu TtyRead (part 1)

```
/* ttyRead.c - ttyRead, readcopy */

#include <xinu.h>

local   void    readcopy(struct ttycblk *, char *, int32);

/*------------------------------------------------------------------------
 *  ttyRead - read character(s) from a tty device (interrupts disabled)
 *------------------------------------------------------------------------
 */
devcall ttyRead(
          struct dentry *devptr,        /* entry in device switch table */
          char  *buff,                  /* buffer of characters         */
          int32 count                   /* count of character to write  */
        )
{
        struct  ttycblk *typtr;         /* pointer to tty control block */
        int32   avail;                  /* charscters available in buff.*/
        int32   nread;                  /* number of characters read    */

        if (count < 0) {
                return SYSERR;
        }
        typtr= &ttytab[devptr->dvminor];
        avail = semcount(typtr->tyisem);
```

Xinu TtyRead (part 2)

```
        /* count of zero means "returns all available characters" */

        if (count == 0) {
                if (avail == 0) {
                        return 0;
                } else {
                        count = avail;
                }
        }
        nread = count;

        /* if count characters are in the buffer, copy them */

        if (count <= avail) {
                readcopy(typtr, buff, count);
        } else {
                if (avail > 0) {                /* read avaialble chars */
                        readcopy(typtr, buff, avail);
                        buff += avail;
                        count -= avail;
                }
                for ( ; count>0 ; count-- ) {   /* get remaining chars  */
                        *buff++ = ttyGetc(devptr);
                }
        }
        return nread;
}
```

Xinu TtyRead (part 3)

```
/*------------------------------------------------------------------------
 *  readcopy - high speed copy procedure used by ttyRead
 *------------------------------------------------------------------------
 */
local   void    readcopy(
          struct ttycblk        *typtr,         /* pointer to tty control block */
          char  *buff,                        /* buffer of characters         */
          int32 count                         /* count of character to read   */
        )
{
        int32   n;                            /* loop index                   */

        for (n = count; n > 0; n--) {
                *buff++ = *typtr->tyihead++;

                /* wrap around to beginning of buffer, if needed */

                if ( typtr->tyihead >= &typtr->tyibuff[TY_IBUFLEN]) {
                        typtr->tyihead = typtr->tyibuff;
                }
        }
        semreset(typtr->tyisem, semcount(typtr->tyisem)-count);
}
```

Xinu TtyControl (part 1)

```
/* ttyControl - ttyControl */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  ttyControl  -  control a tty device by setting modes
 *------------------------------------------------------------------------
 */
devcall ttyControl(
          struct dentry *devptr,        /* entry in device switch table */
          int32  func,                  /* function to perform          */
          int32  arg1,                  /* argument 1 for request       */
          int32  arg2                   /* argument 2 for request       */
        )
{
        struct  ttycblk *typtr;         /* pointer to tty control block */
        char    ch;                     /* character for lookahead      */

        typtr = &ttytab[devptr->dvminor];
```

Xinu TtyControl (part 2)

```
/* process the request */

switch ( func ) {

case TC_NEXTC:
        wait(typtr->tyisem);
        ch = *typtr->tyitail;
        signal(typtr->tyisem);
        return (devcall)ch;

case TC_MODER:
        typtr->tyimode = TY_IMRAW;
        return (devcall)OK;

case TC_MODEC:
        typtr->tyimode = TY_IMCOOKED;
        return (devcall)OK;

case TC_MODEK:
        typtr->tyimode = TY_IMCBREAK;
        return (devcall)OK;

case TC_ICHARS:
        return(semcount(typtr->tyisem));
```

Xinu TtyControl (part 3)

```
        case TC_ECHO:
                typtr->tyiecho = TRUE;
                return (devcall)OK;

        case TC_NOECHO:
                typtr->tyiecho = FALSE;
                return (devcall)OK;

        default:
                return (devcall)SYSERR;
        }
}
```

Xinu TtyInterrupt (part 1)

```
/* ttyInterrupt.c - ttyInterrupt */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  ttyInterrupt - handle an interrupt for a tty (serial) device
 *------------------------------------------------------------------------
 */
interrupt ttyInterrupt(void)
{
        struct  dentry  *devptr;        /* pointer to devtab entry      */
        struct  ttycblk *typtr;         /* pointer to ttytab entry      */
        struct  uart_csreg *uptr;       /* address of UART's CSRs       */
        int32   iir = 0;                /* interrupt identification     */
        int32   lsr = 0;                /* line status                  */
        int32   ichars;                 /* incremented by count of chars*/
                                        /*  added to the input buffer   */

        /* For now, the CONSOLE is the only serial device */

        devptr = (struct dentry *)&devtab[CONSOLE];

        /* Obtain the CSR address for the UART */

        uptr = (struct uart_csreg *)devptr->dvcsr;

        /* Obtain a pointer to the tty control block */

        typtr = &ttytab[ devptr->dvminor ];
```

