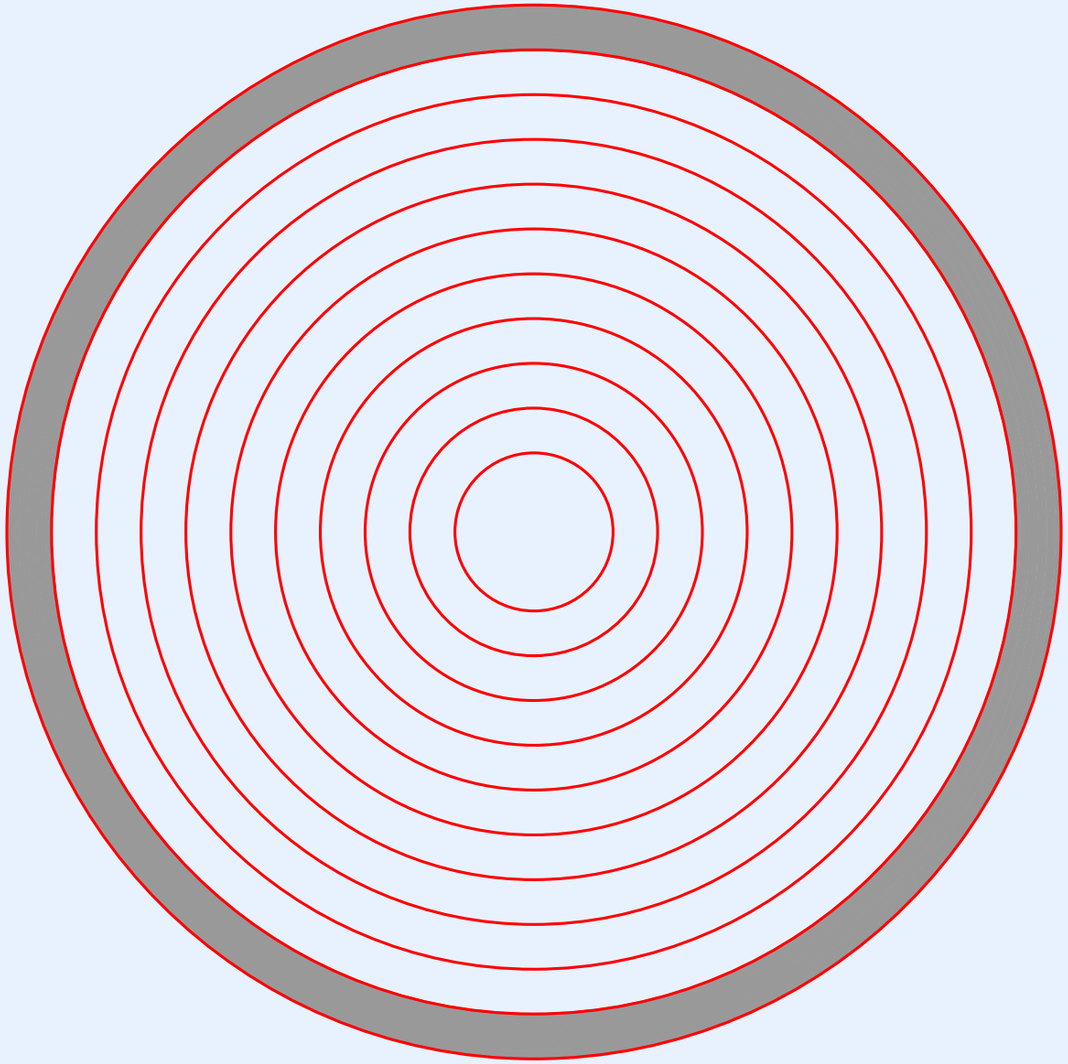
Module XXVII

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*
  - We have seen examples
- 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
- Command Line Interface
  - Makes the file system visible
  - Parses textual commands
  - Arguments passed to commands can allow the user to specify an arbitrary level of detail

A Command Interpreter (CLI)

- Software that accepts commands entered by users and performs the specified action
- Two implementations of command interpreters have been used
  - Early systems and some small embedded systems: the command interpreter is integrated into the operating system
  - Multics/Unix and later systems: the command interpreter consists of an application that is separate from the operating system

A Command Interpreter Built Into OS

- Advantage: because the interpreter understands command syntax and semantics, it can
  - Offer command completion capability
  - Prompt for required arguments
  - Check arguments for correctness
  - Warn users about meaningless or dangerous requests
- Disadvantages
  - A user is limited to exactly the commands the OS provides
  - A user cannot select a non-standard command interpreter
  - Adding new commands is difficult and requires recompiling the OS

