# CS146

## Software Tools and System Programming

Practical notes: ssh, cygwin, remote access; man. Cover regular expressions, variables, and quoting early; and have a discussion / study group in Week 1; eg., sections 3+4 need to be covered early for Ass't #1; put a comment that all questions in a2 except prargs have 1-line solutions.

Wayne Hayes
Fall 2010

# Course Goals

- Using Unix for software development (RCS, make, compilers, debuggers)
- Unix systems programming (system call interface, Unix kernel)
- Concurrent programming (threads, process synchronization)



# About these slides

These slides derive much of their content from the originals by David A. Penny and the modifications made by Wayne Hayes, for a similar course at University of Toronto. Sean M. Culhane's ideas were also used. The original LaTeX slides were converted to PowerPoint by Arthur Tateishi.



# Section #1

Basic UNIX Structure

and

OS Concepts

# What is UNIX good for?

- A generic interface to computing equipment
- Supports many users running many programs at the same time, all sharing (transparently) the same computer system
- Promotes information sharing
- Geared for high programmer productivity. “Expert friendly”
- Generic framework allows flexible tailoring for users.
- Services include:

  File system, Security, Process/Job Scheduling, Network services/abstractions.



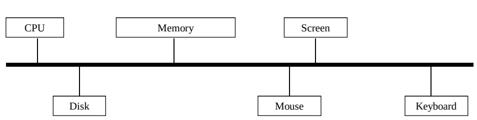
# History

- Ken Thompson working at Bell Labs in 1969 wanted a small MULTICS for his DEC PDP-7
- He wrote UNIX which was initially written in assembler and could handle only one user at a time
- Dennis Ritchie and Ken Thompson ported an enhanced UNIX to a PDP-11/20 in 1970
- Ritchie ported the language BCPL to UNIX in 1970, cutting it down to fit and calling the result “B”
- In 1973 Ritchie and Thompson rewrote UNIX in “C” and enhanced it some more
- Since then it has been enhanced and enhanced and enhanced and …



# Computer Hardware

- CPU - Central Processing Unit carries out the instructions of a program
- Memory - used for “small” information storage (e.g. < 4GB)
- I/O devices - used for communicating with the outside world such as screen, keyboard, mouse, disk, tape, modem, network
- Bus - links CPU, I/O, and Memory






# Machine Language

- CPU interprets machine language programs:

```
1100101 11111111 11010000 00000000
1010001 01011101 00000010 00000000
1100101 00000000 11111111 00100100
```

- Assembly language instructions bear a one-to-one correspondence with machine language instructions

```
MOVE   FFD0, D0         % b = a * 2
MUL    #2, D0
MOVE   D0, FFDC
```

# Compilation

- High Level Language (HLL) is a language for expressing algorithms whose meaning is (for the most part) independent of the particular computer system being used
- A *compiler* translates a high-level language into *assembly language* (object files).
- A *linker* translates assembly language programs (object files) into a *machine language* program (an executable)
- Example:
  - create object file “**`fork.o`**” from C program “**`fork.c`**”:

    **`gcc -c fork.c -o fork.o`**
  - create executable file “**`fork`**” from object file “**`fork.o`**”:

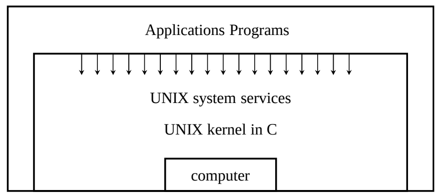
    **`gcc fork.o -o fork`**



# UNIX Kernel

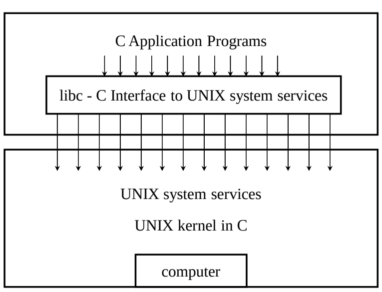
- A large C program that implements a general interface to a computer to be used for writing programs:

```
fd = open("/dev/tty", O_WRONLY);
write(fd, "Hello world!", 12);
```





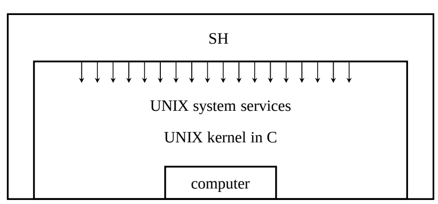

# C and libc



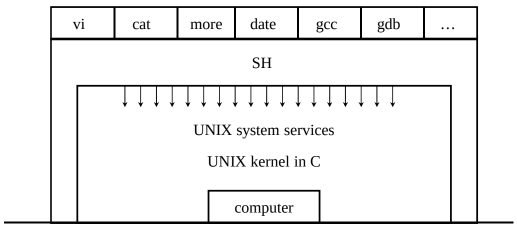



# Shell

- The shell (sh) is a program (written in C) that interprets commands typed to it, and carries out the desired actions.
- The shell is that part of Unix that most users see. Therefore there is a mistaken belief that sh is Unix.
- sh is an applications program running under Unix
- Other shells exists (ksh, csh, tcsh, bash)



# Tools and Applications






# Section #2

## UNIX File Abstraction
## and
## File System Organization



# What is a File?

- A file is the most basic entity in a UNIX system.
- Several different kinds of files:
  - Regular
  - Directory
  - Character Special
  - Block Special
  - Socket
  - Symbolic Link
- They are accessed through a common interface (i.e. you need only learn how to use one set of systems calls to be able to access any sort of file.)



# Regular Files

- A regular file is a named, variable length, sequence of bytes.
- UNIX itself assumes no special structure to a regular file beyond this.
- Most UNIX utility programs, however, do assume the files have a certain structure.
- e.g.