Xinu TtyInterrupt (part 2)

```
/* Decode hardware interrupt request from UART device */

/* Check interrupt identification register */
iir = uptr->iir;
if (iir & UART_IIR_IRQ) {
        return;
}

/* Decode the interrupt cause based upon the value extracted    */
/* from the UART interrupt identification register.  Clear      */
/* the interrupt source and perform the appropriate handling    */
/* to coordinate with the upper half of the driver              */

/* Decode the interrupt cause */

iir &= UART_IIR_IDMASK;         /* mask off the interrupt ID */
switch (iir) {

    /* Receiver line status interrupt (error) */

    case UART_IIR_RLSI:
        lsr = uptr->lsr;
        return;
```

Xinu TtyInterrupt (part 3)

```
/* Receiver data available or timed out */

case UART_IIR_RDA:
case UART_IIR_RTO:

    ichars = 0;

    /* For each char in UART buffer, call ttyInter_in */

    while (uptr->lsr & UART_LSR_DR) { /* while chars avail */
            ttyInter_in(typtr, uptr, &ichars);
    }

    /* all input has been extracted from the device and     */
    /* processed - now signal the input semaphore and start */
    /* output, if needed (e.g., for ^Q or echo)             */

    if (ichars > 0) {
            signaln(typtr->tyisem, ichars);
    }

    return;
```

Xinu TtyInterrupt (part 4)

```
            /* Transmitter output FIFO is empty (i.e., ready for more)  */

            case UART_IIR_THRE:
                lsr = uptr->lsr;  /* Read from LSR to clear interrupt */
                ttyInter_out(typtr, uptr);
                return;

            /* Modem status change (simply ignore) */

            case UART_IIR_MSC:
                return;
            }
}
```

Xinu TtyInter_in (part 1)

```
/* ttyInter_in.c.c ttyInter_in, erase1, eputc, echoch */

#include <xinu.h>

local   void    erase1(struct ttycblk *, struct uart_csreg *);
local   void    echoch(char, struct ttycblk *, struct uart_csreg *);
local   void    eputc(char, struct ttycblk *, struct uart_csreg *);

/*------------------------------------------------------------------------
 *  ttyInter_in  --  handle one arriving char (interrupts disabled)
 *------------------------------------------------------------------------
 */
void    ttyInter_in (
          struct ttycblk *typtr,        /* ptr to ttytab entry          */
          struct uart_csreg *uptr,      /* address of UART's CSRs       */
          int32  *ichars                /* ptr to input counter         */
        )
{
        char    ch;                     /* next char from device        */
        int32   avail;                  /* chars available in buffer    */

        ch = uptr->buffer;              /* extract char. from device    */

        /* compute chars available */

        avail = semcount(typtr->tyisem);
        if (avail < 0) {                /* processes waiting */
                avail = 0;
        }
```

Xinu TtyInter_in (part 2)

```
/* Handle raw mode */

if (typtr->tyimode == TY_IMRAW) {
        if (avail + *ichars >= TY_IBUFLEN) {    /* no space, so */
                return;                         /* ignore input */
        }

        /* place char in buffer with no editing */

        *typtr->tyitail++ = ch;
        *ichars = *ichars + 1;

        /* wrap buffer pointer  */
        if (typtr->tyotail >= &typtr->tyobuff[TY_OBUFLEN]) {
                typtr->tyotail = typtr->tyobuff;
        }
        return;
}
```

Xinu TtyInter_in (part 3)

```
/* Handle cooked and cbreak modes (common part) */

if ( (ch == TY_RETURN) && typtr->tyicrlf ) {
        ch = TY_NEWLINE;
}

if (typtr->tyoflow) {           /* are we doing flow control?   */
        if (ch == typtr->tyostart) {    /* ^Q starts output     */
                typtr->tyoheld = FALSE;
                ttyKickOut(typtr, uptr);
                return;
        } else if (ch == typtr->tyostop) {  /* ^S stops output  */
                typtr->tyoheld = TRUE;
                return;
        }
}

typtr->tyoheld = FALSE;         /* any other char starts output */
```