Command Interpreter Implemented By An Application

- Introduced by MULTICS; popularized by Unix
- The interpreter only handles basic command syntax
- Individual programs must check and interpret arguments
- Advantages
  - 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 process (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 and global variables
  - Uses the term *environment* for the set of global variables
- Environment variables
  - Import variable definitions from the user's environment
  - Allow 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
export QQQ   # Export QQQ to the environment
```

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 macro substitution: replace *$X* with value of string *X*

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

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

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 check 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

Late And Early Path Binding

- Two forms of path binding have been used: late and early
- Late binding
  - Was used in the original Borne shell
  - The shell searches directories along the path each time a user enters a command
- Early binding (introduced in BSD Unix's C shell)
  - When started, the shell searches the path and caches the names of files in each directory
  - When a user enters a command, the shell searches the cache to find where the command resides

Path Binding And Efficiency

- Late binding
  - Guarantees that the shell will find a new command immediately after the command is added to a directory on the path
  - Is somewhat inefficient because it reads each directory on the path each time a user enters a command
- Early binding
  - Is more efficient because it avoids searching directories along the path each time a command is entered
  - Cannot detect new commands added to directories or other changes in directory contents automatically
  - May require a user to enter a *rehash* command to recreate the cache

Automatic Rehash

- A technique that allows a shell to find changes in directories
- The idea: when a user enters a command and the command is not found in the cache
  - The shell does not immediately report the problem to the user
  - Instead, the shell automatically triggers *rehash* to recreate the cache, and retries the command lookup in the refreshed cache
  - If the command is found after the rehash, the shell runs the command
  - If the command is not found after the rehash, the shell reports “command not found” to the user
- Note that automatic rehash makes the path binding somewhat later, but not as late as the original shell

A Question About Automatic Rehash

- Question: does automatic rehash work?
- Answer: it depends.
- Case #1:
  - A user adds an executable program, *x*, to a directory on the path, and no other directory on the path contains a file named *x*
  - The user tries to run command *x*
  - The shell does not find *x* in its cache, so the shell invokes *rehash*
  - After *rehash* runs, *x* appears in the cache
  - Result: automatic rehash works: the shell finds program *x* and runs it

A Question About Automatic Rehash (continued)

- Case #2:
  - A user adds an executable program, *y*, to the first directory on the path, but a program named *y* had previously appeared in another directory along the path
  - What happens?

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 And A Challenge

- 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 Principles For A Command Interface

- Functionality: sufficient for all needs
- Orthogonality: only one way to perform a given task
- Consistency: commands follow a consistent pattern
- Least astonishment: a user should be able to predict results

Design And Implementation Of An Example Shell

The Xinu Shell

- Is not fancy — it merely illustrates the basics
- Has a fixed set of commands compiled into shell itself
- Interprets each line to be a pipeline of one or more commands

  command [| command]*

- Each command follows a familiar command syntax:

  command_name arg*

- The first command may specify input redirection (<file), and the last command may specify output redirection (>file) and background execution (&)
- The notation X* means “zero or more of X”, and [X] means “X is optional”

A Warning About Xinu Shell

- The Xinu shell is written ad hoc
- The data structures and algorithms are unusual
- The idea is to
  - Show a minimal implementation
  - Illustrate shell organization without unnecessary complexity

Lexical Tokens In The Xinu Shell

Symbolic Name (Token Type)	Numeric Value	Input Characters	Description
SH_TOK_AMPER	0	&	ampersand
SH_TOK_LESS	1	<	less-than symbol
SH_TOK_GREATER	2	>	greater-than symbol
SH_TOK_PIPE	3	|	pipe symbol
SH_TOK_OTHER	4	'...'	quoted string (single-quotes)
SH_TOK_OTHER	4	"..."	quoted string (double-quotes)
SH_TOK_OTHER	4	other	sequence of non-whitespace

- Only seven lexical tokens are needed and only five types
- A string that starts with one type of quote can contain the other type of quote

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

The Overall Syntax

pipeline → **command [i_redirect] [commands] [o_redirect] [backgnd]**

command → **name [args]**

commands → **SH_TOK_PIPE command [commands]**

name → **SH_TOK_OTHER**

args → **SH_TOK_OTHER [args]**

i_redirect → **SH_TOK_LESS SH_TOK_OTHER**

o_redirect → **SH_TOK_GREATER SH_TOK_OTHER**

backgnd → **SH_TOK_AMPER**

Organization Of Xinu Shell

- The shell is organized like an on-line compiler (i.e., an interpreter) with two main parts
- 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

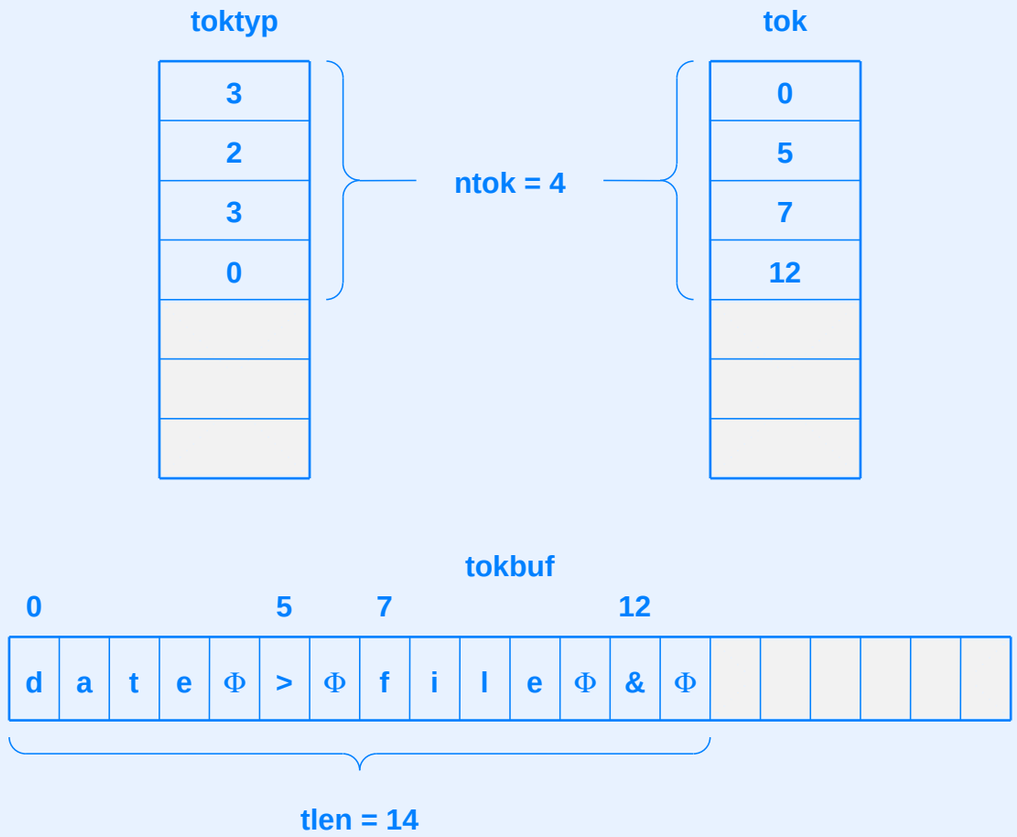
- Because the Xinu shell only handles one line at a time, the shell
  - Reads an entire line
  - Calls a lexical analyzer to divide the line into tokens
- The lexical analyzer
  - Eliminates whitespace (i.e., blanks and tabs)
  - Checks for invalid tokens (e.g., <<file>>)
  - Returns the number of tokens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

Token Storage

- Is unusual
- Uses two *parallel arrays* plus an array of characters (*tokbuf*)
- Each array has *ntok* entries for an input line with *ntok* tokens
- One array (*toktyp*) tells the type of the token
- Another array (*tok*) gives the index in tokbuf where the token begins
- Note: each token in tokbuf ends with the null character

Token Storage Example

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



Approach To Parsing

- Recall: the lexical analyzer divides an entire input line into tokens before the parser runs
- The parser uses occurrences of the pipe symbol to divide the line into *segments* that each consist of a command with arguments, and then handles each segment independently
- The next slides give more details

Steps The Parser Takes (Part 1)

loop forever {

1. Read the next input line and call lexan to divide it into tokens.
2. Divide tokens into segments separated by the pipe symbol, and set each segment's input and output devices to the device the shell is using (usually CONSOLE).
3. Check for background (i.e., & as the last token) and remove the token.
4. Check for input redirection on the first segment and output redirection on the last segment, and remember the names of the input and output files, if any.
5. Look up the command name in each segment.

Steps The Parser Takes (Part 2)

6. Open the redirected input and output files, if any, and record the device ID as the first segment input device or the last segment output device.
7. Create N–1 pipes for N segments, and record the device ID as the output device for segment i and the input device for segment i+1.
8. Create a process to run each segment and add arguments.
9. If running in foreground, wait for the last process to exit (i.e., wait for a message to arrive with the process ID of the last process in the pipeline).

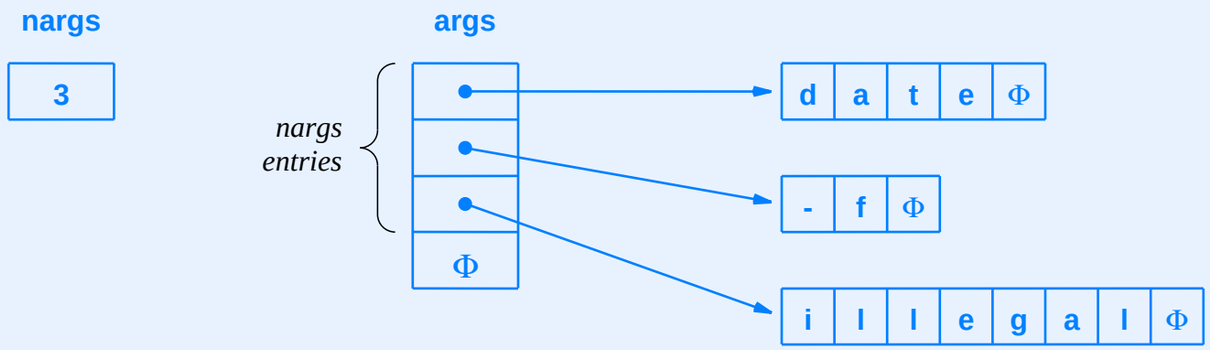
}

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 Command Arguments

- Consider the input line `date -f illegal`
- When a process runs the data command, the process receives two arguments
  - An integer count
  - An array of pointers to strings
- Illustration of the arguments



Xinu Command Execution

- Each command is executed by a separate child process (i.e., the shell creates a process to run the command)
- Input and output
  - The first process in a pipeline can have input redirected to a file; the last can have output redirected to a file
  - Processes in the middle of the pipeline receive input from a pipe and send output to a pipe

Implementation Of
The Xinu Lexical Analyzer

Lexan.c (Part 1)

