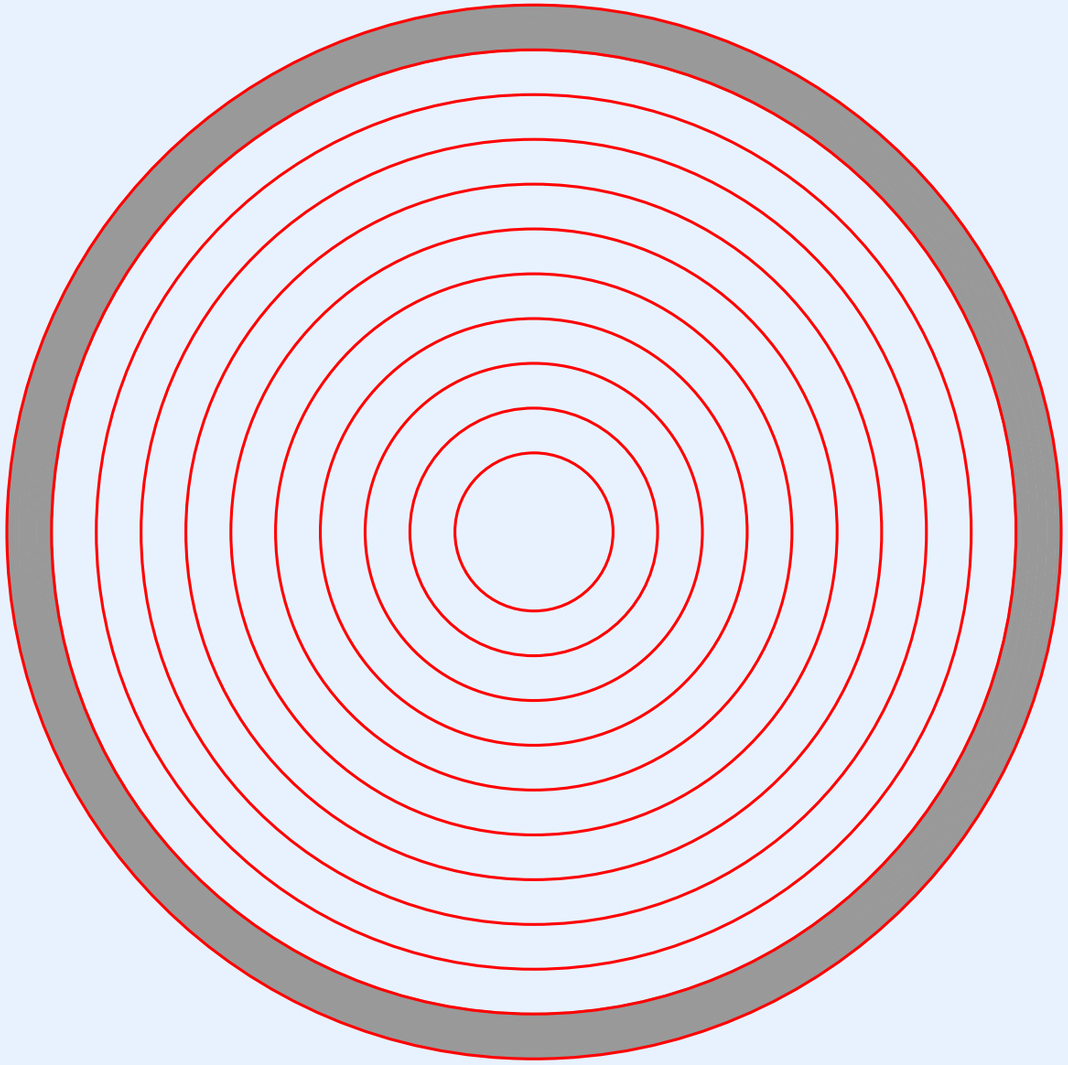
Module XIV

User Interface

Location Of The User Interface In The Hierarchy



The Two Operating System Interfaces

- Operating systems provide two ways to access services
- An interface for applications
  - Generically called an API (Application Program Interface)
  - Consists of a set of *system calls*
- An interface for human users
  - Usually interactive
  - Can be a Command Line Interface (CLI) or Graphical User Interface (GUI)
  - Gives the system a “personality”

Characteristics Of User Interfaces

- GUI
  - Allows users to launch applications
  - May include *copy-and-paste* and *drag-and-drop* mechanisms
  - Relies on applications to handle most tasks
- CLI
  - Parses textual commands
  - Arguments on commands allow the user to specify an arbitrary level of detail