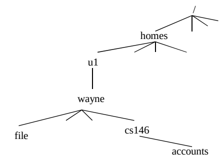
```
$ cat > file
hello world!
^D
$ ls -l file
-rw-r--r--   1    wayne         13  May  8  16:44   file
$ cat file
hello world!
$ od -cb file
0000000   h   e   l   l   o       w   o   r   l   d   !  \n
        150 145 154 154 157 040 167 157 162 154 144 041 012
0000015
```

# Regular Files (cont.)

- Regular files are used to store:
  - English Text
  - Numerical Results
  - Program Text
  - Compiled Machine Code
  - Executable Programs
  - Databases
  - Bit-mapped Images
  - etc...



# Directories & Filenames




- Directories are special kinds of files that contain references to other files and directories.
- Directory files can be read like a regular file, but UNIX does not let you write to them.
- There are two ways of specifying a filename
  - absolute: /homes/u1/wayne/file
  - relative: cs146/accounts
- With an absolute pathname the search for the file starts at the *root* directory.



# Relative Pathnames

- With a relative pathname the search for the file starts at the current working directory.
- Every process under UNIX has a CWD. This can be changed by means of a system call.
- e.g.

```
$ pwd
/homes/u1/wayne
$ cd cs146
$ pwd
/homes/u1/wayne/cs146
$ cd /
$ pwd
/
```



# Device Files

- All forms of I/O in UNIX go through the file interface.
- To write to a terminal's screen, for instance, you just write to the appropriate device file:

```
$ cat > /dev/ttya
Hi guy!^D
```

- This will cause the text "Hi guy!" to appear on a screen.
- To read from a terminal's keyboard you just read from the appropriate device file:

```
$ cat /dev/ttya
```

- The same holds true for disks, tapes, mice, tablets, robot arms, the computer's ram memory, etc…

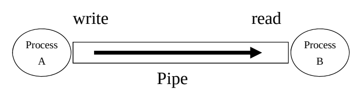
## Block Special & Character Special Device Files

- There are two kinds of interfaces to devices in UNIX:
  - block interface
  - character interface
- If input and output are buffered in fixed-size blocks within the operating system, the device has a block special file as its interface.
- If the input and output are unbuffered, the device has a character special file as its interface.



## Sockets & Pipes

- Pipes are special files used to pass bytes between two processes.




- Sockets are similar, but are used to connect two processes on different machines across a network.



## File Permissions

- Every user of the system has a login name.
- The file /etc/passwd associates a UID, GID, and password with each login name.
- When a file is created, the UID and GID of the creator are remembered.
- Every named file has associated with it a set of permissions in the form of a string of bits.

| Owner | Group | Others |
|---|---|---|
| r w x s | r w x s | r w x |

| mode | regular/device | directory |
|---|---|---|
| r | read | list contents |
| w | write | create and remove |
| x | execute | query and chdir |
| s | setuid/gid | (see “man chmod”) |



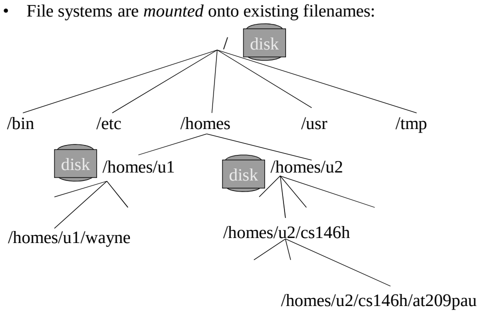
## Inodes

- Each distinct file in UNIX has an *inode* that refers to it.
- An *inode* contains:
  - type of file
  - time of inode last modified
  - time file data last written
  - time file data last read
  - creator’s user ID
  - creator’s group ID
  - number of directory links
  - file size
  - pointers to disk blocks containing data

  or the major and minor device ID
  - permission bits
  - sticky bit

# Mounting

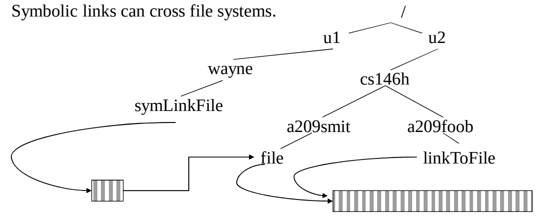
- A *file system* is contained on a disk.
- File systems are *mounted* onto existing filenames:




CS146 25

# Hard Links & Symbolic Links

- Directory files contain (filename, i-number) pairs.
- Each such entry is called a *link*.
- A file can have more than one link.
- Regular links (hard links) are not allowed to cross file systems.
- A different kind of link, a *symbolic link*, contains the pathname of the linked to file.
- Symbolic links can cross file systems.




CS146 26

# Section #3

## UNIX Processes

## and

## Shell Internals

CS146 27

# The Shell

- A UNIX *shell* is a program that interprets commands
  - It *translates* commands that you type into system calls.
- The shell is a tool that is used to increase productivity by providing a suite of features for running other programs in different configurations or combinations.
- We will be discussing “sh”, otherwise known as the Bourne Shell.
  - Other shells exist:
    - csh - The C Shell
    - ksh - The Korn Shell
    - bash - The GNU Bourne-Again Shell.

CS146 28

# File Descriptors

- In UNIX, all **read** and **write** system calls take as their first argument a *file descriptor* (not a filename).
- To get a file descriptor you must perform an open or a creat system call.

```
int fd;
fd = open(pathname, rwmode);
```

- You are given the lowest numbered free file descriptor available (starting from 0).
- The **open** and **creat** system calls allocate resources within the operating system to speed up subsequent file access.
- When a program is done with a file it should call **close**:

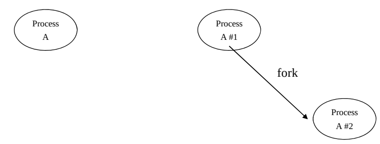
```
close(fd);
```

- When a process terminates execution, all its open files are automatically closed.



# Fork

- The **fork** system call is used to create a duplicate of the currently running program.




- The duplicate (child process) and the original (parent process) both process from the point of the fork with exactly the same data.
- The only difference between the two processes is the **fork** return value.



