COMP 242 Class Notes
Section 4: Interrupt Processing

1 Time Management

We shall now study how the operating system manages a list of "events", ordered by the time at which they should occur. Some of these events are scheduled by the operating system itself, others are scheduled by user processes. We study both kinds of events below.

1.1 Preemption Events

An operating system schedules **preemption events** to guarantee that equal priority processes receive round-robin service, discussed earlier. These events are scheduled at context switches, and ensure that a process cannot run continuously for more than a time determined by the **granularity of preemption**. Thus if a process becomes *current* at time 't', then a preemption event is scheduled for time 't + QUANTUM', where QUANTUM is a constant whose value determines the granularity of preemption. A preemption event is cancelled if a process relinquishes control of the CPU before its time expires.

An operating system has to be careful in selecting a suitable granularity of preemption. Setting it too low, say 0.01 seconds, would be inefficient since the processor would spent too much time rescheduling. Setting it too high, say 100 seconds, would allow compute-bound jobs to 'hog' the computer. In Xinu the granularity of preemption is 1 second. Older versions of Unix also used this granularity but BSD uses 0.1 second. Few processes, however, use their allocated quantums, since they make frequent calls to I/O and other routines that call *resched*.

1.2 Wakeup Events

A user process schedules a **wakeup event** when it asks the operating system to suspend it for a period specified by the process. A wakeup event is scheduled to occur at the end of the period. (Why are wakeup events useful?)

In Xinu, a process schedules wakeup events by calling either *sleep10(n)* or *sleep(n)*. The parameter of the former specifies time in one tenths of seconds, while the parameter of the latter specifies it in seconds. Thus the former can be used to schedule delays of upto 2**15 - 1 tenths of seconds (3276.7 seconds) and the latter 32767 seconds.

1.3 Implementation

We shall now see how an operating system supports wakeup and preemption events. Like some other OS features we have seen earler, these events require hardware support in the form of a real-time clock. We shall study first this hardware and then other features required to support time management.

Real-Time Clock

A real-time clock is a hardware device that pulses regularly an integral number of times each second, and interrupts the CPU each time a pulse occurs. It is important to distinguish this device from the **time-of-day** clock that keeps the current time. A time-of-day clock also emits pulses at regular intervals. However, unlike the real time clock, it does not interrupt the CPU after each pulse. Instead, it increments a counter, which the CPU may read to determine the current time. A real-time clock is also different from the CPU clock, which is not visible to an operating system.

The number of times a clock pulses in a second is called **clock rate**. Typical clock rates are 10Hz, 60Hz, 100Hz, and 1000Hz. In Xinu, the clock rate of the real-time clock is 60Hz.

It is important that clock interrupts be serviced promptly to ensure that they are not lost. The hardware helps by giving a clock the highest priority among the various devices that can interrupt. (In x86 architectures, power outage and interprocessor interrupts are given higher priority than clock, but these are not really devices.) Interrupts are serviced in the order of their priority, and a higher level interrupt can preempt a lower-level one. The software helps by keeping making clock interrupt processing efficient by reducing the number of procedure calls made.

Even if clock interrupt processing can keep up with the clock rate, it is important that the processor not spend all its time processing clock interrupts. This may happen with slow CPUs such as the LSI-11. Therefore it is necessary to adjust the clock rate to match the system.

Ideally, the hardware clock should be slowed down, but this is inconvenient or impossible. Instead, a slower clock can be simulated by the interrupt dispatcher. The dispatcher can divide the clock rate by ignoring a certain number of interrupts before it processes one. For instance, the clock dispatcher in Xinu ignores five pulses before it processes one. Thus the clock rate of 60 Hz is divided by 6 to 10 Hz. This slower simulated rate is called the **tick rate**, and determines the smallest size of the granularity of preemption and timed delays. (Is this small size a problem?)

Not all computers are connected to a real-time clock. Systems like Xinu that are designed to run on such systems need to check through software if a clock exists. In Xinu the routine *_setclkr* performs this checking task. It executes a loop 30,000 times, and if the clock interrupts during this time, it sets a global flag *clkruns* to true.

Delta List Processing

We now look at a data structure required to maintain wake-up events. The system keeps a queue of sleeping processes ordered by the time at which they are to be woken up. An element of the queue needs to keep some information about the time at which it is to be woken up. This information is kept in the 'key' field of a process table entry. The field could contain:

absolute time of the wake-up event,

time relative to the current time,

the **delta** (difference) between the times of its wake-up event and the wake-up event of the process in front of it in the queue.

(All times are in units of clock ticks).