Xinu TtyInter_in (part 4)

```
if (typtr->tyimode == TY_IMCBREAK) {    /* just cbreak mode */

        /* if input buffer is full, send bell to user */

        if (avail + *ichars >= TY_IBUFLEN) {
                eputc(typtr->tyifullc, typtr, uptr);
                return;
        } else {        /* input buffer has space for this char */
                *typtr->tyitail++ = ch;
                *ichars = *ichars + 1;

                /* wrap around buffer */

                if (typtr->tyitail>=&typtr->tyibuff[TY_IBUFLEN]) {
                        typtr->tyitail = typtr->tyibuff;
                }

                if (typtr->tyiecho) {   /* are we echoing chars?*/
                        echoch(ch, typtr, uptr);
                }
        }
        return;
```

Xinu TtyInter_in (part 5)

```
} else {        /* cooked mode (see common code above */

        /* line kill character arrives - kill entire line */

        if (ch == typtr->tyikillc && typtr->tyikill) {
                typtr->tyitail -= typtr->tyicursor;
                if (typtr->tyitail < typtr->tyibuff) {
                        typtr->tyihead += TY_IBUFLEN;
                }
                typtr->tyicursor = 0;
                eputc(TY_RETURN, typtr, uptr);
                eputc(TY_NEWLINE, typtr, uptr);
                return;
        }

        /* erase (backspace) character */

        if ( (ch == typtr->tyierasec) && typtr->tyierase) {
                if (typtr->tyicursor > 0) {
                        typtr->tyicursor--;
                        erase1(typtr, uptr);
                }
                return;
        }
```

Xinu TtyInter_in (part 6)

```
/* end of line */

if ( (ch == TY_NEWLINE) || (ch == TY_RETURN) ) {
        if (typtr->tyiecho) {
                echoch(ch, typtr, uptr);
        }
        *typtr->tyitail++ = ch;
        if (typtr->tyitail >= &typtr->tyibuff[TY_IBUFLEN]) {
                typtr->tyitail = typtr->tyibuff;
        }
        /* make entire line available, incl. \n or \r   */
        *ichars += typtr->tyicursor + 1;
        typtr->tyicursor = 0;   /* reset for next line  */
        return;
}

/* normal character - send bell on buffer overflow */

avail = semcount(typtr->tyisem);
if (avail < 0) {
        avail = 0;
}
avail += *ichars;
if ((avail + typtr->tyicursor) >= TY_IBUFLEN-1) {
        eputc(typtr->tyifullc, typtr, uptr);
        return;
}
```

Xinu TtyInter_in (part 7)

```
                /* echo the character */

                if (typtr->tyiecho) {
                        echoch(ch, typtr, uptr);
                }

                /* insert in the input buffer */

                typtr->tyicursor++;
                *typtr->tyitail++ = ch;

                /* wrap around if needed */

                if (typtr->tyitail >= &typtr->tyibuff[TY_IBUFLEN]) {
                        typtr->tyitail = typtr->tyibuff;
                }
                return;
        }
}
```

Xinu TtyInter_in (part 8)

```
/*------------------------------------------------------------------------
 *  erase1  --  erase one character honoring erasing backspace
 *------------------------------------------------------------------------
 */
local   void    erase1(
          struct ttycblk        *typtr,         /* ptr to ttytab entry          */
          struct uart_csreg *uptr       /* address of UART's CSRs       */
        )
{
        char    ch;                     /* character to erase           */

        if ( (--typtr->tyitail) < typtr->tyibuff) {
                typtr->tyitail += TY_IBUFLEN;
        }

        /* pick up char to erase */

        ch = *typtr->tyitail;
        if (typtr->tyiecho) {           /* are we echoing? */
                if (ch < TY_BLANK || ch == 0177) {      /* nonprintable */
                        if (typtr->tyevis) {    /* visual cntl chars */
                                eputc(TY_BACKSP, typtr, uptr);
                                if (typtr->tyieback) { /* erase */
                                        eputc(TY_BLANK, typtr, uptr);
                                        eputc(TY_BACKSP, typtr, uptr);
                                }
                        }
```

Xinu TtyInter_in (part 9)

```
                    eputc(TY_BACKSP, typtr, uptr); /* go by up arrow */
                    if (typtr->tyieback) {
                            eputc(TY_BLANK, typtr, uptr);
                            eputc(TY_BACKSP, typtr, uptr);
                    }
            } else {        /* normal character */
                    eputc(TY_BACKSP, typtr, uptr);
                    if (typtr->tyieback) {  /* erase */
                            eputc(TY_BLANK, typtr, uptr);
                            eputc(TY_BACKSP, typtr, uptr);
                    }
            }
    }
    return;
}
```