```
/* lexan.c  - lexan */

#include <xinu.h>

/*------------------------------------------------------------------------
 * lexan  -  Ad hoc lexical analyzer to divide command line into tokens
 *------------------------------------------------------------------------
 */

int32   lexan (
          char          *line,          /* Input line terminated with   */
                                        /*   NEWLINE or NULLCH          */
          int32         len,            /* Length of the input line,    */
                                        /*   including NEWLINE          */
          char          *tokbuf,        /* Buffer into which tokens are */
                                        /*   stored with a null         */
                                        /*   following each token       */
          int32         tok[],          /* Array of pointers to the     */
                                        /*   start of each token        */
          int32         toktyp[]        /* Array that gives the type    */
                                        /*   of each token              */
        )
{
```

Lexan.c (Part 2)

```
char    quote;                  /* Character for quoted string  */
uint32  ntok;                   /* Number of tokens found       */
char    *p;                     /* Pointer that walks along the */
                                /*   input line                 */
int32   tbindex;                /* Index into tokbuf            */
char    ch;                     /* Next char from input line    */

/* Start at the beginning of the line with no tokens */

ntok = 0;
p = line;
tbindex = 0;

/* While not yet at end of line, get next token */

while ( (*p != NULLCH) && (*p != SH_NEWLINE) ) {

        /* If too many tokens, return error */

        if (ntok >= SHELL_MAXTOK) {
                return SYSERR;
        }

        /* Skip white space before token */

        while ( (*p == SH_BLANK) || (*p == SH_TAB) ) {
                p++;
        }
```

Lexan.c (Part 3

```
/* Stop parsing at end of line (or end of string) */

ch = *p;
if ( (ch==SH_NEWLINE) || (ch==NULLCH) ) {
        return ntok;
}

/* Set next entry in tok array to be an index to the    */
/*   current location in the token buffer               */

tok[ntok] = tbindex;    /* The start of the token       */

/* Set the token type */

switch (ch) {

    case SH_AMPER:      toktyp[ntok] = SH_TOK_AMPER;
                        tokbuf[tbindex++] = ch;
                        tokbuf[tbindex++] = NULLCH;
                        ntok++;
                        p++;
                        continue;
```

Lexan.c (Part 4)

```
        case SH_PIPE:       toktyp[ntok] = SH_TOK_PIPE;
                            tokbuf[tbindex++] = ch;
                            tokbuf[tbindex++] = NULLCH;
                            ntok++;
                            p++;
                            continue;

        case SH_LESS:       toktyp[ntok] = SH_TOK_LESS;
                            tokbuf[tbindex++] = ch;
                            tokbuf[tbindex++] = NULLCH;
                            ntok++;
                            p++;
                            continue;

        case SH_GREATER:    toktyp[ntok] = SH_TOK_GREATER;
                            tokbuf[tbindex++] = ch;
                            tokbuf[tbindex++] = NULLCH;
                            ntok++;
                            p++;
                            continue;

        default:            toktyp[ntok] = SH_TOK_OTHER;
    };
```

Lexan.c (Part 5)

```
/* Handle quoted string (single or double quote) */

if ( (ch==SH_SQUOTE) || (ch==SH_DQUOTE) ) {
        quote = ch;     /* Remember opening quote */

        /* Copy quoted string to arg area */

        p++;    /* Move past starting quote */

        while ( ((ch=*p++) != quote) && (ch != SH_NEWLINE)
                        && (ch != NULLCH) ) {
                tokbuf[tbindex++] = ch;
        }
        if (ch != quote) {  /* String missing end quote */
                return SYSERR;
        }

        /* Finished string - count token and go on      */

        tokbuf[tbindex++] = NULLCH; /* Terminate token  */
        ntok++;         /* Count string as one token    */
        continue;       /* Go to next token             */
}
```

Lexan.c (Part 6)

```
                /* Handle a token other than a quoted string            */

                tokbuf[tbindex++] = ch; /* Put first character in buffer*/
                p++;

                while ( ((ch = *p) != SH_NEWLINE) && (ch != NULLCH)
                        && (ch != SH_LESS)  && (ch != SH_GREATER)
                        && (ch != SH_BLANK) && (ch != SH_TAB)
                        && (ch != SH_AMPER) && (ch != SH_SQUOTE)
                        && (ch != SH_DQUOTE)&& (ch != SH_PIPE)) {
                                tokbuf[tbindex++] = ch;
                                p++;
                }

                /* Report error if other token is appended */

                if (       (ch == SH_SQUOTE) || (ch == SH_DQUOTE)
                        || (ch == SH_LESS)   || (ch == SH_GREATER) ) {
                        return SYSERR;
                }

                tokbuf[tbindex++] = NULLCH;     /* Terminate the token  */

                ntok++;                         /* Count valid token    */

        }
        return ntok;
}
```

A Few Shell Declarations (Part 1)