Thus if there are four processes with wake-up events scheduled at times 1017, 1027, 1028, 1032, then at time 1000 the keys could contain one of the following three possibilities:

```
1017 1027 1028 1032--absolute
17 27 28 32--relative
17 10 1 4--delta
```

A disadvantage of keeping absolute times is that the key fields and the counter that keeps the current time might overflow. A disadvantage of keeping relative times is that every key element needs to be updated on each clock tick. Therefore deltas are kept, and the queue is called the **delta list**.

The delta list is associated with the procedure *insertd (pid, head, key)*, which, given the relative time of the wake-up event of process *pid* in *key*, inserts the process in the appropriate place in the delta list "head" with the appropriate delta stored in the key field. To find the place of the process, the procedure keeps subtracting from *key* the delta of the processes it visits, till it comes across a process *q* whose delta is greater than *key*. It inserts *p* in front of *q* with its key equal to the current value of the parameter *key*. It also subtracts *key* from q's delta. Thus if at time 1000, a wake-up event for time 1030 was to be inserted, then the following are the values of *key* as various elements in the list are visited:

```
initial: 30
visits 1st process: 30 > 17 so key = 30 - 17 = 13
visits 2nd process: 13 > 10 so key = 13 - 10 = 3
visits 3rd process: 3 > 1 so key = 3 -1 = 2
visits 4th process: 2 < 4, so insert process with delta 2, and modify next delta to
4 - 2 = 2.
```

Sleep10

Given the delta list and procedure *insertd*, the implementation of *sleep10 (n)* is easy. It needs to block the process by changing its state and putting it in the delta list. To do so, it:

calls *insertd(currpid, clockq, n)*, where *clockq* is a global variable.

changes state of process to SLEEP and calls resched

It also takes the following steps for efficiency reasons, which are crucial to make clock interrupt processing fast:

It indicates in global variable *slnempty* that the delta list is non-empty (the clock dispatcher uses this variable for efficient processing)

It stores address of the key of the process at the head of the queue in variable *sltop* (to save the dispatcher from doing this job at clock interrupt time)

Sleep

The routine *sleep(n)* calls sleep10() multiple times. A naive implementation would simply call sleep10(n) 10 times. Xinu does better than this in terms of the number of system calls made. The call *sleep(n)* invokes sleep10(10*1000) n **div** 1000 times and then once sleep10(10* (n **mod** 1000)). Thus the argument of each of these sleep10 is less than 32767 and together all calls delay 10*n tenths of seconds. (Can we do even better than this in terms of number of system calls made?)

1.3.1 Clock Dispatcher

This is the routine called on each clock interrupt. We shall first look at the functions of the dispatcher and then its implementation.

Functions of the Dispatcher

The clock dispatcher provides the following functions:

conversion of clock rate to tick rate

service of preemption event scheduled for current clock tick.

service of wake-up event(s) scheduled for current tick.

deferred processing (discussed below)

Deferred Processing

Interrupts are disabled while a clock routine is manipulating global data such as the delta and ready lists. This situation presents a problem in Xinu on LSI 11/2 since interrupts from fast communication devices cannot be serviced. As a result data can be lost.

To prevent such loss of data, the dispatcher supports **deferred processing**. When a clock is in the deferred mode (similar to the delay mode of the output dispatcher discussed under input/output) clock ticks are accumulated without servicing events. The dispatcher schedules these deferred events when it next goes into the normal mode. Special procedures are provided to set the mode of the dispatcher. The Xinu network routines call these procedures. They put the clock in a deferred mode at the start of the transfer of data and back into the normal mode at the end of this transfer.

Implementation

The clock dispatcher takes the following steps:

Is this the 6th interrupt? If yes go ahead, otherwise return.

Are clock interrupts deferred? If yes, increment the count of deferred clock ticks and return, otherwise go ahead.

Are there any sleeping processes (is *slnempty* true)? If yes process wake-up events otherwise go ahead.

Process preemption events.

Processing Wake-Up Events

Processing of wake-up events is done as follows:

The delta key stored of process at head of the delta list is decremented. Recall that the address of this key is kept in *sltop*.

If the key goes to zero, then this process and all other processes with a delta of 0 compared to this process are put in the ready list, and *resched* is called.

The variables *sltop* and *slnempty* are reevaluated.

Processing of Preemption Events

Processing of preemption events is easy.

A global variable PREEMPT keeps the number of clock ticks before the current process is to be preempted. This variable is decremented.

If the variable goes to zero, then *resched* is called, which reschedules and resets PREEMPT to QUANTUM– the granularity of preemption.

2 Input/Output

A cpu is connected to one or more devices with which it can communicate information. Examples of these devices are disks, terminals and magnetic tapes. Each of these devices defines a hardware interface which may be used by a program to communicate with it. This interface, however, is *low-level* and unusable by most programs. Therefore an operating system provides a *high-level* **input/output** interface which user programs use to interact with devices. This interface serves two purposes:

It **embellishes** the low-level hardware interface. This feature is consistent with an operating system's goal of 'beautifying' the physical machine to give a more pleasant virtual machine.