Command Interpreter Implementations

- Two implementations of command interpreters have been used
- The command interpreter is integrated into the operaing system
- The command interpreter consists of an application that is separate from the operating system

Command Interpreter Built Into OS

- Typical of early systems and some small, embedded systems
- Advantages
  - The interpreter understands command syntax and semantics
  - Can offer command completion capability
  - Can prompt for arguments and check arguments for correctness
  - Can warn user about meaningless combinations
- Disadvantages
  - A user is limited to exactly the commands the OS provides
  - A user cannot select a non-standard command interpreter
  - Difficult to add commands (requires changing the OS)

Command Interpreter Implemented By An Application

- Introduced by MULTICS; popularized by Unix
- The interpreter only handles basic command syntax
- Individual commands must check and interpret arguments
- Advantage
  - Each user can choose their own interpreter
  - New commands can be added at any time
- Disadvantages
  - Nonuniformity among commands and arguments
  - No built-in semantic checks (argument errors are reported after a command starts running)

Example Of A Separate Interpreter: The Unix Shell

- Runs as a standard application (no special privilege is required)
- Provides per-line processing
- Interprets each line as a command
- Uses the same syntax for scripts as for interactive input
- Offers basic programming language constructs
  - Variables
  - Sequence of statements
  - Definite and indefinite iteration
  - Conditional execution

Shell Variables

- Have you used shell variables?
- Do you really understand how they work?
- The basics (from Korn shell)

```
X=hello
echo $X
```

  produces a line of output containing the word *hello*

- Given the above, the command

```
gcc $X.c
```

  compiles file hello.c

Shell Binding Times

- Now consider a more complex example
- Suppose the current directory contains files

```
aaa  bbb  ccc
```

- What do the following lines mean if typed into a shell?

```
BEES=b*      # Define variable BEES
ls -l $BEES  # This will list file bbb
touch bb     # Add another file that starts with b
ls -l $BEES  # Will this line list just the file
             #   named bbb, or both bb and bbb?
```

Another Example Of Binding Times

- The shell
  - Has both local a global variables
  - Uses the term *environment* for the set of global variables
- Environment variables
  - Imports variable definitions from the user's environment
  - Allows a user to *export* specific variables to the environment
  - The environment is passed to each child process that the shell executes
- Note: programs such as *make* allow environment variables to be accessed

Environment Variable Binding Times

- Suppose a user defines an environment variable QQQ

```
QQQ=CS354    # Define variable QQQ
export QQQ   # Export QQQ to the environment
myscript     # Run a shell script as a command
echo $QQQ    # Print the value of QQQ
```

- What will the output be if *myscript* contains the following lines?

```
echo $QQQ    # Print the current value of QQQ
QQQ=CS503    # Redefine QQQ
```

Environment Variable Binding Times
(continued)

- The answer
  - A copy of the environment is kept for each process
  - A process inherits a copy from its parent when the process starts
  - Changes only affect the local copy (and processes that are created)
- In the example, the output is

```
CS354
CS354
```

The Basic Unix Shell Evaluation Algorithm

A shell repeats the following steps:

A. Read and parse the next command, dividing it into tokens

B. Perform variable assignment (*var = string*)

C. Perform macro substitution: replace *$X* with value of string *X*

D. Perform file name matching (e.g., eliminate “*”)

E. Search the user’s *PATH* for the command named by first token

F. Invoke the command, passing remaining tokens as arguments

Data Types In The Unix Shell Language

- The shell supports one data type: string
- Builtin commands handle
  - Iteration (*while* and *for*)
  - Conditional execution (*if-then-else*)
- Quotes prevent substitution (delay binding)
  - Single quotes inhibit interpretation within the string
  - Double quotes allow variable substitution within the string
- Each command is executed by a separate process
- A command *pipeline* connects the output from one process to the input of another

Unix Shell Parsing In Practice

- The shell acts like a compiler
- Compound statements (*while*, *for*, *if-then-else* can span multiple lines of input
- A long pipeline can span multiple lines as well
- The shell must also handle file redirection
- Consequence: a shell must checks for balanced delimiters (e.g., if $\rightarrow$ fi)

Unix Shell Data Conversions

- Output from a command can be converted to a string

  `` `command` ``

- The contents of file can be assigned to a string

  `` `cat file` ``

- The contents of a variable can be converted to command input

  ```
  echo $string | command
  ```