```
/* shell.h - Declarations and constants used by the Xinu shell */

/* 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 */

/* Message constants */

/* Shell banner (assumes VT100) */

#define SHELL_BAN0      "\033[31;1m"
#define SHELL_BAN1      "-----------------------------------------"
#define SHELL_BAN2      "   __    __   _____    _   _    _    _    "
#define SHELL_BAN3      "   \\ \\  / /  |__ __|  | \\ | |  | |  | |   "
#define SHELL_BAN4      "    \\ \\/ /     | |    |  \\| |  | |  | |   "
#define SHELL_BAN5      "    / /\\ \\     _| |_   | \\   |  | |  | |   "
#define SHELL_BAN6      "   / /  \\ \\   |     |  | | \\ |  \\  --  /   "
#define SHELL_BAN7      "   --    --   -----    -   -     ----      "
#define SHELL_BAN8      "-----------------------------------------"
#define SHELL_BAN9      "\033[0m\n"
```

A Few Shell Declarations (Part 2)

```
/* Messages shell displays for user */

#define SHELL_PROMPT    "xsh $ "        /* Command prompt               */
#define SHELL_STRTMSG   "Welcome to Xinu!\n"/* Welcome message          */
#define SHELL_EXITMSG   "Shell closed\n"/* Shell exit message           */
#define SHELL_SYNERRMSG "Syntax error\n"/* Syntax error message         */
#define SHELL_CREATEMSG "Cannot create process\n"/* command error       */
#define SHELL_INERRMSG  "Cannot open file %s for input\n" /* Input err  */
#define SHELL_OUTERRMSG "Cannot open file %s for output\n"/* Output err */
                                /* Builtin cmd error message    */
#define SHELL_BGERRMSG  "Cannot redirect I/O or background a builtin\n" */
#define SHELL_PIPEMSG   "Cannot create a pipe\n"/* error opening a pipe */

/* 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       */
#define SH_PIPE         '|'             /* Pipeline symbol              */
```

A Few Shell Declarations (Part 3)

```
/* 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_PIPE     3               /* Pipeline token               */
#define SH_TOK_OTHER    4               /* 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              */
        int32   (*cfunc)(int32,char*[]);/* Function for command         */
};

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

A Slight Detour

- Before looking at shell code, we will
- Consider the constraints on arguments passed to commands
- Examine a clever way to handle arguments

Command Arguments (A Review)

- Recall that
  - A newly-created Xinu process can be passed arguments
  - The arguments are placed in a pseudo call on the new process stack
  - The create function takes a number of arguments, each of which is one word (integer, pointer, etc)
- Also recall
  - A command process receives two arguments, a count and an array of pointers to strings

A Question

- When creating a process to run a command, the shell must pass two values to create
  - The count of command-line arguments
  - The address of an array of pointers to strings
- Question: where should the shell store
  - The array of pointers to strings?
  - The actual strings?

Argument Storage With Pipelines And Background Processing

- A pipeline and background processing complicate argument passing
  - Multiple commands can execute concurrently
  - Each command can have a different set of arguments
  - The shell cannot use a single (local or global) variable to store the argument strings for all commands
- Important idea: storage for command-line arguments should be released when the process executing the command exits

Storing Command Arguments

- One possibility
  - Modify create
  - Change the pseudo-call
  - Push command-line arguments on the top of new process's stack first and then push on the pseudo-call
- Another possibility
  - Keep create unmodified
  - Place command-line arguments somewhere else
  - Ensure storage for the arguments is freed when the process exits

Freeing Storage When A Command Process Exits

- Perhaps we could use separate storage for arguments
  - Have the shell call *getmem* to allocate storage for arguments
  - Modify *kill* to release the storage used for arguments when a process exits
- Perhaps the shell could store arguments somewhere else in the stack of the process that runs the command
  - Kill can remain unmodified
  - The stack will be freed automatically when the process exits

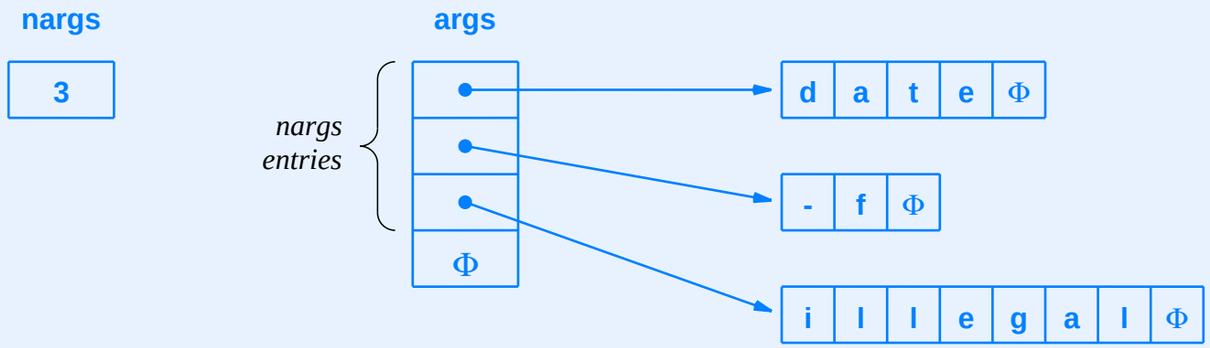
A Trick Used To Store Command-Line Arguments

- Create a process to run a command
- Before resuming the process, insert the arguments into bottom of the command's process stack
  - Store both the *args* array and the actual argument strings
  - Use a single contiguous area of the stack
  - Added advantage: if a process accidentally overwrites its arguments, no other process will be affected
- One wrinkle
  - Arguments for a new process must be specified when (create) is called
  - The location of the args array is not known before calling *create*

Delayed Argument Binding

- When calling *create* make the second argument the address of a temporary variable, *t*
- Add command-line arguments to the bottom of the newly-created stack
- Let the location of the args array be *s*
- Replace the temporary address with the location of the args array
  - Search the new process's stack for address *t*
  - Replace the address with *s*
- When the process is resumed, the second argument will point to the args array

The Layout Of Xinu Shell Command Arguments At Runtime



- After creating a process, the shell
  - Computes the size needed for arguments, and copies them into the bottom of the process's stack
  - Changes the second argument in the pseudo-call to point to the args array in the bottom of the stack (i.e., the location where the command-line arguments are stored)
  - Resumes the process