It provides fair, safe, and efficient policies to **share** a common pool of devices.

The hardware interface provided by a device, the (high-level) I/O interface that embellishes it, and the implementation of the I/O have both some **device independent** and **dependent** components. We shall first study the device independent parts of the three, and then the device dependent aspects as we cover each device individually. Our discussion will be centered around the XINU OS and 11/2 hardware.

2.1 Hardware Aspects

Device Registers

A device is associated with several **device registers**, which in the LSI 11 are accessible as part of the physical memory. In the 11/2 the last 8K of addresses are used to name these registers. While these registers appear to a process as part of real memory, they are actually part of the device. They are used for four purposes:

To transmit *control* information from the cpu to a device.

To transmit *status* information from a device to the cpu.

To transfer *output* data from cpu to a device.

To transfer *input* data from a device to the cpu.

These four functions need not be served by four different registers. For instance, in the LSI 11, the control and status information of the terminal device are stored in one register. Moreover, not all devices provide both output and input registers. For instance a cardreader is associated only with an input register.

The device registers are assigned contiguous addresses. We shall call the lowest of these the **device address** of the device. The address of any device register can be computed from this address.

Polling vs Interrupts

The cpu needs some mechanism to determine when an I/O operation completes. Two mechanisms are provided: **polling** and **interrupts**. Polling means checking the status information of all busy devices periodically to see if the operation has completed. (A *ready* bit determines if a device has completed an operation or not.) Polling has the disadvantage of *busy waiting*. On the other hand it is easier to implement

I/O routines using polling rather than interrupts. Operating systems typically provide a polling based print routine that is used to debug interrupt routines.

Because polling is time consuming, it is more common to request the device to cause an interrupt instead when the device becomes ready. In the 11/2 an interrupt causes the following:

The current PC and PS are pushed on the stack.

A new PC and PS are loaded from an area in memory called the **interrupt vector** of the device. This area is addressed by the **interrupt vector address** of the device. In the 11/2 addresses from 0 to 0777 are used for interrupt vectors.

The code addressed by the new PC is executed. This code, called the **interrupt handler** of the device, eventually returns control to the place at which the user's program was interrupted.

Some devices may be associated with separate **input interrupt vectors** and **output interrupt vectors**. Such devices may be associated with distinct **input handlers** and **output handlers** that may be used to provide separate handling of input and output. We shall see later how the input and output interrupt vectors of a device are initialized.

2.2 Input/Output Primitives

We now study the high-level input/output primitives. We first look at some of the issues in the design of these primitives and then examine the primitives provided in Xinu.

2.2.1 Issues

There are several similarities between process-device communication and process-process communication. Therefore some of the issues we see here are reminiscent of those we saw in interprocess communication.

Synchronous vs Asynchronous

An I/O primitive is **synchronous** if it delays the process until the operation completes. Some other process may, however, utilize the CPU. It is **asynchronous** if it only initiates the operation and lets the process continue execution while the device performs the operation.

The arguments for and against these two kinds of primitives are similar to the ones we saw in interprocess communication. Synchronous communication is normally preferred because of its ease of use.

Device Dependent/Independent

Another issue is whether a general set of primitives be provided that may be used for all devices or if each device should be associated with its own set of primitives. Device Independent I/O promotes uniformity and allows one device to be substituted for another. Its disadvantage is that a general set of primitives may not fit the needs of a current or future device. For instance the operation *seek* is applicable to disks but not to terminals.

Binding Time Issues

A process refers to a device through a **device descriptor**. This descriptor needs to be **bound** to an actual device. This binding may take place at:

program writing time, if device descriptors are device addresses

system generation time, if device addresses refer to devices indirectly (via a table) that is initialized at system generation time

program start time, if the command interpreter does the binding

execution time, if the program or the operating system can change the binding at run time.

Early binding is efficient whereas late binding is flexible.

Pseudo Devices

An operating system often provides **pseudo** devices that are built on top of actual devices. For instance Unix-inspired operating systems provide files, which are pseudo devices built on top of a disk. NT also defines as pseudo devices filters for communicating with servers. Supporting pseudo devices has the advantage that resources other than real devices can be accessed through a device-independent interface.

When pseudo devices are provided, device descriptors refer to both actual and pseudo devices.

Size of Data

When device independent I/O is provided, it is important to decide the increment of data transferred: byte or block. Some devices work with single bytes while others work with blocks. A general purpose OS will need to provide both kinds of sizes.