- Literal text can be converted to command input

  ```
  command <<!
    ...literal text goes here
  !
  ```

The Unix Shell: Paths And Command Invocation

- The shell maintains “search path”
  - The path specifies a list of directories
  - The shell uses the path during command lookup
- To find a file to execute, the shell searches the path one directory at a time
  - It prepends the next directory to the command name
  - It checks to see if the result is a file
  - It stops if the file exists
- Once a file has been found, the shell checks to see that the file is executable
- The current directory (denoted “.”) works like any other directory name in a path

Efficiency And Path Binding

- There are two possibilities: late or early binding
- Late binding (used in the original shell)
  - The shell searches the path each time a user enters a command
  - Is inefficient
- Early binding (introduced in BSD Unix)
  - The shell presearches the path and caches the names of files in each directory
  - Is more efficient
  - Cannot detect changes in directory contents automatically
  - A user enters a *rehash* command to perform a new binding
  - The shell can automatically invoke *rehash* and retry if a command is not found

Efficiency And Path Binding
(continued)

- Does automatic rehash work?
- Answer: it depends
- Consider a case where
  - A user adds an executable program to a directory on the path
  - The program has a unique name (not found in any other directory along the path)
- Now consider a case where
  - A user adds an executable program to the first directory on the path
  - An executable program with the same name appears in a later directory on the path

I/O Redirection

- A user can redirect input or output
- The shell provides separate redirection for
  - Standard input
  - Standard output
  - Standard error
- Syntax is
  - Output: > file
  - Input: < file
- The syntax for standard error redirection depends on the shell

Synchronization Of Processes

- Background processing
  - The shell always creates a process to execute a command; background execution allows the shell to continue processing concurrently
  - The syntax is "**&**"
- Pipeline
  - The output from one process is fed to the input of another
  - An arbitrary pipeline is allowed
  - The "pipe" between two processes consists of a finite buffer
  - The operating system handles process synchronization

Shell Script

- The name given to a file that contains a set of shell commands
- The file must be executable
- A shell script uses the same syntax as an interactive shell (earlier operating systems used a special syntax for scripts)
- BSD Unix introduced the use of a two-byte *magic number* consisting of the ASCII characters #!
- If a file name follows the magic number, the file is taken to be the program to run with the script as input

#!/users/me/bin/my_program

Shell Input

- One can invoke a shell script, *X*, by:

  ksh < X

  or by naming X as an argument to the shell:

  ksh X

- Challenge: create a shell script that behaves differently when invoked in the two ways shown above
- Note: feel free to use whatever shell you prefer (e.g., *bash*)

Design And Implementation Of An Example Shell

The Xinu Shell

- We will use the Xinu shell to illustrates the basics
- The Xinu shell
  - Is not fancy
  - Has a fixed set of commands compiled into shell itself
  - Uses a familiar command syntax:

    command_name arg* [redirection] [background]

    where *redirection* includes both input (< file) and output (> file), and *background* (&) allows a user to run a command in background
  - The notation arg* means “zero or more args”, and [x] means “x is optional”
  - The syntax and basic approach is similar to Unix

Lexical Tokens In The Xinu Shell

Token Type (num. value)	Character	Description
SH_TOK_AMPER (0)	&	ampersand
SH_TOK_LESS (1)	<	less-than symbol
SH_TOK_GREATER (2)	>	greater-than symbol
SH_TOK_OTHER (3)	'...'	quoted string (single quotes)
SH_TOK_OTHER (3)	"..."	quoted string (double-quotes)
SH_TOK_OTHER (3)	other	sequence of non-whitespace

- Only six lexical tokens are needed
- A string that starts with one type of quote can contain the other type of quote

```
"Don't blink!"
```