Xinu TtyInter_in (part 10)

```
/*------------------------------------------------------------------------
 *  echoch  --  echo a character with visual and ocrlf options
 *------------------------------------------------------------------------
 */
local   void    echoch(
          char  ch,                     /* character to echo            */
          struct ttycblk *typtr,        /* ptr to ttytab entry          */
          struct uart_csreg *uptr       /* address of UART's CSRs       */
        )
{
        if ((ch==TY_NEWLINE || ch==TY_RETURN) && typtr->tyecrlf) {
                eputc(TY_RETURN, typtr, uptr);
                eputc(TY_NEWLINE, typtr, uptr);
        } else if ( (ch<TY_BLANK||ch==0177) && typtr->tyevis) {
                eputc(TY_UPARROW, typtr, uptr); /* print ^x             */
                eputc(ch+0100, typtr, uptr);    /* make it printable    */
        } else {
                eputc(ch, typtr, uptr);
        }
}
```

Xinu TtyInter_in (part 11)

```
/*------------------------------------------------------------------------
 *  eputc - put one character in the echo queue
 *------------------------------------------------------------------------
 */
local   void    eputc(
          char  ch,                     /* character to echo            */
          struct ttycblk *typtr,        /* ptr to ttytab entry          */
          struct uart_csreg *uptr       /* address of UART's CSRs       */
        )
{
        *typtr->tyetail++ = ch;

        /* wrap around buffer, if needed */

        if (typtr->tyetail >= &typtr->tyebuff[TY_EBUFLEN]) {
                typtr->tyetail = typtr->tyebuff;
        }
        ttyKickOut(typtr, uptr);
        return;
}
```

Xinu TtyKickOut

```
/* ttyKickOut.c - ttyKickOut */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  ttyKickOut - "kick" the hardware for a tty device, causing it to
 *              generate an output interrupt (interrupts disabled)
 *------------------------------------------------------------------------
 */
void    ttyKickOut(
         struct ttycblk *typtr,         /* ptr to ttytab entry          */
         struct uart_csreg *uptr        /* address of UART's CSRs       */
        )
{
        /* Set output interrupts on the UART, which causes */
        /*   the device to generate an output interrupt    */

        uptr->ier = UART_IER_ERBFI | UART_IER_ETBEI | UART_IER_ELSI;

        return;
}
```

Xinu TtyInter_out (part 1)

```
/* ttyInter_out.c - ttyInter_out */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  ttyInter_out - handle an output on a tty device by sending more
 *                 characters to the device FIFO (interrupts disabled)
 *------------------------------------------------------------------------
 */
void    ttyInter_out(
         struct ttycblk *typtr,         /* ptr to ttytab entry         */
         struct uart_csreg *uptr        /* address of UART's CSRs      */
        )
{
        int32   ochars;                 /* number of output chars sent */
                                        /*   the to UART               */
        int32   avail;                  /* available chars in output buf*/
        int32   uspace;                 /* space left in onboard UART  */
                                        /*   output FIFO               */

        /* If output is currently held, simply ignore the call */

        if (typtr->tyoheld) {
                return;
        }
```

Xinu TtyInter_out (part 2)

```
/* If echo and output queues empty, there is nothing to do */

if ( (typtr->tyehead == typtr->tyetail) &&
     (semcount(typtr->tyosem) >= TY_OBUFLEN) ) {
        return;
}

/* Initialize uspace to the size of the transmit FIFO */

uspace = UART_FIFO_SIZE;

/* While onboard FIFO is not full and the echo queue is */
/* is nonempty, xmit chars from the echo queue          */

while ( (uspace>0) &&  typtr->tyehead != typtr->tyetail) {
        uptr->buffer = *typtr->tyehead++;
        if (typtr->tyehead >= &typtr->tyebuff[TY_EBUFLEN]) {
                typtr->tyehead = typtr->tyebuff;
        }
        uspace--;
}
```