Addargs.c (Part 1)

```
/* addargs.c - addargs */

#include <xinu.h>
#include "shprototypes.h"

/*------------------------------------------------------------------------
 *  addargs  -  Add local copy of argv-style arguments to the stack of
 *                a command process that has been created by the shell
 *------------------------------------------------------------------------
 */
status  addargs(
          pid32         pid,            /* ID of process to use         */
          int32         ntok,           /* Count of arguments           */
          int32         tok[],          /* Array of token indices       */
          char          *tokbuf,        /* Array of null-term. tokens   */
          void          *dummy          /* Dummy argument that was      */
                                        /*   used at creation and must  */
                                        /*   be replaced by a pointer   */
                                        /*   to an argument vector      */
        )
{
        intmask mask;                   /* Saved interrupt mask         */
        struct  procent *prptr;         /* Ptr to process' table entry  */
        uint32  aloc;                   /* Argument location in process */
                                        /*   stack as an integer        */
        uint32  *argloc;                /* Location in process's stack  */
                                        /*   to place args vector       */
        char    *argstr;                /* Location in process's stack  */
                                        /*   to place arg strings       */
```

Addargs.c (Part 2)

```
uint32  *search;                /* Pointer that searches for    */
                                /*   dummy argument on stack    */
uint32  *aptr;                  /* Walks through args array     */
int32   i;                      /* Index into tok array         */
int32   len;                    /* Length of argument strings   */
char    *first, *last;          /* Address of first and last    */
                                /*   tokens in tokbuf           */

mask = disable();

/* Check argument count and data length */

if (ntok <= 0) {
        restore(mask);
        return SYSERR;
}

prptr = &proctab[pid];

/* Compute lowest location in the process stack where the       */
/*      args array will be stored followed by the argument      */
/*      strings                                                 */

aloc = (uint32) (prptr->prstkbase
        - prptr->prstklen + sizeof(uint32));
argloc = (uint32*) ((aloc + 3) & ~0x3); /* Round to mult. of 4  */
```

Addargs.c (Part 3)

```
/* Compute the first location beyond args array for the strings */

argstr = (char *) (argloc + (ntok+1));  /* +1 for a null ptr    */

/* Set each location in the args vector to be the address of    */
/*      string area plus the offset of this argument            */

for (aptr=argloc, i=0; i < ntok; i++) {
        *aptr++ = (uint32) (argstr + tok[i]-tok[0]);
}

/* Add a null pointer to the args array */

*aptr++ = (uint32)NULL;
```

Addargs.c (Part 4)

```
        /* Copy the argument strings from tokbuf into process's stack   */
        /*      just beyond the args vector                             */

        first = &tokbuf[tok[0]];
        last  = &tokbuf[tok[ntok-1]];
        len = last - first + strlen(last) + 1;

        memcpy(aptr, first, len);

        /* Find the second argument in process's stack */

        for (search = (uint32 *)prptr->prstkptr;
             search < (uint32 *)prptr->prstkbase; search++) {

                /* If found, replace with the address of the args vector*/

                if (*search == (uint32)dummy) {
                        *search = (uint32)argloc;
                        restore(mask);
                        return OK;
                }
        }

        /* Argument value not found on the stack - report an error */

        restore(mask);
        return SYSERR;
}
```

Shell.c (Part 1)

```
/* shell.c  -  shell */

#include <xinu.h>
#include <stdio.h>
#include "shprototypes.h"

/*------------------------------------------------------------------------*/
/* Table of Xinu shell commands and the function associated with each   */
/*------------------------------------------------------------------------*/
const   struct  cmdent  cmdtab[] = {
        {"argecho",     xsh_argecho},
        {"arp",         xsh_arp},
        {"cat",         xsh_cat},
        {"clear",       xsh_clear},
        {"date",        xsh_date},
        {"devdump",     xsh_devdump},
        {"echo",        xsh_echo},
        {"help",        xsh_help},
        {"ls"   ,       xsh_ls},
        {"kill",        xsh_kill},
        {"memdump",     xsh_memdump},
        {"memstat",     xsh_memstat},
        {"ns",          xsh_ns},
        {"netinfo",     xsh_netinfo},
        {"ping",        xsh_ping},
        {"ps",          xsh_ps},
```

Shell.c (Part 2)

```
        {"sleep",       xsh_sleep},
        {"tee",         xsh_tee},
        {"udp",         xsh_udpdump},
        {"udpecho",     xsh_udpecho},
        {"udpeserver",  xsh_udpeserver},
        {"uptime",      xsh_uptime},
        {"?",           xsh_help}
};

uint32  ncmd = sizeof(cmdtab) / sizeof(struct cmdent);

/*------------------------------------------------------------------------*/
/*                                                                        */
/* cmdlookup -- return the index in cmdtab of a command name              */
/*                                                                        */
/*------------------------------------------------------------------------*/

int32   cmdlookup(char *name) {
        int32   indx;           /* Index into cmdtab                      */
        int32   len;            /* Length of command name                 */

        for (indx=0; indx < ncmd; indx++) {
                len = strlen(cmdtab[indx].cname);
                if ( strncmp(name,cmdtab[indx].cname, len) == 0 ) {
                        return indx;
                }
        }
        return SYSERR;
}
```

Shell.c (Part 3)

```
/************************************************************************/
/* shell  -  Provide an interactive user interface that executes        */
/*           commands.  An input line contains one or more segments     */
/*           separated by a pipe symbol '|'.  Each segment begins with  */
/*           a command name, and has a set of optional arguments,  The  */
/*           last segment in a pipeline can have output redirected.     */
/*           The entire pipeline can be run in background by ending the */
/*           line with an ampersand, "&".  The syntax is:               */
/*                                                                      */
/*                 seg [ pipe seg ]*  [ out_redir ]  [ & ]              */
/*           where:                                                     */
/*                                                                      */
/*              pipe is         |                                       */
/*                                                                      */
/*              seg is          command_name [args*]                    */
/*                                                                      */
/*              out_redir is    > output_file                           */
/*                                                                      */
/************************************************************************/
```

Shell.c (Part 4)

```
process shell (
                did32   dev             /* ID of tty device from which  */
        )                               /*   to accept commands         */
{
        char    buf[SHELL_BUFLEN];      /* Input line (large enough for */
                                        /*   one line from a tty device */
        int32   len;                    /* Length of line read          */
        char    tokbuf[SHELL_BUFLEN +   /* Buffer to hold a set of      */
                        SHELL_MAXTOK];  /*   contiguous null-terminated */
                                        /*   strings of tokens          */
        int32   tok[SHELL_MAXTOK];      /* Index of each token in       */
                                        /*   array tokbuf               */
        int32   toktyp[SHELL_MAXTOK];   /* Type of each token in tokbuf */
        int32   ntok;                   /* Number of tokens on line     */
        bool8   backgnd;                /* Run command in background?   */
        char    *inname;                /* File name for input re-      */
                                        /*   direction on first segment */
        did32   indesc;                 /* Descriptor for redirected    */
                                        /*   input                      */
        char    *outname;               /* File name for output re-     */
                                        /*   direction on last segment  */
        did32   outdesc;                /* Descriptor for redirected    */
                                        /*   output                     */
        int32   nsegs;                  /* Number of segments found     */
```