Organization Of Shell Software

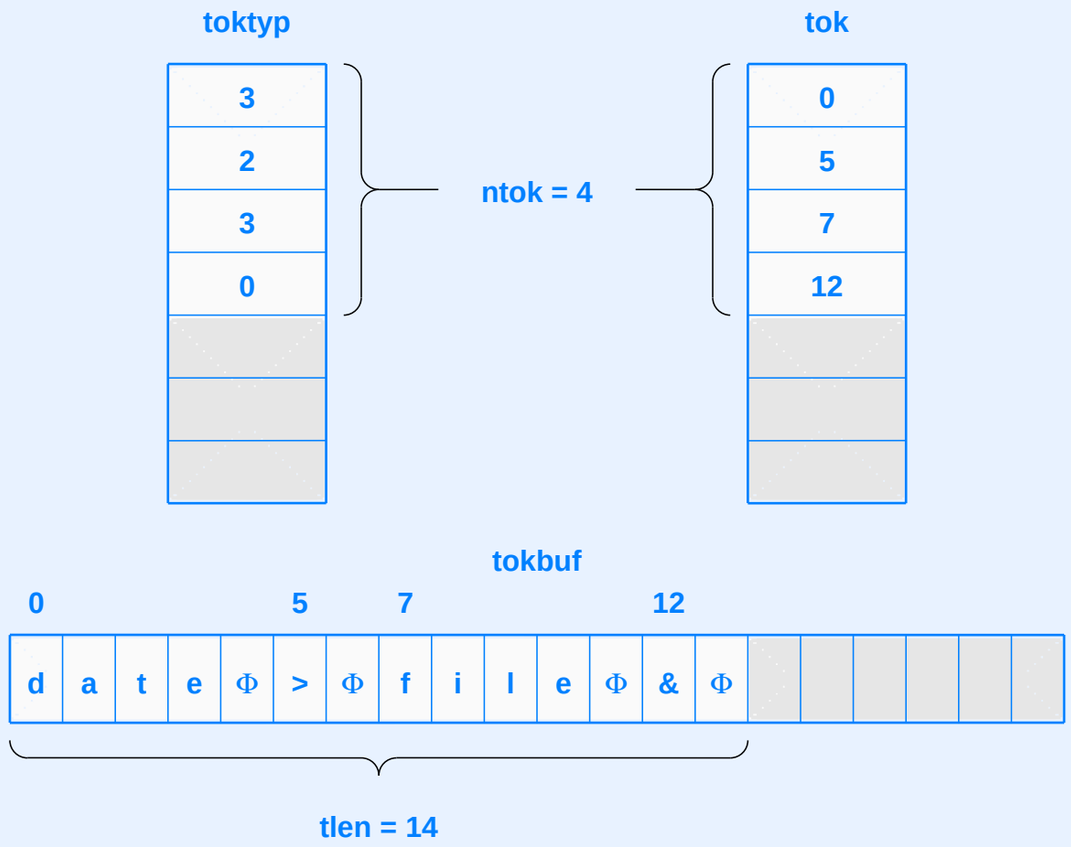
- The shell is organized like a compiler (interpreter)
- A *lexical analyzer*
  - Divides an input line into a series of tokens
  - Stores each token (the characters that make up the token) along with the type
- A *parser*
  - checks to ensure the sequence of tokens is valid
  - Turns the tokens into a command with arguments
  - executes the command

Lexical Analysis

- Because the Xinu shell only handles one line at a time, the shell
  - Reads an entire line
  - Calls a lexical analyzer to divide the line into tokens
- The lexical analyzer
  - Eliminates whitespace (i.e., blanks and tabs)
  - Checks for invalid tokens (e.g., file<)
  - Returns the number of tokens found

Token Storage

- Given the input line: `date > file &`
- The shell stores the tokens by storing an index and a type in two arrays

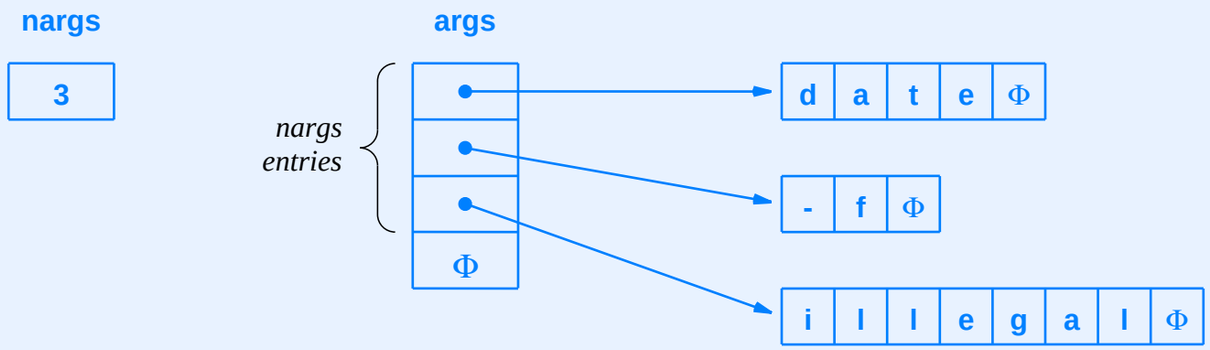


Parsing And Execution

- The first token must be the name of a command
- The final tokens may specify
  - Background processing
  - I/O redirection (input or output)
- The remaining tokens are taken to be arguments to the command