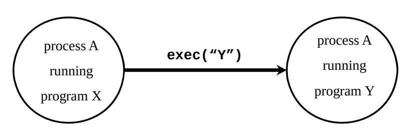
# Fork example

```
int i, pid;
i = 5;
printf( “%d\n”, i );
pid = fork();

if( pid != 0 )
   i = 6; /* only the parent gets to here */
else
   i = 4; /* only the child gets to here */

printf( “%d\n”, i );
```



# Exec

- The *exec* system call replaces the program being run by a process by a different one
- The new program starts executing from its beginning




- Variations on exec: **`execl()`**, **`execv()`**, etc. which will be discussed later in the course

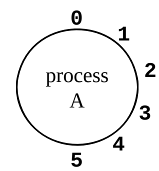
# Exec example

```
PROGRAM X
int i;
i = 5;
printf( “%d\n”, i );

exec( “Y” );

i = 6;
printf( “%d\n”, i );

PROGRAM Y
printf( “hello” );
```



# Processes and File Descriptors

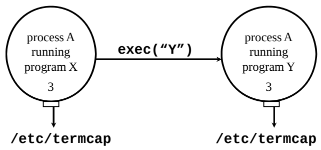
- File descriptors belong to processes. (Not programs!)
- They are a process’ link to the outside world.






# PIDs and FDs across an exec

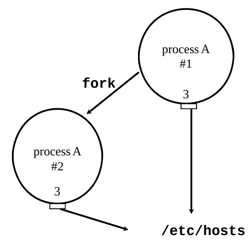
- File descriptors are maintained across *exec* calls:






# PIDs and FDs across a fork

- File descriptors are maintained across *fork* calls:



## Fork: PIDs and PPIDs

- System call: **`int fork()`**
- If **`fork()`** succeeds, it returns the child PID to the parent and returns 0 to the child; if it fails, it returns -1 to the parent (no child is created)

- System call: **`int getpid()`**
  **`int getppid()`**
- **`getpid()`** returns the PID of the current process, and **`getppid()`** returns the PID of the parent process (note: ppid of 1 is 1)
- example (*see next slide …*)



## PID/PPID example

```
#include <stdio.h>
int main( void )
{
   int pid;
   printf( "ORIGINAL: PID=%d PPID=%d\n", getpid(), getppid() );
   pid = fork();
   if( pid != 0 )
      printf( "PARENT: PID=%d PPID=%d child=%d\n",
                                  getpid(), getppid(), pid );
   else
      printf( "CHILD:  PID=%d PPID=%d\n", getpid(), getppid() );

   printf( "PID %d terminates.\n\n", getpid() );
   return( 0 );
}
```



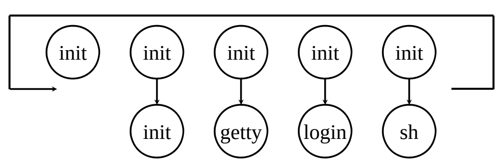
## Initializing UNIX

- The first UNIX program to be run is called "**`/etc/init`**"
- It *forks* and then *exec*s one "**`/etc/getty`**" per terminal
- [NEW] Or, it starts the X-Window system, which we'll discuss later
- getty sets up the terminal properly, prompts for a login name, and then *exec*s "**`/bin/login`**"
- login prompts for a password, encrypts a constant string using the password as the key, and compares the results against the entry in the file "**`/etc/passwd`**"
- If they match, "**`/usr/bin/csh`**" (or whatever is specified in the passwd file as being that user's shell) is *exec*'d
- When the user exits from their login shell, the process dies. Init finds out about it (*wait* system call), and *forks* another getty process for that terminal



## Initializing UNIX

- The first UNIX program to be run is called **init**.
- It **fork**s and then **exec**s one **getty** per terminal.
- getty sets up the terminal properly, prompts for a login name, and then **exec**s login.
- login prompts for a password, encrypts a constant string using the password as the key, and compares the results against the entry in the file "/etc/passwd".
- If they match, the shell (e.g. /bin/sh) is **exec**'d
- When the user exits from his shell, the process dies. Init finds out about it (**wait** system call), and forks another process for that terminal.



# Standard Streams

- The forked inits open the terminals they are assigned to 3 times.
- The result is that when sh is eventually started up, the first three file descriptors (0, 1, 2) are pre-assigned, and refer to the login terminal.

| Descriptor | Name | Purpose |
|---|---|---|
| 0 | Standard Input | Read Input |
| 1 | Standard Output | Write Results |
| 2 | Standard Error | Report Errors |

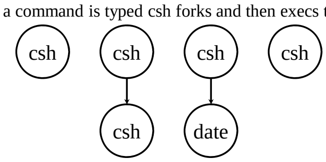
- sh reads its commands from the standard input



# How sh runs commands

```
> date
Sun May 25 23:11:12 EDT 1997
```

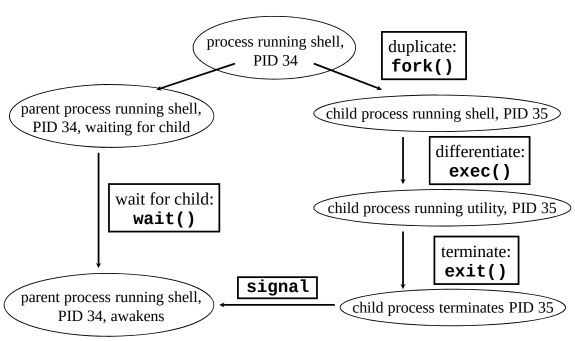
- When a command is typed csh forks and then execs the typed command:




- After the **fork** and **exec**, file descriptors 0, 1, and 2 still refer to the standard input, output, and error in the new process
- By UNIX programmer *convention*, the executed program will use these descriptors appropriately