Shell.c (Part 5)

```
struct segent {                 /* One segment of the pipeline  */
    int32       sstart;         /* Starting token index         */
    int32       send;           /* Ending token index           */
    int32       scindex;        /* Index in cmdtab of the       */
                                /*   command in this segment    */
    did32       soutdev;        /* Output device (pipe, except  */
                                /*   for first segment)         */
    did32       sindev;         /* Input device (pipe, except   */
                                /*   for the last segment)      */
    pid32       spid;           /* Process ID for this segment  */
};
struct  segent  segtab[SHELL_MAXTOK];/* One entry per segment   */
struct  segent  *segptr;        /* Pointer to a segtab entry    */
int32   seg;                    /* Index into segtab            */
int32   cindex;                 /* Index of command returned    */
                                /*   by cmdlookup               */
did32   pipedev;                /* Device ID of a pipe device   */
int32   i;                      /* Index into array of tokens   */
char    *p;                     /* Pointer to cmd name          */
bool8   err;                    /* Did an error occur?          */
int32   msg;                    /* Message from receive() for   */
                                /*   child termination          */
int32   tmparg;                 /* Temporary address used when  */
                                /*   creating a child process;  */
                                /*   later replaced by addargs  */
```

Shell.c (Part 6)

```
/* Print shell banner and startup message */

fprintf(dev, "\n\n%s%s\n%s\n%s\n%s\n%s\n%s\n%s\n%s\n%s\n",
        SHELL_BAN0,SHELL_BAN1,SHELL_BAN2,SHELL_BAN3,SHELL_BAN4,
        SHELL_BAN5,SHELL_BAN6,SHELL_BAN7,SHELL_BAN8,SHELL_BAN9);

fprintf(dev, "%s\n\n", SHELL_STRTMSG);

/* Continually prompt the user, read input, and execute command */

while (TRUE) {

        /* Display prompt */

        fprintf(dev, SHELL_PROMPT);

        /* Read a command */

        len = read(dev, buf, sizeof(buf));

        /* Exit gracefully on end-of-file */

        if (len == EOF) {
                break;
        }
```

Shell.c (Part 7)

```
/* If line only contains 'exit', exit gracefully */

if ( (len==5) && (strncmp(buf,"exit\n", 5)==0) ) {
        break;
}

/* If line contains only NEWLINE, go to next line */

if (len <= 1) {
        continue;
}

buf[len] = SH_NEWLINE;  /* Terminate line */

/* Parse input line and divide into tokens */

ntok = lexan(buf, len, tokbuf, tok, toktyp);

/* Handle parsing error */

if (ntok == SYSERR) {
        fprintf(dev,"%s\n", SHELL_SYNERRMSG);
        continue;
}
```

Shell.c (Part 8)

```
/* If line is empty, go to next input line */

if (ntok == 0) {
        fprintf(dev, "\n");
        continue;
}

/* Set default input for first segment and output for   */
/*   last segment to the device used to call the shell  */

/* See if the last token is '&', and set background */

if (toktyp[ntok-1] != SH_TOK_AMPER) {
        backgnd = FALSE;
} else {
        backgnd = TRUE;
        ntok-- ;
        if (ntok == 0 ) {
                fprintf(dev, "\n");
                continue;
        }
}
```

Shell.c (Part 9)

```
/* Use the pipe tokens to divide the pipeline into      */
/*  segments by placing the index of pipe tokens in     */
/*  successive locations of array segtab.               */

nsegs = 0;
segptr = &segtab[nsegs];
segptr->sstart = 0;
for (i=0; i<ntok; i++) {
        /* Check for pipe at end of segment */
        if (toktyp[i] == SH_TOK_PIPE) {
                /* Finish old segment and start new */
                segptr->send = i - 1;
                /* Check for empty segment */
                if (segptr->sstart > segptr->send) {
                    fprintf(dev,"%s\n", SHELL_SYNERRMSG);
                    break;
                }
                /* Move to new segment */
                nsegs++;
                segptr = &segtab[nsegs];
                segptr->sstart = i+1;
        }
}
```

Shell.c (Part 10)

```
if (i < ntok) {
        /* Error occurred, so go to next input */
        continue;
}

/* Fill in remaining details for last segment */

segptr->send = ntok - 1;
if (segptr->sstart > segptr->send) {
        fprintf(dev,"%s\n", SHELL_SYNERRMSG);
        continue;
}
nsegs++;

/* Check for output redirection on last segment */

segptr = &segtab[nsegs - 1];
outname = NULL;
if ( ((segptr->send-segptr->sstart) > 1) &&
        (toktyp[segptr->send-1] == SH_TOK_GREATER) ) {
        if (toktyp[segptr->send] != SH_TOK_OTHER) {
                fprintf(dev,"%s\n", SHELL_SYNERRMSG);
                continue;
        }
        outname = &tokbuf[tok[segptr->send]];
        segptr->send -= 2;
}
```

Shell.c (Part 11)

```
/* Check for input redirection on first segment */

segptr = &segtab[0];
inname = NULL;
if ( ((segptr->send-segptr->sstart) > 1) &&
        (toktyp[segptr->send-1] == SH_TOK_LESS) ) {
        if (toktyp[segptr->send] != SH_TOK_OTHER) {
                fprintf(dev,"%s\n", SHELL_SYNERRMSG);
                continue;
        }
        inname = &tokbuf[tok[segptr->send]];
        segptr->send -= 2;
}
```

Shell.c (Part 12)

```
ft C
                /* Check that all tokens in each segment are "other" */

                err = FALSE;
                for (seg = 0; seg < nsegs; seg++) {
                        segptr = &segtab[seg];
                        for (i=segptr->sstart; i<= segptr->send; i++) {
                            if (toktyp[i] != SH_TOK_OTHER) {
                                fprintf(dev,"%s\n", SHELL_SYNERRMSG);
                                err = TRUE;
                                break;
                            }
                        }
                        if (err) {
                            break;
                        }
                }
                if (err) {
                        continue;
                }
```

Shell.c (Part 13)

```
/* For each command, look up the name of the command    */

for (seg = 0; seg< nsegs; seg++) {
    segptr = &segtab[seg];
    p = &tokbuf[tok[segptr->sstart]];
    cindex = cmdlookup(p);
    if (cindex == SYSERR) {
        fprintf(dev, "command %s not found\n", p);
        break;
    }
    segptr->scindex = cindex;
}
if (seg < nsegs) {
        /* Error occurred, so go to next input */
        continue;
}
```