2.3 XINU I/O Primitives

getc(descrp): receive a single character from device

putc(descrp, ch): send a single character to device

read(descrp, buff, count): get *count* number of characters into buff.

write(descrp, buff, count): send *count* number of characters from *buff* to device

control(descrp, func, addr, addr2): ‘control’ device

seek(descrp, pos): special form of control that positions device and is applied only to randomly accessed devices.

open(descrp, nam, mode): inform device that data transfer will begin

close(descrp): inform device that data transfer has ended

Not all of these operations apply to all devices. For instance, none of these apply to the clock. These operations apply only to devices that may be considered sequential streams of data. Moreover, *open* and *close* are not needed for terminal devices and *seek* is not possible on such devices.

2.4 Implementation

We now study (device-independent aspects of the) implementation of the Xinu I/O primitives.

Device Drivers

Each device is associated with several **device drivers** that implement the abstract I/O primitives for the device. For instance, a terminal device is associated with the device drivers *ttygetc*, *ttyputc*, *ttyread* ..., which implement the *getc*, *putc*, *read* ... abstract operations on terminal devices.

A device driver provides a device-specific implementation of an abstract operation. Different device types are associated with different device drivers, while devices of the same type can share device drivers.

Device Switch Table

Xinu provides a *device switch table* which maps abstract operations to device drivers, maps device descriptors to device addresses, and specifies contents of interrupt vectors. Each device is associated with an entry in the table which contains the following information about the device:

the device descriptor,

the addresses of the device drivers of the device,

device address,

addresses of input and output interrupt vectors,

addresses of input and output interrupt routines,

buffer pointer, and

minor number, which distinguishes between different devices of the same type and allows them to share the same device drivers.

The device switch table is indexed by device descriptors.

Mapping Abstract Operations to Device Drivers

The device switch table is used to map abstract operations such as *read* into calls to device drivers such as *ttyread*. For instance the body of the procedure *read(descrp, buff, count)* uses *descrp* to locate the appropriate device switch table entry, get the address of the 'read' driver for the device and calls the driver with the arguments *devptr*, *buff*, and *count*, where *devptr* is the address of the device table entry, and can be used by the device driver to access information in the entry. The bodies of other operations are similar.

Null and Error Entries in Device Table

A device table entry contains a procedure address for every abstract operation. However, some operations are not necessary for certain devices and others are erroneous. Therefore Xinu provides two routines *ionull* and *ioerr*. The former does nothing and returns a status of *OK*. Entries for unnecessary operations such as an *open* of a terminal device point to this procedure. The latter returns an error status. Entries for erroneous operations such as a *seek* of a terminal device point to this procedure.

Initializing the Device Table

The declaration of both the device table and its initial values (which do not change) is contained in a file called *conf.c* (see pages 154-155 of text). This file is automatically generated at system generation time. Thus the device table is initialized by the time the system is compiled.

Note that *conf.c* does not declare the structure of a device table entry, but only the contents. The former is found in another file (*conf.h*).

2.4.1 Interrupt Processing

Structure of Interrupt Handlers

In the writing of interrupt handlers we face the following two contradictory goals:

they should be written in high-level languages so that they are easy to understand and modify

they should be written in assembly language for efficiency reasons and because they manipulate hardware registers and use special call/return sequences that cannot be coded in high-level languages

To satisfy both goals Xinu employs the following two-level strategy. Interrupts branch to low-level interrupt **dispatch routines** that are written in assembly language. These handle low-level tasks such as saving registers and returning from the interrupt when it has been processed. However, they do little else – they call high-level **interrupt routines** to do the bulk of interrupt processing, passing them enough information to identify the interrupting device.

Xinu provides three interrupt dispatchers: one to handle input interrupts, one to handle output interrupts, and one to handle clock interrupts. Input and output dispatchers are separated for convenience, and a special clock dispatcher is provided for efficiency reasons.

NT provides an even more modular structure. A single routine, called the trap handler, handles both traps (called exceptions by NT) and interrupts, saving and restoring registers, which are common to both. If the asynchronous event was an interrupt, then it calls an interrupt handler. The task of this routine is to raise the processor priority to that of the device interrupting (so that a lower-level device cannot preempt), call either an internal kernel routine or an extrenal routine called an ISR, and then restore the processor priority. These two routines roughly correspond to our high-level interrupt routine and the trap handler corresponds to our low-level routine. Thus, NT trades off the efficiency of 2 levels for the reusability of 3 levels.

The device table entry for an input or output interrupt handler points at the high-level part of the interrupt handler, which is device-specific, and not the low-level part which is shared by all devices (except the clock).

Identifying the High-Level Routine

If all input (output) interrupts branch to the same input (output) dispatch routine, how does the dispatcher know which device-specific interrupt routine to call? The input (output) dispatch routine needs some way to discover the device that interrupted it, so that it can use this information to call the appropriate high-level routine. There are several ways to identify an interrupting device. Here are two of them:

The dispatcher may use a special machine instruction to get either the device address or the interrupt vector address of the device. Not all machines have such instructions.