# How sh runs (cont.)






# I/O redirection

$ cat < f1 > f2

- After the fork, but before the exec, sh can redirect the standard input, output, or error streams (or any other stream for that matter):

```
while(not end of standard input) {
      print(stdout, "% ");
      read_cmd(stdin, command);

      pid = fork();
      if (pid == 0) {
         /* The child executes from here */
         if (inputRedirected) {
             close(stdin);
             open(inputFile, O_RDONLY);
         }
         if (outputRedirected) {
             close(stdout);
             creat(outputFile);
         }
         exec(command);
      } else
         /* parent: wait for child to terminate */
} /* end while */
```

# Pipes

$ ls /u/cs146h | cat

- For a *pipeline*, the standard output of one program is connected to the standard input of the next program.
- Pipelines can be (almost) arbitrarily long.
- Commands in a pipeline are run *concurrently*!
- The output of a pipeline *could* be produced using temporary files, but
  - pipes are implemented in RAM, which is faster than disk.
  - you would lose on the store-and-forward delays
  - programs requiring little CPU can produce lots of I/O, so why not run them concurrently rather than wait for one to finish before starting the next one?
  - you might fill up the disk with large intermediate files.



# Exec arguments

$ echo hello world!
hello world!

- The exec system call has a parameter (not shown previously) that is used to pass command line arguments to the executed commands:

```
char * argv[4];

argv[0] = “echo”;
argv[1] = “hello”;
argv[2] = “world!”;
argv[3] = NULL; /* (char*) 0 */

exec(“/bin/echo”, argv);
```



# Environment Arguments

- The **exec** system call has another parameter (not shown previously) that is used to pass the state of the environment to executed commands:

```
char * envp[2];
envp[0] = “TERM=xterm”;
envp[1] = NULL;

exec(“/bin/echo”, argv, envp);
```

- sh may be told to pass these environment parameters to executed programs by way of the *export* command.

```
% TERM=xterm; export TERM
```



# Section #4

## Bourne Shell

## Shell Communications

- Pre-opened file descriptors:
  $ cat < f   > g
- Exec (command line) arguments:
  $ grep ‘hello’   f
- Environment parameters:
  $ PRINTER=lw; export PRINTER
  $ lpr  document



## Basic Redirection

- Direct output from file descriptor **n** to file **f**:
  **n > f**   $  2>err  ls   1>foo
  If **n** is absent, the default is the standard output (1).
- Append output from file descriptor **n** to the end of file **f**:
  **n >> f**   $ cat x >> f
  If **n** is absent, the default is the standard output (1).
- Direct input to file descriptor **n** from file **f**:
  **n < f**   $ 3<bar  foo
  If **n** is absent, the default is the standard input (0).
- Redirect standard output (1) from program 1 to the standard input (0) for program 2:
  **p1 | p2**   $ ls | grep foo



## Advanced Redirection:
### “Here” Documents

**n << word**
**n << -word**

- The shell input is read up to a line that is the same as **word**, or to an end-of-file.
- The resulting document becomes the input on file descriptor n (defaults to the standard input, 0).
- If a minus sign (-) is appended to the <<, all leading TABs are stripped.

```
# put “hello world!” into file f.
cat > f  <<-END
  hello world!
END
# done
```



## Advanced Redirection:
### dup’ing & close’ing

**n<&m   n>&m   n<&-   n>&-**

- **dup** system call:
  int fd1, fd2;
  fd1 = open(“file” O_RDWR);
  fd2 = dup(fd1);
- At the end of this sequence, **fd1** and **fd2** both refer to exactly the same thing.
- The phrase, **n>&m** or **n<&m** , causes file descriptor **n** to be a **dup** of the (pre-opened) file descriptor **m**.
- The phrase, **n<&-** or **n>&-** closes file descriptor **n**.
- The shell checks that n is open for input(<), or output(>), respectively.
- The defaults for absent **n** are stdout (1) for >, and stdin (0) for <.

# Filename Generation
## (globbing)

- Words on the command line are expanded if they contain one of the characters “*”, “?”, “[“.
- The word is replaced with a sorted list of filenames that match the given pattern.
- If no matching filenames are found, the word is left unchanged.

| | |
|---|---|
| * | Matches any string (including null). |
| ? | Matches any single character. |
| […] | Matches any one of the enclosed characters. |
| [x-y] | Matches any character pair lexically between the pair. |
| [!…] | Matches any character not enclosed. |

- The character “.” at the start of a filename or immediately following a “/” as well as the character “/” itself, must be matched explicitly.



# Shell Variables:
## setting and unsetting

- The shell maintains an internal table that gives string values to shell variables.
- A shell variable is initially assigned a value(set), or subsequently has its value changed, by a command of the form **variable**=**value**.
  $ x=3 y=4
- A shell variable is removed by the built-in command **unset**.
  $ unset x
- A shell variable can be **export**ed to the environment of commands that are executed.
  $ export x



# Shell Variables:
## retrieving

- The value of a shell variable may be substituted in a command by a “$” phrase.

| | |
|---|---|
| $var | The value of **var** is substituted. |
| ${var} | The value of **var** is substituted. (The braces are required only when **var** is followed by a letter, digit, or underscore.) |
| ${var:-w} | If **var** is set and non-null, substitute its value, otherwise substitute **w**. |
| ${var:=w} | If **var** is not set or is null, set it to **w**. The value of **var** is substituted. |
| ${var:?w} | If **var** is set and non-null, substitute its value, otherwise print **w** and exit from the shell. (Default message if **w** is absent.) |
| ${var:+w} | If **var** is set and non-null, substitute **w**, otherwise substitute nothing. |



# Shell Variables:
## positional parameters

- Shell variables that are composed of a single digit are positional parameters.

| | |
|---|---|
| $0 | 0th positional parameter. |
| $1 | 1st positional parameter. |
| … | |
| $9 | 9th positional parameter. |
| $# | The number of positional parameters as a decimal (base 10) string. |
| $* | All the positional parameters, starting with $1, are substituted (separated by spaces). |
| $@ | Similar to $*. However they differ when quoting is used (later). |

# Shell Variables:
## the “set” command

- The command

  $ set

  will print out the values of all shell variables.
- The command

  $ set a b c

  will set positional parameters 1, 2, and 3 to “a”, “b”, and “c” respectively.
- The set command with arguments starting with “+” or “-” will turn on and off the shell options. e.g.