Arguments Passed To A Command

- Like Unix, Xinu only passes two arguments to a command
  - An integer count (nargs)
  - An array of pointers to argument strings
- Also like Unix, the first argument is the command name
- An example of arguments for input line `date -f illegal`



Builtin Commands

- Xinu allows a command to be *builtin*, which means the command is executed by the shell itself without creating a child process
- However
  - Builtin commands may not be run in background and may not have I/O redirection
- Examples
  - Exit (causes the shell to exit)
  - Kill (kill a process)
  - Clear (send the “screen clear” sequence to standard output

Executing A Non-Builtin Command

- The shell must
  - Create a process to run the command
  - Construct an argument list at pass it to the command
  - Redirect input and/or output, if specified
  - Resume the process
- If the user did not specify background processing, the shell waits for the command to complete
- Note: when a process exits, Xinu sends a message to the parent with value equal to the process ID of the exiting process

A Few Shell Declarations (Part 1)

```
/* excerpt from shell.h */

/* Size constants */

#define SHELL_BUFLEN    TY_IBUFLEN+1    /* Length of input buffer       */
#define SHELL_MAXTOK    32              /* Maximum tokens per line      */
#define SHELL_CMDSTK    8192            /* Size of stack for process    */
                                        /*    that executes command     */
#define SHELL_ARGLEN    (SHELL_BUFLEN+SHELL_MAXTOK) /* Argument area    */
#define SHELL_CMDPRIO   20              /* Process priority for command */

/* Constants used for lexical analysis */

#define SH_NEWLINE      '\n'            /* New line character           */
#define SH_EOF          '\04'           /* Control-D is EOF             */
#define SH_AMPER        '&'             /* Ampersand character          */
#define SH_BLANK        ' '             /* Blank character              */
#define SH_TAB          '\t'            /* Tab character                */
#define SH_SQUOTE       '\''           /* Single quote character       */
#define SH_DQUOTE       '"'             /* Double quote character       */
#define SH_LESS         '<'             /* Less-than character  */
#define SH_GREATER      '>'             /* Greater-than character       */
```

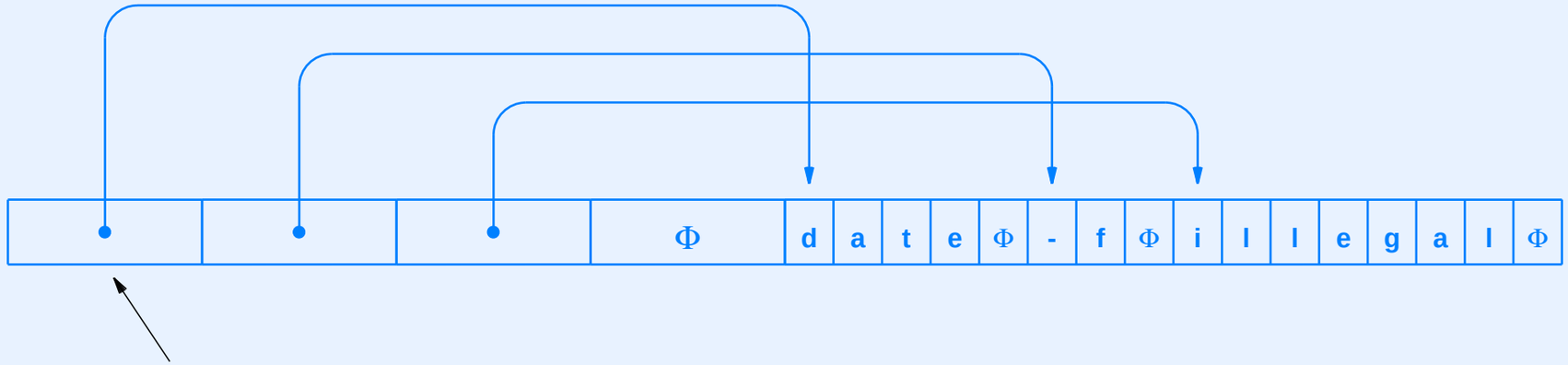
A Few Shell Declarations (Part 2)

```
/* Token types */

#define SH_TOK_AMPER    0               /* Ampersand token              */
#define SH_TOK_LESS     1               /* Less-than token              */
#define SH_TOK_GREATER  2               /* Greater-than token           */
#define SH_TOK_OTHER    3               /* Token other than those       */
                                        /*   listed above (e.g., an     */
                                        /*   alphanumeric string)       */

/* Shell return constants */

#define SHELL_OK         0
#define SHELL_ERROR      1
#define SHELL_EXIT      -3

/* Structure of an entry in the table of shell commands */

struct  cmdent  {                       /* Entry in command table       */
        char    *cname;                 /* Name of command              */
        bool8   cbuiltin;               /* Is this a builtin command?   */
        int32   (*cfunc)(int32,char*[]);/* Function for command         */
};

extern  uint32  ncmd;
extern  const   struct  cmdent  cmdtab[];
```