Shell.c (Part 14)

```
/* Open files for redirected input and output */

indesc = outdesc = dev;

if (inname != NULL) {
        indesc = open(NAMESPACE,inname,"ro");
        if (indesc == SYSERR) {
                fprintf(dev, SHELL_INERRMSG, inname);
                continue;
        }
}
segtab[0].sindev = indesc;

if (outname != NULL) {
        outdesc = open(NAMESPACE,outname,"w");
        if (outdesc == SYSERR) {
                fprintf(dev, SHELL_OUTERRMSG, outname);

                /* Close input file if it was opened */
                if (indesc != dev) {
                        close(indesc);
                }

                continue;
        }
        control(outdesc, F_CTL_TRUNC, 0, 0);
}
segtab[nsegs-1].soutdev = outdesc;
```

Shell.c (Part 15)

```
/* Create nsegs-1 pipes (to go "between" the segments)  */
err = FALSE;
for (seg = 0; seg < nsegs - 1; seg++) {
        segptr = &segtab[seg];
        pipedev = open(PIPE,NULLSTR,"rw");
        if (pipedev == SYSERR) {
                err = TRUE;
                fprintf(dev,"Pipe open failed\n");
                break;
        }
        segptr->soutdev = pipedev;
        segtab[seg+1].sindev= pipedev;
}
if (err) {
        /* Close previously opened pipes */
        for(seg--; seg >= 0; seg--) {
                close(segtab[seg].soutdev);
        }
        /* Close previously opened infile */
        if (indesc != dev) {
                close(indesc);
        }
        /* Close previously opened outfile */
        if (outdesc != dev) {
                close(outdesc);
        }
        continue;
}
```

Shell.c (Part 16)

```
/* Spawn a process for each segment  */
int32 ntokens;
err = FALSE;
for (seg = 0; (seg<nsegs) && !err; seg++) {
    segptr = &segtab[seg];
    ntokens = segptr->send - segptr->sstart +1;
    segptr->spid = create(cmdtab[segptr->scindex].cfunc,
        SHELL_CMDSTK, SHELL_CMDPRIO,
        cmdtab[segptr->scindex].cname, 2,
        ntokens, &tmparg);
    /* If creation fails, report the error */
    if (segptr->spid == SYSERR) {
        fprintf(dev, SHELL_CREATEMSG);
        err = TRUE;
        continue;
    }
    /* If adding arguments fails,report the error */
    if (addargs(segptr->spid,ntokens,&tok[segptr->sstart],
                tokbuf, &tmparg)== SYSERR) {
        fprintf(dev, SHELL_CREATEMSG);
        err = TRUE;
        break;
    }
}
```

Shell.c (Part 17)

```
if (err) {
    /* Undo processes created before the error */
    for(seg-- ; seg >= 0; seg--) {
        kill(segtab[seg].spid);
    }
}

/* Redirect input and output for each process */
for (seg=0; seg< nsegs; seg++) {
    segptr = &segtab[seg];
    proctab[segptr->spid].prdesc[0] = segptr->sindev;
    proctab[segptr->spid].prdesc[1] = segptr->soutdev;
}

msg = recvclr();

/* Resume each process in the pipeline (the shell will  */
/*   remain running because it has higher priority)     */

for (seg = 0; seg < nsegs; seg++) {
        resume(segtab[seg].spid);
}
```

Shell.c (Part 18)

```
                /* Either block to wait for the last process in the     */
                /*   pipeline to finish or allow the pipeline to run    */
                /*   in background                                      */

                pid32   tmppid = segtab[nsegs-1].spid; /* Last seg. pid */

                if (! backgnd) {
                        msg = receive();
                        while (msg != tmppid) {
                                msg = receive();
                        }
                }
        }

    /* Terminate the shell process by returning from the top level */

    fprintf(dev,SHELL_EXITMSG);
    return OK;
}
```

An Example Shell Command (sleep Part 1)

```
/* xsh_sleep.c - xsh_sleep */

#include <xinu.h>
#include <stdio.h>
#include <string.h>

/*------------------------------------------------------------------------
 * xsh_sleep  -  Shell command to delay for a specified number of seconds
 *------------------------------------------------------------------------
 */
shellcmd xsh_sleep(int nargs, char *args[])
{
        int32   delay;                  /* Delay in seconds             */
        char    *chptr;                 /* Walks through argument       */
        char    ch;                     /* Next character of argument   */

        /* For argument '--help', emit help about the 'sleep' command   */

        if (nargs == 2 && strncmp(args[1], "--help", 7) == 0) {
                printf("Use: %s\n\n", args[0]);
                printf("Description:\n");
                printf("\tDelay for a specified number of seconds\n");
                printf("Options:\n");
                printf("\t--help\t display this help and exit\n");
                return 0;
        }
```

An Example Shell Command (sleep Part 2)

```
        /* Check for valid number of arguments */

        if (nargs > 2) {
                fprintf(stderr, "%s: too many arguments\n", args[0]);
                fprintf(stderr, "Try '%s --help' for more information\n",
                                args[0]);
                return 1;
        }

        if (nargs != 2) {
                fprintf(stderr, "%s: argument in error\n", args[0]);
                fprintf(stderr, "Try '%s --help' for more information\n",
                                args[0]);
                return 1;
        }

        chptr = args[1];
        ch = *chptr++;
        delay = 0;
        while (ch != NULLCH) {
                if ( (ch < '0') || (ch > '9') ) {
                        fprintf(stderr, "%s: nondigit in argument\n",
                                args[0]);
                        return 1;
                }
                delay = 10*delay + (ch - '0');
                ch = *chptr++;
        }
        sleep(delay);
        return 0;
}
```

Mice And Windowing Systems

A Mouse (Or Trackpad)

- A pointing device invented as a companion to a 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

- A mouse contains
  - Two motion detectors
    * Arranged at right angles
    * Labeled *X* and *Y*
  - Two A-to-D converters
  - *1* to *3* buttons
  - Scroll wheel(s) or touch controls for scrolling

Communication Between A Mouse And A Computer

- Can use a variety of physical hardware connections
  - Traditional serial port (RS232)
  - Serial communication over USB
  - Serial communication over Bluetooth
- Uses the same interface as a keyboard
  - Individual characters
  - Sends or receives one character at a time

Two-Way Mouse Communication

- Is unexpectedly sophisticated
- A mouse uses two-way interaction
- A mouse can
  - Accept commands from the computer
  - Respond to queries from the computer
  - Transmit data to the computer asynchronously

Mouse Modes

- Two basic modes are used
  - Polling mode
  - Streaming mode
- A given system may support both modes and allow the windowing system and/or individual applications to choose