  $ set -x

  will cause all commands and their arguments to be printed as they are executed.
- These options may also be set when invoking the shell.

  $ sh -x foo



# Shell Variables:
## pre-set

- The following shell variables are pre-set.

| | |
|---|---|
| $- | The options supplied to the shell on invocation or by the **set** command. |
| $? | The exit status returned by the last command executed in the foreground as a string in decimal notation. |
| $$ | The process ID of this shell. |
| $! | The process ID of the last background command invoked. |
| $PATH | The directories to search in order to find a command. |
| $PS1 | Primary prompt string. |
| $PS2 | Secondary prompt string. |
| $MAILCHECK | How often to check for mail. |
| $IFS | Internal field separator. |



# Environment Parameters

- The environment, a list of name-value pairs, is passed to the shell and to every command that the shell invokes.
- When the shell starts up, it makes a shell variable out of each name-value pair.
- Shell variables and environment parameters may be bound together by means of the **export** command.
- Entries in the environment may be modified or added to by binding an existing or yet to exist shell variable. Subsequent changes to that variable will be reflected in the environment list.
- Entries may be deleted by performing an unset on the corresponding shell variables.
- The environment for any simple command may be augmented by prefixing it with one or more assignments to parameters. e.g.

  $ X=5 Y=6 fooscript



# Environment Parameters
## used by sh

| | |
|---|---|
| HOME | Default argument for **cd**. (set by **login**) |
| PATH | The search path for commands. |
| CDPATH | the search path for **cd**. |
| MAIL | File where the user’s mail arrives. (set by **login**) |
| MAILCHECK | How often to check for mail. |
| MAILPATH | Set of files to check for mail. (used in preference to MAIL if set) |
| PS1 | Primary prompt string. |
| PS2 | Secondary prompt string. |
| IFS | The characters that separate arguments on a command line. |
| SHELL | If set and value contains an “r”, the shell becomes a restricted shell. (set by **login**) |

## Command Substitutions

- The standard output for a command enclosed in a pair of back-quotes (``) may be used as part or all of a word.
- Trailing newlines are removed.

```
$ echo `pwd`
/homes/u1/wayne
```



## Quoting

- The following characters have a special meaning to the shell:
  `;  &  (  )  |  ^  <  >  NL  SPACE  TAB`
- A single character may be quoted by preceding it with a backslash(\).
- A backslash(\) character followed by a newline is ignored.
- All characters enclosed between single quotes (‘) are quoted (except for (‘).
- Inside double quote marks(“) shell variable substitution and command substitution occurs. (“\” is used to quote the characters \ ‘ “ and $.

```
$*     =   $1 $2 … $n
“$*”   =   “$1 $2 … $n”
“$@”   =   “$1” “$2” … “$n”
```



## Putting it all Together

- Whenever a command is read, either from a shell script or from the terminal, the following sequence of substitutions occur:

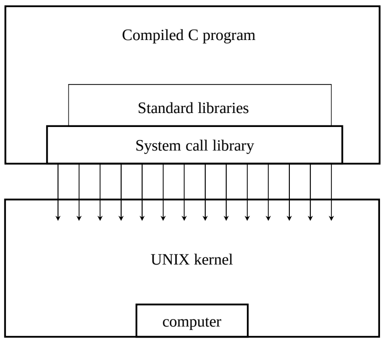
1) **Comments**
   A word beginning with the “#” causes the word and all the following characters up to the end of the line to be ignored.
2) **Command substitution**
   Commands enclosed in back-quotes are executed.
3) **Parameter substitution**
   All “$” references are expanded.
4) **Blank interpretation**
   The results up to here are scanned for characters in IFS and split into distinct arguments. Explicit nulls are retained (“”), implicit ones are removed.
5) **Filename expansion**
   Each argument is then filename expanded.
6) **I/O Redirection**
   I/O redirection is now separated from command line arguments.



# Section #5

## UNIX Program Execution

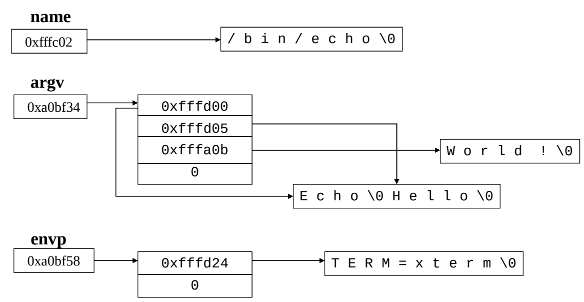
# C Program Execution






# EXECVE

execve(name, argv, envp)






# Executable Files

- **execve** will fail unless the file to execute has the appropriate execute permission bit turned on.
- The file must also be in one of the correct formats.
- There are two general classes of executable files:
  1) Executable object files (machine code and data).
  2) Files of data for an interpreter (usually ascii).



# Interpreter Files

- The UNIX kernel, during an **execve**, reads the first few bytes of a file it is asked to execute.
- *Interpreter files* begin with a line of the form:
  #! **interpreter arguments**
  e.g.
  #!/bin/sh -x
- The kernel executes the named interpreter with the name of the original (data) file as one of the arguments.
  e.g.
  execv("foo", <"foo", "a", "b", "c">)
  is transformed into:
  execv("/bin/sh", <"sh", "-x", "foo", "a", "b", "c">)
- This should explain why so many UNIX commands use '#' for a comment line indicator.