The dispatcher may poll devices until it finds one with an interrupt pending.

The 11/2 has no special instruction to help identify the interrupting device. Therefore, the following 'trick' is used, which is common to other operating systems for the same architecture. The device descriptor (not the device address) is stored in the second word of the interrupt vector. Recall that this word stores the value to be loaded into the PS register when the interrupt routine is called. Xinu uses the lower order 4 bits, which are used for condition codes, to store the descriptor of the device. These four bits are then used to identify the high-level routine.

Interrupt Dispatch Table

An interrupt **dispatch table** is used to relate device descriptors with (high-level) interrupt routines. The table is indexed by a device descriptor and each entry contains the following information:

the address of the input interrupt routine

an *input code* which is passed as an argument to the input interrupt routine

the address of the output interrupt routine

an *output code* which is passed as an argument to the above routine

The input (output) dispatch routine uses the device descriptor to access the appropriate dispatch table entry and calls the input (output) interrupt routine with the input (output) code as an argument code. The input and output codes can be anything the high-level routines need. In Xinu, they are initially the minor number of the device. Thus only one interrupt routine is needed for all devices of the same type. The minor number is used to distinguish between these devices.

Initialization of Dispatch Table and Interrupt Vectors

A routine called *ioinit(descrp)* initializes both the dispatch table entry and the input and output vectors of the device. The dispatch table is initialized as follows:

the addresses of the input and output interrupt routines are obtained from the device table

the input and output codes are assigned the minor number of the device

The input (output) vector is assigned as follows:

the PC points to the input (output) dispatcher shared by all devices

the condition four bits of the PS are assigned the device descriptor and the priority bits the highest possible priority so that interrupts are disabled.

Input and Output Dispatchers

The input (output) dispatcher does the following tasks:

saves the value of PS on the stack (why?)

saves the value of register *r0* and *r1*, which are the only registers it will modify

gets device descriptor from saved value of PS

stores address of the input (output) routine in a register

pushes the input (output) code on stack

does a *jsr* to the interrupt routine

restores *r0* and *r1* from the stack

pops the arg, saved values of *r0* and *r1* and PS from stack

executes a 'return from interrupt' instruction that restores the PC and PS value current when the interrupt routine was called.

Rules for Interrupt Processing

There are several rules for interrupt processing: First, they should ensure that shared data are not manipulated simultaneously by different processes. One way to do so is to make interrupt routines uninterruptible. Thus the PS value stored in the interrupt vector of a device disables interrupts. This value is loaded when the interrupt handler is invoked. As a result, the interrupt routine is uninterruptible while the PS maintains this priority.

However, the PS may be changed while the interrupt routine is executing if it calls *resched*, which may switch to a process that has interrupts enabled. Therefore, the interrupt routine has to ensure that it completes changes to global data structures before it makes any call that results in context switching.

An interrupt routine should also make sure that it does not keep interrupts disabled too long. For instance, if the processor does not accept a character from an input device before another arrives, data will be lost.

Finally, interrupt routines should never call routines that could block the current process (that is the process executing when the interrupt occurred) in some queue. Otherwise, if the interrupt occurs while the null process is executing, the ready list will be empty. However, resched assumes that there is always some process to execute! Some process should always be runnable to that interrupts can be executed. Thus interrupt routines need to call only those routines that leave the current process in the *ready* or *current* state, and may not call routines such as *wait*.

Rescheduling while Processing an Interrupt

We assumed above that interrupt routines could call *resched*. We now answer the following questions: First, is it useful to do so? Second, is it safe?

It is useful to call *resched* from an interrupt routine. An output routine after removing a character from a buffer may *signal* a semaphore to allow another process to write data to the buffer space that it makes available. Similarly, an input routine might *send* data it obtains from the device to a process. In each case, the routine *resched* is called.

It is also safe to call *resched*. Intuitively it may not seem so, because switching to a process that has interrupts enabled could lead to a sequence of interrupts piling up until the stack overflowed. However, such a danger does not exist for the following reason: A process that is executing an interrupt handler cannot be interrupted again (why?). Some other process, however, can be. Thus a process's stack will hold the PS and PC value for only one interrupt and there will never be more interrupts pending than the number of processes in the system.

3 The Terminal Device

We now study an implementation of the I/O interface described earlier for the terminal device. We first study the hardware, and then the drivers and interrupt routines for the device.

3.1 Hardware

Serial Line Unit

From the computer's point of view the terminal device consists of a **serial line unit** (SLU) attached to the system bus. A cpu may have more than one such unit attached to it. Each SLU is connected to a terminal via a cable consisting of three wires: one carries data from the transmitter of a terminal to the receiver of the device, another carries data from the transmitter of the SLU to the receiver of the terminal, and the third provides an electrical ground.