Storage For Arguments

- Background processing means
  - Multiple commands can execute concurrently
  - Each command can have a different set of arguments
  - Arguments must be stored with the command process, not with the shell
- A Xinu trick: when creating a process to run a command, insert the arguments into bottom of the command's process stack
  - Store the *args* array plus the actual arguments
  - The arguments will be released along with the stack when the process exits
  - A single contiguous array is used
  - Added advantage: if a process accidentally overwrites its arguments, no other process will be affected

The Xinu Argument Layout At Runtime



args array starts at lowest byte of user's stack or next multiple of 4 bytes beyond it

- The shell
  - Creates a process to run a command, but does not resume the process
  - Computes the size needed for arguments, and copies them into the bottom of the process's stack
  - Resumes the process

For The Next Class

- Read Chapter 26 on the Xinu shell
- Look at
  - The lexical analysis function, *lexan.c*
  - The parser, *shell.c*
  - The function to handle arguments, *addargs.c*
- Come back with questions

Mice And Windows

A Mouse

- A pointing device invented as a companion to the bit-mapped display
- Operates as an I/O device
- The hardware
  - Detects movement
  - Reports motion in 2-dimensions
- The hardware interface is surprising

Mouse Hardware

- Contains
  - Two motion detectors
    * Arranged at right angles
    * Labeled *X* and *Y*
  - Two A-to-D converters
  - *1* to *3* buttons
  - Thumb wheels or touch controls
- Communicates with a computer using a character-oriented serial interface (the same interface as a keyboard)

Mouse Hardware
(continued)

- Mouse hardware is unexpectedly sophisticated
- It uses two-way communication to
  - Accept commands from the computer
  - Respond to queries from the computer
  - Transmit data to computer asynchronously

Mouse Communication And Interaction

- Two modes are used
  - Polling mode
  - Streaming mode

Polling Mode

- A processor sends a request for information
- The mouse responds by reporting
  - Motion since last request
  - The status of the buttons
- Seldom used except after communication has been temporarily lost

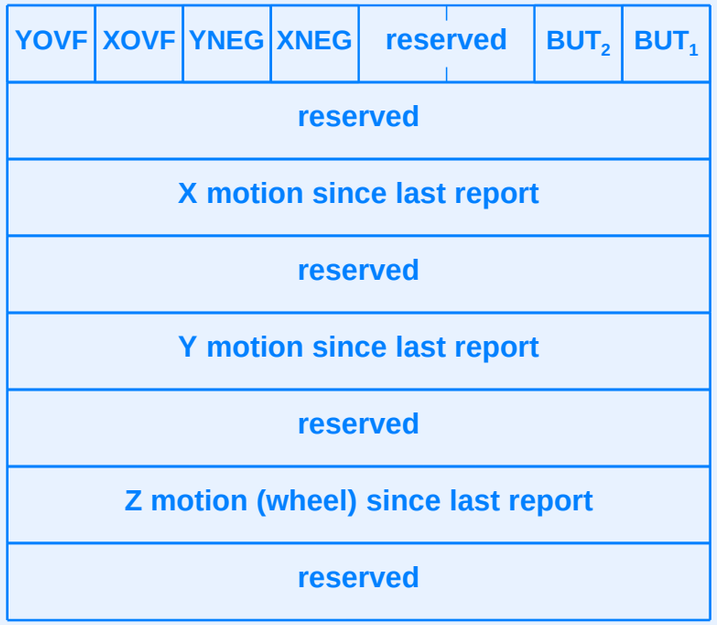
Streaming Mode

- A processor
  - Specifies the resolution and scaling to be used for motion detection
  - Sends one request to start the stream
- The mouse transmits new information
  - When movement exceeds a predetermined threshold
  - When a button is depressed or released
  - When thumbwheel moves