Xinu TtyInter_out (part 3)

```
        /* While onboard FIFO is not full and the output queue  */
        /* is nonempty, xmit chars from the output queue        */

        ochars = 0;
        avail = TY_OBUFLEN - semcount(typtr->tyosem);
        while ( (uspace>0) &&  (avail > 0) ) {
                uptr->buffer = *typtr->tyohead++;
                if (typtr->tyohead >= &typtr->tyobuff[TY_OBUFLEN]) {
                        typtr->tyohead = typtr->tyobuff;
                }
                avail--;
                uspace--;
                ochars++;
        }
        if (ochars > 0) {
                signaln(typtr->tyosem, ochars);
        }
        return;
}
```

Xinu TtyInit (part 1)

```
/* ttyInit.c - ttyInit */

#include <xinu.h>

struct  ttycblk ttytab[Ntty];

/*------------------------------------------------------------------------
 *  ttyInit - initialize buffers and modes for a tty line
 *------------------------------------------------------------------------
 */
devcall ttyInit(
          struct dentry *devptr         /* entry in device switch table */
        )
{
        struct  ttycblk *typtr;         /* pointer to ttytab entry      */
        struct  uart_csreg *uptr;       /* address of UART's CSRs       */

        typtr = &ttytab[ devptr->dvminor ];

        /* Initialize values in the tty control block */

        typtr->tyihead = typtr->tyitail =       /* set up input queue   */
                &typtr->tyibuff[0];             /*    as empty          */
        typtr->tyisem = semcreate(0);           /* input semaphore      */
        typtr->tyohead = typtr->tyotail =       /* set up output queue  */
                &typtr->tyobuff[0];             /*    as empty          */
        typtr->tyosem = semcreate(TY_OBUFLEN);  /* output semaphore     */
```

Xinu TtyInit (part 2)

```
typtr->tyehead = typtr->tyetail =      /* set up echo queue    */
        &typtr->tyebuff[0];            /*    as empty          */
typtr->tyimode = TY_IMCOOKED;          /* start in cooked mode */
typtr->tyiecho = TRUE;                 /* echo console input   */
typtr->tyieback = TRUE;                /* honor erasing bksp   */
typtr->tyevis = TRUE;                  /* visual control chars */
typtr->tyecrlf = TRUE;                 /* echo CRLF for NEWLINE*/
typtr->tyicrlf = TRUE;                 /* map CR to NEWLINE    */
typtr->tyierase = TRUE;                /* do erasing backspace */
typtr->tyierasec = TY_BACKSP;          /* erase char is ^H     */
typtr->tyikill = TRUE;                 /* allow line kill      */
typtr->tyikillc = TY_KILLCH;           /* set line kill to ^U  */
typtr->tyicursor = 0;                  /* start of input line  */
typtr->tyoflow = TRUE;                 /* handle flow control  */
typtr->tyoheld = FALSE;                /* output not held      */
typtr->tyostop = TY_STOPCH;            /* stop char is ^S      */
typtr->tyostart = TY_STRTCH;           /* start char is ^Q     */
typtr->tyocrlf = TRUE;                 /* send CRLF for NEWLINE*/
typtr->tyifullc = TY_FULLCH;           /* send ^G when buffer  */
                                       /*   is full            */
/* Initialize the UART */

uptr = (struct uart_csreg *)devtab[CONSOLE].dvcsr;
uptr->lcr = UART_LCR_8N1;      /* 8 bit char, No Parity, 1 Stop*/
uptr->fcr = 0x00;              /* Disable FIFO for now         */
/* OUT2 value is used to control the onboard interrupt tri-state*/
/* buffer. It should be set high to generate interrupts         */
uptr->mcr = UART_MCR_OUT2;     /* Turn on user-defined OUT2    */
```

Xinu TtyInit (part 3)

```
        /* Enable interrupts */

        /* Enable UART FIFOs, clear and set interrupt trigger level     */
        uptr->fcr = UART_FCR_EFIFO | UART_FCR_RRESET
                        | UART_FCR_TRESET | UART_FCR_TRIG2;

        /* Register the interrupt handler for the dispatcher */

        interruptVector[devptr->dvirq] = (void *)devptr->dvintr;

        /* Ready to enable interrupts on the UART hardware */

        enable_irq(devptr->dvirq);

        ttyKickOut(typtr, uptr);

        return OK;
}
```

Summary

- OS component to handle I/O known as *device manager*
- 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

Summary
(continued)

- To accommodate null process, interrupt handler must leave executing process in *current* or *ready* states
- Rescheduling during interrupt safe if
  - Global data structures valid
  - No process explicitly enables interrupts
- Device driver functions
  - Are divided into upper-half and lower-half
  - Can use process synchronization primitives
- Many details complicate a basic tty driver