Polling Mode

- A processor sends a request for information
- The mouse responds by reporting
  - Motion since last request
  - The status of the buttons
- Typically polling is only used
  - By low-end embedded systems
  - To reset 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 pressed or released
  - When the thumbwheel moves

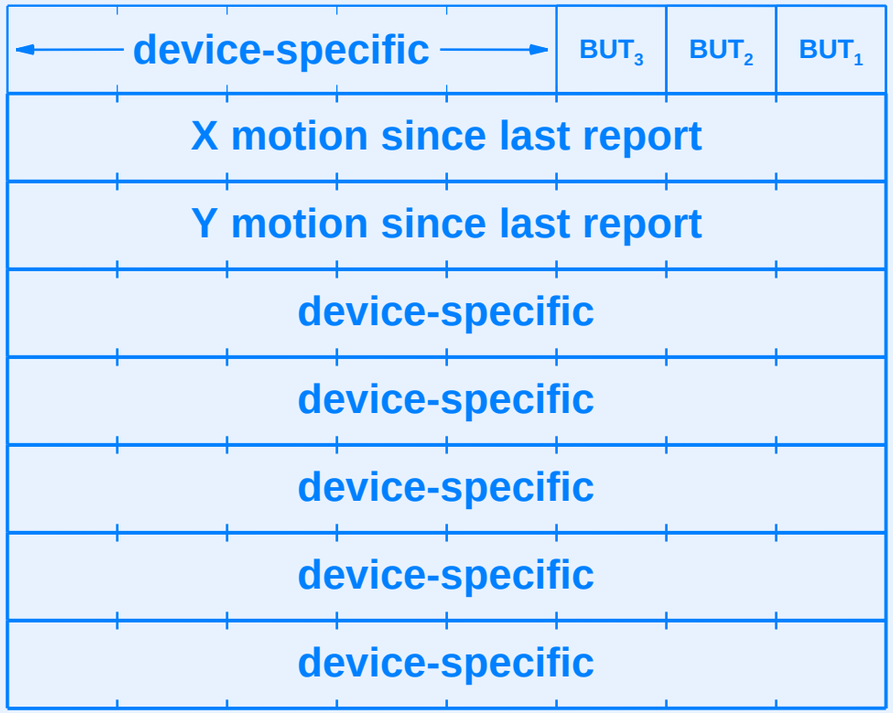
The Mouse Communications Interface

- Communication with a mouse is
  - Asynchronous (can occur at any time)
  - Full duplex (both directions operate independently)
  - Performed with 8-bit bytes, not just printable characters
- Consequence: a mouse may send a report at the same time a computer sends a command to the mouse

The Mouse Communications Interface
(continued)

- Each message occupies multiple 8-bit bytes
- Example: each report sent by a mouse
  - Consists of one message with multiple fields
  - Each field in the message is a fixed length
  - The format is known as a *mouse packet*
- Notes
  - Although we use the term *packet*, the serial hardware only transfers individual bytes
  - The values in fields are interpreted according to current parameter settings

An Example Mouse Packet (USB 3-Button Mouse)



- Each mouse packet is eight 8-bit bytes
- BUT_1, BUT_2, and BUT_3 give button status
- Device specific fields depend on the vendor and model
- Example device-specific information: X or Y overflow

Mouse Parameters

- Are set by the computer
- Determine
  - How motion is measured and scaled
  - How the mouse uses thresholds to decide when to send reports
- Users may be able to specify preferences

Typical Mouse Parameters

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

Memory Mapped Display Screens

- A screen is divided into pixels
- Most displays are *memory mapped*, which means
  - A portion of the memory address space is reserved for each display
  - The operating system writes values to the display memory to change pixels on the screen
  - The display hardware repeatedly scans the display memory and updates the screen accordingly
- A modern graphics card keeps the display memory on the card and includes hardware that scans the display memory and updates the display faster than possible with normal DRAM

Display Hardware

- Early screens were black-and-white
  - One bit in the display memory corresponded to one pixel on the screen
  - The early displays were known as *bit-mapped* screens
- Current screens display color
  - One or more bytes of display memory correspond to a pixel on the screen
  - Color displays are sometimes called *byte-mapped*, but the term *bit-mapped* persists

Color Specification

- The hardware uses three primary colors: red, green, blue (*RGB*)
- Originally, the colors came from phosphors illuminated by an electron beam; modern displays use LEDs to produce the colors
- Think back to elementary school art class
  - Everyone learns that the primary colors are red, blue, and yellow
  - Other colors can be made by mixing primary colors
  - Example: yellow plus blue produces green
- Questions
  - Was the art teacher incorrect?
  - Why don’t computer displays use the same primary colors

Color Specification
(continued)

- Answer:
  - Art classes use *reflective colors*
  - Computer screens use *generated color*
- A major difference
  - Artists draw on white paper, so the absence of color is white (less color results in a lighter shade)
  - On a computer, the absence of color is black (less color results in a darker shade)

Combining Colors

- On a display, all colors result from a combination of 0% to 100% of red, green, and blue
- RGB colors combine in interesting and unexpected ways
  - Example 100% red + 100% green + 0% blue gives yellow
  - Recall that adding less of a color makes a darker shade (closer to black), so starting with full blue and reducing the amount will make a darker blue
- In practice
  - An integer value is used for each color instead of a floating point percentage
  - Typically, the value for each color occupies one byte and ranges from 0 to 255
  - A total of 16,777,216 colors are possible

Extending Colors

- Additional colors can be provided two ways
  - Increase the number of bits (or bytes) per pixel
  - Use *color map* technology
- Color map technology
  - Observe that a given image or set of images do not contain many colors
  - Build mapping hardware that stores an array of multi-byte colors
  - Use an RGB value as an index into the array
  - Change the array when changing to a set of images with new colors

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 (but other shapes have been tried)
  - A window must be created before it can be used
  - The OS presents an application with a separate coordinate space for each window, where (0,0) is a 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 and labels
- 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, but means users may not all see the same display

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 (i.e., a mouse packet arrives)
  - Map the new position to the correct window so mouse clicks can be forwarded to the process that owns the window

The Cursor Update Algorithm

- When the cursor moves, the window manager must
  - 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
- Delay makes cursor motion jerky
- To avoid long delays and achieve smooth cursor motion, the system must perform cursor update in the lower-half of device driver as part of 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: unlike early computers, modern computers have fast, multicore processors that make lost interrupts unlikely

Summary

- Operating system support two styles of user interface
  - Graphical User Interface
  - Command-line interpreter
- 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”
- A computer display
  - Has primary colors red, green, and blue that add to generate all possible colors rather than the red, yellow, and blue primary colors used in art class to reflect color
  - Uses a memory-mapped approach where a processor writes bytes in a display memory and display hardware constantly scans the memory and updates the display
  - For a color display, the hardware display memory stores three values for each pixel that correspond to 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?