The Communications Interface

- A mouse uses a conventional serial line (or serial transmission over USB)
  - Full duplex
  - 8-bit characters
  - The RS232 standard can be used, just like a keyboard
- Each report sent by a mouse
  - Requires multiple characters
  - Follows a fixed format
  - The format is known as a *mouse packet*

Example Mouse Packet (2-Button Mouse)



- Reserved bits must be zero
- *XOVF* and *YOVF* report overflow
- *XNEG* and *YNEG* are sign bits
- BUT_1 and BUT_2 give button status

Typical Mouse Parameters

- Samples per second
  - Selectable from 10, 20, 30,... 200 samples per second
  - Standard is 100 samples per second
- Tracking
  - Linear (e.g., 2:1)
  - Non-linear (e.g., exponential)
- Resolution
  - Example: 4 counts per millimeter
- The point: a seemingly simple device, a mouse, is quite complex

Display Screen

- A screen is divided into pixels
- Most screens are *memory mapped*, which means the operating system writes values to a display memory to change pixels on the screen
- Early screens were black-and-white
  - One bit in the memory per pixel
  - They were known as *bit-mapped* screens
- Current screens display color
  - One or more bytes per pixel
  - They are sometimes called *byte-mapped* screens

Color Specification

- The hardware uses three phosphors: red, green, blue
- All colors result from a combination of 0% to 100% of each of the three
  - The colors add in interesting ways
  - Example 100% red + 100% green + 0% blue gives yellow
- In practice
  - An integer value is used for each color instead of a percentage
  - Typically, the value for each color ranges from 0 to 255

Windowing Systems

- Windows are an operating system abstraction handled by software
- Most window systems allow windows to be
  - Created/destroyed at any time
  - Moved/resized/iconified
- A typical implementation
  - Each window is a rectangular region on screen
  - The window must be created before being used
  - The OS presents an application with a separate coordinate space where (0,0) is the corner of the window

Window Display Parameters

- Many details are involved
  - Background/foreground colors
  - A title for the window
  - The location of scroll bars
  - Borders
- The details are controlled by a piece of software known as a *window manager*
- In Unix systems, each user can choose their own window manager
- Allowing users to choose a window manager means each user can see windows displayed in their preferred style

Cursor Movement

- The goal
  - A cursor should appear on the screen at all times
  - The cursor on the screen should track the mouse/touchpad movement
- Unfortunately
  - Mouse hardware is not directly linked to video hardware
- Consequence: software must
  - Update the cursor when the mouse moves
  - Map the new position to the correct window

The Cursor Update Algorithm

- When the cursor moves
  - Undo the cursor at old position by repainting the original display values
  - Determine the new position for the cursor, *P*
  - Save video memory at position *P* for a later “undo” operation
  - Use cursor color to paint cursor at position *P*

Optimizing Cursor Update

- A cursor update is required at each mouse interrupt
- Switching context to a window manager process introduces significant delay
- To reduce the overhead and avoid delay, perform the cursor update in the device driver during interrupt processing
- The downside: if the processor becomes overloaded, interrupt processing can be delayed or missed, which means the cursor on the screen may lag mouse movements or some mouse movements may be missed
- Good news: multicore processors make lost interrupts unlikely

Summary

- Operating system support two styles of user interface
  - Command line interpreter
  - Graphical
- The Unix shell
  - Runs as a separate application
  - Provides a miniature programming language
  - Supports concurrency and data pipelining
  - Limits variables to strings
  - Uses quotes to delay binding
  - Provides conversions between strings and command input/output

Summary
(continued)

- A mouse
  - Is surprisingly complex
  - Uses a two-way communication system and delivers “packets”
- Display Screen
  - Uses a memory-mapped approach
  - The hardware for a pixel uses three primary colors: red, green, and blue

Summary
(continued)

- Windowing systems
  - Are a software abstraction
  - Window manager handles details
- Cursor update
  - Is performed by software
  - Is often handled at interrupt time

Questions?