Signals on data wires consist of a series of positive or negative pulses corresponding to the 1's and 0's being transmitted. They are *serial* because they travel down the wire one bit at a time, and *asynchronous* because the transmitter sends a character whenever one is available; there is no synchronization between the transmitter and receiver to control the start of a character.

In order to permit the receiver to recognize the start of a character, a **start** bit is transmitted directly before each character. To improve reliability one or two **stop** bits are sent directly after each character. A character may be prefixed by a **parity** bit for error detection purposes.

Errors occur when the receiver cannot make sense out of the signals it receives. A **framing error** occurs when a receiver's samples of a pulse contain both 1's and 0's. **Character overrun errors** occur when the CPU does not extract a character from a receiver before another is received. A **break** error occurs when the line remains idle in the wrong state for an extended period (greater than the time required to transmit a character). Receivers report break errors as framing error. Some systems use the break condition to trigger special handling. For example, the 11/2 can be wired so breaks on the console terminal line cause the processor to halt. The Xinu downloader uses this mechanism to gain control of the system.

Device Registers

An SLU is associated with four 16-bit device registers: the **receiver control and status register** (RCSR), the **received data buffer** (RBUF), the **transmitter control and status register** (XCSR), and the **transmitted data buffer** (XBUF). The higher bytes of all registers except RBUF are unused.

The XBUF register is used to output data to a terminal. Writing a character to the XBUF register address causes the SLU to capture the character and start transmitting it on the serial output line.

The RBUF register is used for receiving input data. The CPU reads from the RBUF address to retrieve a received character. The low-order byte of the register contains the character and the higher order byte is used to indicate overflow, framing, and parity errors (see section 2.1.10 in the text). Accessing the register has the side effect of clearing the receiver and enabling it to receive the next character.

RCSR is used to control the receiver and receive status information from it. Only two bits of the register are used: bit 6 is the **interrupt bit** and bit 7 is the **ready bit**. If the interrupt bit is set, the SLU will post an interrupt when a character is received. The ready bit is set by the SLU when a character has been received and is ready for the CPU to access it. This bit may be used for polled input.

XCSR is similar to RCSR; it contains a ready bit and an interrupt bit. The ready bit is set when the transmitter finishes transmission of a character and is ready to receive another. This bit may be used for polled output. If the interrupt bit is on, an interrupt is posted when the transmitter is ready. In addition to these two bits, XCSR also contains a **break bit**. When this bit is set to 1, the transmitter forces the output line into a break condition.

3.2 The Terminal Driver

We now study the device drivers for the terminal device. Together, these drivers are referred to as the **terminal driver**. A terminal driver implements two interfaces: an **I/O interface** that processes may use to interact with a user and a **user interface** that users may use to interact with processes. We saw the I/O interface earlier. The user interface has the following main characteristics:

characters are echoed
backspace and line-kill are supported
user is allowed to start and stop output

In the raw mode characters are passed directly to the process without any special processing. In the cbreak mode characters are echoed and special start and stop characters are supported. In the **cooked** mode back space and line-kill characters are supported in addition. We shall study these modes in greater detail when we look at their implementation.

Control Functions

A process may invoke the following control functions:
turn on BREAK in transmitter
turn off BREAK
look ahead 1 character
set input mode to raw
set input mode to cooked
set input mode to cbreak
return number of input chars
turn on echo
turn off echo
set 'fill character'–the character to echo when input buffer is full

Device Driver Organization

Tty (and other device) driver routines can be divided into two sets: the **upper-half** and the **lower-half**. The upper half routines are called by the device-independent I/O routines such as read and write. They do not manipulate the device directly. Instead, they enqueue requests for transfer, and rely on routines in the lower-half to perform transfers later. The drivers use interrupt-driven processing to avoid the drawbacks of polling; therefore, the lower-half routines are the interrupt routines called by the input and output interrupt dispatchers.

The upper- and lower-half routines exchange data by writing to and reading from two circular buffers: the **input** and **output buffers**. These are needed because communication between the cpu and an SLU device is asynchronous. An input buffer contains characters received from the terminal by the input interrupt routine but not read by any upper-half routine. The output buffer contains data written by a upper-half routine but not transferred by the output interrupt routine.

Synchronization of the Upper and Lower Halves

Both the input and output buffers are instances of bounded buffers. For the input buffer, the (lower-half input) interrupt routine is the producer and the upper-half input routines such as *ttyread* and *ttygetc* are the consumers. For the output buffer, the upper-half output routines such as *ttywrite* and *ttyputc* are the producers and the (lower-half) output interrupt routine is the consumer.