# Executable Object Files

- An executable object file has the following 7 sections:

1) header
   - magic number
   - text size
   - data size
   - bss size
   - symbol table size
   - entry point
   - text relocation size
   - data relocation size

2) text (machine code)
   - zero filled to nearest page (e.g. 8K) boundary

3) initialized data
   - zero filed to nearest page boundary



# Executable Object Files (cont.)

4) text relocation information
   - address
   - size (byte, half-word, word)
   - symbol number

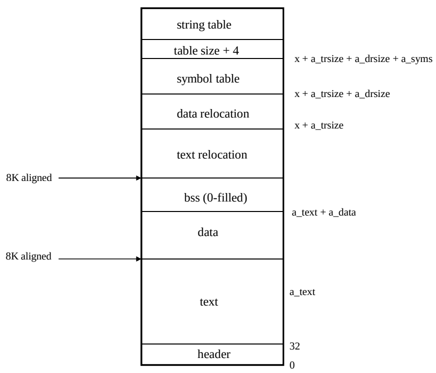
5) data relocation information
   - same as above

6) symbol table
   - index into string table
   - type of symbol
   - value

7) string table
   - size in first four bytes
   - zero-terminated strings

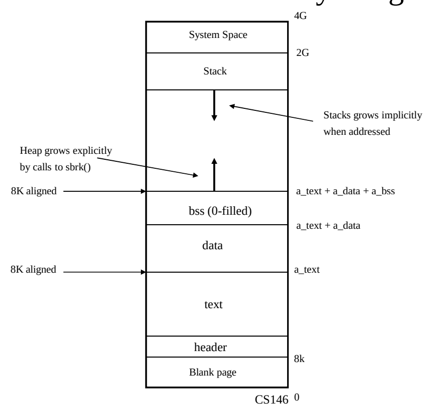


# Executable Object File Format






# Virtual Memory Image



## "Hello world!"
(in 68000 assembly language)

```
.text
movl      #13, sp@- | # bytes to write
pea msg             | address
movl      #1, sp@-  | standard output
movl      #0, sp@-
movl      #4, sp@-  | WRITE
trap      #0

addw      #20, sp   | clean the stack

movl      #0, sp@-  | exit code
movl      #0, sp@-
movl      #1, sp@-  | _EXIT
trap      #0

.data
msg:.ascii "Hello world!\12\0"
```

```
$ as hello.s
$ ld a.out
$ ./a.out
Hello world!
$
```

CS146 73

## "Hello world!"
(in C using only system call library)

```
char msg[] = "Hello world!\n";

int
main()
{
    int bytesWritten;
    bytesWritten = write(1, msg, 13);
    return 0;
}
```

```
$ gcc hello.c
$ ./a.out
Hello world!
$
```

CS146 74

## libc

- libc contains the object code for:
  - the interfaces to system calls
  - the standard libraries

- For example, the file "write.s" is that part of the source for the system call interface library that interfaces to the **write** system service.

```
err:  jmp cerror
      .globl _write
_write:
  movl    #4, sp@-
  trap    #0
  jcs     err
  rts
```

CS146 75

## Error Statuses Returned from System Calls

- Every system call returns a status.
- If the status is negative then the system call interface library will call the routine **cerror**.
- Cerror will store the error status (returned by the system call in a general purpose register) into a global variable called **errno**.

```
extern int errno;
main()
{
   int fd;
   fd = open("foo", 0, 0);
   if (fd == -1)
       fprintf(stderr, "Error
   on open: %d\n", errno);
}

Error on open: 2
```

```
#include <stdio.h>
#include <errno.h>
main()
{
   int fd;
   fd = open("foo", 0, 0);
   if (fd == EOF)
       perror("foo");
}

foo: No such file or directory
```

CS146 76

# “Hello world!”
(in C using standard I/O library)

```
#include <stdio.h>

main()
{
    printf(“Hello world\n”);
}
```

```
$ gcc hello.c
$ ./a.out
Hello world!
$
```



# Section #6

## Shell Scripting



# Shell Scripting: 1

- “ls -F” is much more useful than simple “ls”. It tells you concisely what each file is without the bother of doing “ls -l” all the time.
- We want it to be so that when we type “ls”, we get “ls -F”.
  - $HOME/bin/ls



# Shell Scripting: 1(a)

$HOME/bin/ls

```
ls –F
```

2 Things Wrong

1. Since this script version of ‘ls’ was probably run as the first ‘ls’ in the **PATH**, the ‘ls’ in the script will run the script again. Infinite recursion.
2. Arguments are being ignored. That means ‘ls /etc’ would not work as expected.

# Shell Scripting: 1(b)

$HOME/bin/ls

exec /bin/ls –F $*

A corrected version would call /bin/ls to avoid the infinite loop. The $* variable will pass the arguments to the real 'ls'. The '*exec*' avoids the shell waiting around for the completion of 'ls'.



# Shell Scripting: 1(c)

The Bourne Shell has a function syntax that can solve our problem elegantly. It can be added to the .profile startup file so it is loaded for login shells.

$HOME/.profile

ls () { /bin/ls –F $*; }

– In other shells, there is an *alias* command used like

alias ls ls –F

or

alias ls="ls –F"



# Shell Scripting: 2

- We want to set the shell prompt to be 'machine-> '
- I logon to many different machines. Often several at once from the same workstation. I want only one .profile file.
- Program "hostname" will give you the machine in the form:
  – machinename.domainname



# Shell Scripting: 2(a)

- The first approach demonstrates the use of *IFS* and *set* but it is quite convoluted. Using *set* in shell scripts has the notable drawback that arguments are destroyed and hence must be parsed first or saved for later.

- $HOME/.profile

```
oldIFS=$IFS
IFS='.'; set `hostname`; PS1="$1-> " ; export PS1
IFS=$oldIFS; unset oldIFS
```

## Shell Scripting: 2(b)

- The following version can be considered simpler. It sends the output of *hostname* through *sed* with a substitution command.

```
PS1=`hostname | sed 's/\..*//'`; export PS1
```

- The sed command is explained as follows:
  - S - sed command for substitution
  - / - delimiter for regular expression
  - \ - escape character for following character
  - . - a period. Normally, sed interprets periods as the regular expression for "any character". The previous backslash overrides that.
  - . - match any character. This one was not escaped.
  - * - match zero or more of the previous expression. In this case it means match zero or more of "any character".
  - / - separator between the regular expression and replacement part of the substitute command
  - / - the end of the replacement string. We're replacing with nothing.
- So the sed command has been asked to find a period followed by any number of characters and replace it with nothing.