Producers and consumers of bounded buffers need to synchronize their accesses so that a producer waits for a non-full buffer and a consumer waits for a non-empty buffer. But the lower-half interrupt routines cannot wait (recall the rules for interrupt processing). Therefore, only the higher-level input and output routines wait, on semaphores *isem* (input buffer non-empty) and *osem* (output-buffer non-full) respectively. The input interrupt routine signals the former when it inserts a character into the input buffer and the output routine signals the latter when it removes a character from the output buffer. Thus the count field of *isem* (initialized to 0) indicates the number of characters in the input buffer and the count field of *osem* (initialized to size of output buffer) keeps the number of empty positions in the output buffer.

The inability of an input interrupt routine to wait for a non-full input buffer leads to buffer overflow problems. In the Xinu implementation, the input interrupt routine simple discards a character received when the buffer is full.

The inability of the output interrupt routine to wait for the output buffer to be non-empty, however, does not cause any problems. When the output interrupt routine finds an empty buffer, it simply disables interrupts and goes into the idle state. Higher-half output routines are responsible for enabling interrupts when they deposit characters into the output buffer. Enabling an interrupt when the device is idle causes an interrupt to be posted and the output interrupt routine to be executed.

Watermarks

The above strategy of the signalling *osem* whenever a character is output is inefficient. A process can deposit characters into the buffer much faster than the SLU can remove it from the buffer. Thus, when one or more processes have large amounts of data to output, the buffer is almost always full, with the processes waiting for it to be non-full. Under these conditions, a signal to *osem* can cause a higher-priority process waiting for a non-full buffer to unblock and produce another character, make the buffer full again, and thus block again. Thus, each character output can cause an expensive reschedule! Even when processes are of the same priority, a printing process would use (character consumption rate/character production rate) of its time quantum before being blocked.

To alleviate this problem, the output interrupt routine uses a popular technique consisting of **watermarks**. The output routine runs in two modes: *normal* and *delayed*. When in the normal mode, it signals *osem* every time it outputs a character. It goes from normal to delayed when it finds the buffer full beyond the **high watermark**. Under the delayed mode it does not signal *osem* but keeps a count of the number of times it should have done so. It goes from delayed to normal mode when the buffer has drained to the **low watermark**. At this point the routine *msignal* is called to make up for the lost signals. This technique reduces the reschedules since no signals are generated during the delayed mode.

In Xinu the high and low watermarks are defined by the constant OBMINSP. The output interrupt routine goes into delayed mode when OBMINSP + 1 space is left. It goes from delayed to normal mode after OBMINSP characters are output in the delayed mode. (What are the high and low watermarks under this scheme?)

Echo Buffer

In addition to the input/output buffer, the terminal driver maintains an **echo buffer**, which contains a queue of characters to be echoed. This buffer is filled by the input interrupt routine and emptied by the output interrupt routine. (When deciding on the next character to output, should the output routine service the echo or output buffer?)

Data Structures

The terminal driver defines a **control block** for each terminal device connected to the cpu. These control blocks are kept in a **control block table** indexed by the minor number of the device. Each control block contains the following data:

The input and output buffers for the device, and the corresponding semaphores *isem* and *osem*.

An **echo buffer** which contains a queue of characters to be echoed. This buffer is filled by the input interrupt routine and emptied by the output interrupt routine.

Control Parameters, which are parameters set by the control functions and include information such as the mode of the terminal and whether characters should be echoed.

Terminal Parameters, which may vary from terminal to terminal, and include information such as whether the terminal allows erasing of characters, whether control characters are to be echoed as CTRL-X, and whether the line-kill command is to be supported.

User Parameters, which are parameters set by user commands such as the 'stop output' command.

Internal Parameters, miscellaneous information such as whether the output routine is in the normal or delayed mode and the number of characters in the current line. (see pg. 163-164 of text).

3.2.1 Upper-Half Tty Input Routines

The upper-half tty input routines consist of *ttygetc* and *ttyread* which are the consumers of characters filled into the input buffer.

Ttygetc(devptr)

The routine *ttygetc(devptr)* performs the following operations:

waits for *isem*

removes character at the tail of the input queue.

Ttyread(devptr, buff, count)

The routine *ttyread (devptr, buff, count)* may be used to read *count* number of chars from the input buffer or all the pending characters (what is the use of this operation?). It does the following:

If count = 0 then all the available characters are copied from the input buffer to *buff*, and the count of *isem* is adjusted.

If the count is not zero then as many characters as possible are copied from the buffer to *buf* (adjusting count of *isem*) The rest are obtained through *getc*.

3.2.2 Upper-Half Tty Output Routines

The upper half routines consist of *ttyputc* and *ttywrite*, which are the producers of data into the output buffer.

Ttyputc(devptr, ch)

The routine *ttyputc(devptr, ch)*, used to input a single character, does the following:

If the input character is NEWLINE then it recursively calls itself to print a RETURN before the NEWLINE *.

Waits for osem (free space)

Deposits character in output buffer.

'Signals' lower-half output routine by enabling output interrupts

Ttywrite(devptr, buff, count)

The routine *ttywrite(devptr, buff, count)*, used to output *count* number of characters, does the following:

Copies as many characters as it can in the space available, making sure that the count of *osem* is adjusted appropriately.

Uses ttyputc to output the rest.

3.2.3 Lower-Half Output Routines

The routine *ttyoin(iptr)*, which consumes characters from both the output buffer and the echo buffer, does the following:

If there is a character in the echo buffer, it removes it from the buffer and assigns it to the XBUF register and returns.

If the user has typed a 'stop output' command, determined by a user parameter, then the routine disables output interrupts and returns.

If there are no characters in the output buffer, it disables output interrupts and returns.

Otherwise, it removes the character from the tail of the output buffer and signals *osem* 0, 1, or OBMINSP times (what determines how many signals are generated?).

3.2.4 Lower-Half Input Processing

The input interrupt routine is far more complicated than the output interrupt routine because it has to implement the user interface functions. We shall study this routine by looking separately at the actions required to support the three modes: *raw*, *cbreak*, and *cooked*.

Raw Mode

In the raw mode, the terminal driver does not provide any user interface function, and passes the input characters directly to the process wihout any intermediate processing. The following steps are performed:

Extracts the character from RBUF.

If the input buffer is full, it throws away the character and returns.

If the input character has an error, puts the the error value 0200 in the buffer.

Otherwise inserts character into the buffer.
Signals the semaphore *isem*.

Cbreak Mode

Consider now the *cbreak* mode. The following steps distinguish it from the raw mode:

If the input character is a ‘normal’ char, then it is put not only in the input queue but also the echo queue.

If the input character is a RETURN, then it converts it into a NEWLINE before putting it into the queues.* [1]

If the input character is a stop or start output character, then an appropriate user parameter is changed, which is used by *ttyoin*. A start output character enables output interrupts.*.

Cooked Mode

In the *cooked* mode, the *erase* and *line-kill* commands are available to users. These commands allow a user to perform line editing. Characters are made available to a program only when the user types a RETURN or NEWLINE. Thus a user may edit a line of characters before they are consumed by a program. We now look at the steps taken in this mode to support these commands:

A cursor is maintained to keep the count of the number of characters in the current line.

If the user inputs the *erase* character then the head of input buffer and the cursor are decremented and the last character echoed is erased.*.

If the user inputs the *line-kill* character then the head of the input buffer is adjusted and the cursor reset. Also the RETURN-NEWLINE combination is put in the echo queue.*.

If the user input the NEWLINE or RETURN character then the character is echoed, it is put in the input buffer, *isem* signalled (cursor + 1) times, and the line cursor set to 0.

If the user inputs a regular character, then the line cursor is incremented, and the character is put in the input buffer. Semaphore *isem* is not signalled.

Echoing Characters

The following steps are taken to put a character in an echo queue:

If the characters is either a NEWLINE or a RETURN the RETURN-NEWLINE combination is inserted in the echo queue.*

If the character is a control character (ascii value less than ascii of BLANK) or the *del* character, then a printable representation of 2 characters (^followed by (char + 0100)) is stored.

Any other character is put directly into the echo queue.

Erasing Characters

Erasing a character is simple. Normally a backspace character is echoed. However if a terminal parameter says ‘erasing backspace’ is supported, then, in addition, a blank followed by a back space is echoed.

[1]* This action occurs only if an appropriate terminal parameter is set in the control block.

Care has to be taken to erase over the two character representation of control characters.

3.2.5 Miscellaneous

Device Driver Control

The implementation of the *control (devptr, func, addr)* operation is simple. Most functions simply change the appropriate control parameter. The only non-trivial function is the TCNEXTC function, which looks ahead 1 character. It waits on *isem*, reads the character at the tail of the input buffer without changing the tail pointer, and signals *isem* since the character was not actually consumed.

Tty Control Block Initialization

A procedure *ttyinit* is called to do device-specific initialization of the dispatch table entry, device registers, and control block of each SLU device.

It manipulates the dispatch table entry as follows:

For efficiency reasons, it changes the input and output code of in the dispatch table from the minor number to a pointer to the control block.

It initializes the control block as follows:

Sets the terminal parameters.

Initializes control parameters.

Initializes other parameters.

It sets the device registers as follows:

Clears RBUF so that a new character can be received in it.

Enables input interrupts.

Disables output interrupts.