## Shell Scripting: 3

- When I logon, I want to a polite greeting, customized to the time of day.

  Good morning, Arthur!

  Good afternoon, Arthur!

  Good evening, Arthur!

  Good god! What are you doing up so early?
- The *date* command will print out the current date and time.

  $ date

  Mon Jan 30 10:09:27 EST 1998



## Shell Scripting: 3(a)

$HOME/bin/greet

```
# Mon Jan 3 10:09:27 EST 1998
set `date`; IFS=':'; set $4; hour=$1
If [ $hour –lt 9 ]; then
  echo "Good god! What are you doing up so early?"
elif [ $hour –lt 12 ]; then
  echo "Good morning, Arthur!"
elif [ $hour –lt 18 ]; then
  echo "Good afternoon, Arthur!"
else
  echo "Good evening, Arthur!"
fi
```

- Time could be parsed easier using cut.

```
hour=`date | cut –c12-13`
```



## Shell Scripting: 3(b)

- *Date* has some nice options including the ability to format the output in various ways. Yes, it does pay to read the man pages.

```
case `date +%H` in
    0[0-8]       ) echo "Good god…";;
    09 | 1[01]   ) echo "Good morning, Arthur!";;
    1[2-7]       ) echo "Good afternoon, Arthur!";;
    *            ) echo "Good evening, Arthur!";;
esac
```

- I can have the greet command run upon login by adding a line to my .profile to run greet.

## Shell Scripting: 4

- List all regular files in a subtree.
- This is a recursive script that demonstrates the use of $0 to run itself without knowing the name of the script.

$HOME/bin/dtfiles

```
PATH=/bin:/usr/bin:$HOME/bin:$PATH
cd $1
for i in *
do
        if [ -f $i ]; then
            echo $i
        elif [ -d $i ]; then
            $0 $i
        fi
done
```

- With no arguments, the shell script should work on your $HOME directory. To make it work on the current directory by default, we could change the ‘cd’ command to read: **`cd {$1:-.}`**



## Shell Scripting: 5

- n! is “n factorial”
- Mathematically,
  n! = n * (n-1) * (n-2) * … * 2 * 1
- The shell scripting language does not have arithmetic. However, the expr(1) utility can do arithmetic by reading and parsing strings.
- Here are two versions of shell scripts to compute n factorial. Which do you think is better? I recommend that you try both and see.
- When evaluating how to decide which script is better, consider the number of processes forked, the number of active processes during the run, what sorts of commands are used, how many temporary files are needed, maintainability, etc.



## Shell Scripting: 5(a)

```
#!/bin/sh

if [ $# -ne 1 ]; then
   echo “Usage: $0 n” >&2; exit 1
fi

# Check to make sure the argument is a number
If echo $1 | grep ‘^[0-9][0-9]*$’ >/dev/null 2>&1; then
   :
else
   echo “Usage: $0 n” >&2; exit 1
fi

If [ $1 –eq 0 ]; then
   echo 1
else
   m1=`expr $1 – 1`
   expr $1 \* `$0 $m1`
fi
```



## Shell Scripting: 5(b)

```
#!/bin/sh
if [ $# -ne 1 ]; then
   echo “Usage: $0 n” >&2; exit 1
fi

# Check to make sure the argument is a number
If echo $1 | grep ‘^[0-9][0-9]*$’ >/dev/null 2>&1; then
   :
else
   echo “Usage: $0 n” >&2; exit 1
fi

fact=1
number=$1
Until [ $number = 0 ]
do
   fact=`expr $fact \* $number`
   number=`expr $number – 1`
done
echo $fact
```

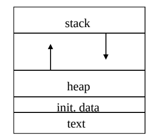
# Section #7

## C Storage Model

## Compilation and Linking



## Setting Aside Storage




- Every data element must have the appropriate number of bytes set aside for it in the program's memory.
- Insofar as variables are concerned, those bytes are either allocated on the stack, or in the heap.
- You tell the C compiler to set aside storage for you by means of a declarations:

```
int i;
unsigned short j;
```



## Stack Data

- Each time that a C function is called, extra stack space is implicitly allocated.
- This stack space contains the *automatic* variables (also called *local variables*) for that function.
- Local variables are all variables declared within a {} block.
- When a function returns, that stack space is implicitly de-allocated and later re-used.

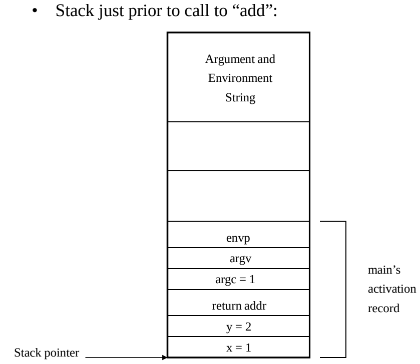
```
/* Pathological Example */
main()
{
   int x = 1;
   int y = 2;

   add(x, y);
   printSum();
}
```

```
add(int i, int j)
{
   int k = i + j;
}

printSum()
{
   int a, b, c;
   printf("%d\n", c); // or maybe a
}
```



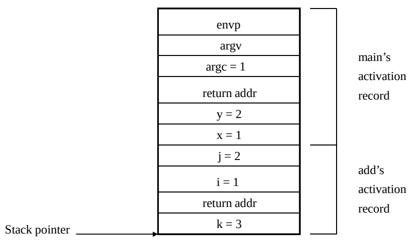
## Example explained #1

- Stack just prior to call to "add":



## Example explained #2

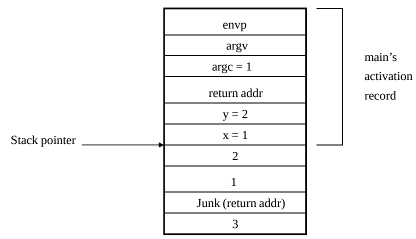
- Stack just after call to “add”:






## Example explained #3

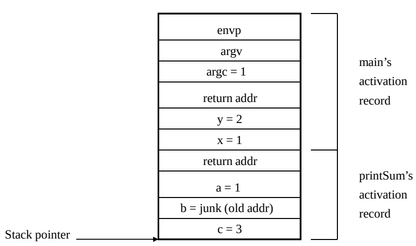
- Stack just prior to call to “printSum”:






## Example explained #4

- Stack just after to call to “printSum”:






## Heap Data

- The heap is divided into three parts:
  - initialized data
  - zero-initialized data
  - dynamically allocated data
- Space for initialized and zeroed data is allocated for each declaration appearing outside of any function (or for in-function declarations prefaced by **static**):

```
int i, j = 3;
main()
{
   static int k = 2;
   i = k + j;
}
```

- Space for dynamically allocated data is allocated explicitly by calls to the library function **malloc**.

```
main()
{
      int *p = malloc(sizeof(int));
      *p = 3;
}
```

## Storage Class

- There are various ways of specifying which storage class an object belongs to:
  - If an object is declared within a {…} block with no storage class specification, or the **auto** storage class specification, they are stored on the stack.
  - If an object declared within a block has the storage class specifier **register**, it is either kept on the stack or in a CPU register if that is possible.
  - If an object within a block has the storage class specifier **static**, it is stored in the heap, but is still semantically local to that block.
  - If an object is declared outside of all blocks, it is stored in the heap.
  - If an object is declared outside of all blocks, and has the specifier **static**, it is local to that file.
  - If an object declared outside of all blocks has the specifier **extern**, or no specifier, it is visible throughout the program.
  - If declared **extern**, no space is allocated. It is assumed that space has been allocated elsewhere (i.e. without the keyword **extern**) and will be resolved by the linker.



## C Compilation

- There are four main phases of C compilation
  - (1) Preprocess
  - (2) Scan & Parse
  - (3) Code Generation
  - (4) Linking



## Preprocess

- The preprocessor (cpp) handles macros, #include, and conditional compilation.

foo.h

```
#define DEBUG 1
#define ADD(a,b) ((a) + (b))
int x;
extern void printi(int);
```

foo.c:

```
#include “foo.h”
void main()
{
   int y, z;
   x = ADD(y, z);
#if DEBUG
   printi(x);
#endif
}
```

After preprocessing:

```
int x;
extern void printi(int);
void main()
{
   int y, z;
   x = ((y) + (z));
   printi(x);
}
```



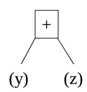
## Scan

- The scanner separates input into logical **tokens** - no meaning is assigned yet.

| | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| int | x | ; | extern | void | printi | ( | int | ) | ; | void | | | | | | |
| main | ( | ) | { | int | y | , | z | ; | x | = | ( | ( | y | ) | + | |
| ( | z | ) | ) | ; | printi | ( | x | ) | ; | } | | | | | | |

# Parse

- The parser derives meaning from the stream of tokens. Syntax checking also occurs here.
- **x** is a global integer initialized to zero (bss segment).
- **main** is a void subroutine with no parameters.
- **{** marks the beginning of main.
- **int y, z;** defines two automatic, uninitialized integers.
- **x = ((y) + (z));** is an expression described by a parse tree;




- **printi(x);** call printi with x as argument.
- **}** marks the end of main.



# Code Generation

- Code generation takes the parsed program (i.e. the compiler now “understands” the program) and generates machine language. We’ll show it as assembly language. Most compilers generate text for an assembler instead of generating code directly.
- Assign x an address, say memory locations 100-103.
- Assign main a starting address, say 1000.
- 68000 assembly language representation of compiled code:

```
_x          = 100
 main       = 1000
   add.l    -8, sp            ; 2 ints, y & z
   move.l   @sp, @_x          ; x = y
   add.l    @sp(4), @_x       ; perform addition y+z
   move.l   @_x, @-sp         ; push x onto stack
   jsr      _printi           ; unresolved link
   add.l    12, sp            ; clean stack
   rts                        ; return from _main
```

- Actual machine language file is called *object file* “foo.o”



# Link

- Linking is the resolving of symbols in object files.
- Each object file has associated with it a list of **<name, address>** pairs called a symbol table.
- Names not defined in the file, called *unresolved references*, have a NULL address. The symbol table for foo.o is:

  [ <_main, 1000>, <_x, 100>, <_printi, 0> ]

  Note y, z do not appear since they are local to main().
- A *library archive* (file extension .a) is a collection of object (.o) files, each containing executable machine code, global data, and a symbol table. Library archives are maintained by **ar**(1).
- “Linking” entails combining multiple object and library files, resolving all unresolved references, and producing an a.out executable file.
- In our example, we assume _printi is resolved by a symbol in an object file in a standard library.
- Sometimes linking happens later, at runtime, using *shared* or *dynamically linked* libraries (DLLs in Windows, .so files in Unix)



# Link example

```
$ gcc -E foo.c      # pre-process only, output to stdout
$ gcc -S foo.c      # PP, scan, parse, produce assembly language file foo.s
$ gcc -c foo.c      # PP, scan, parse, codegen, produce output file  foo.o
$ gcc foo.o         # link foo.o to produce a.out
$ gcc foo.c         # all 4 phases, produce a.out
$ gcc -c foo1.c     # produce foo1.o
$ gcc -c foo2.c     # produce foo2.o
$ gcc foo1.o foo2.o         # link foo1.o and foo2.o to libraries to produce
                            # a.out
```

- If necessary, the linker moves addresses at link time to avoid address conflicts (e.g. foo1.o and foo2.o both claim address 100 for different variables)
- On some systems, the symbol table also includes type information, e.g. x is an int and printi is a function. Most modern UNIX systems do this.

# Makefiles

- A Makefile contains instructions telling make(1) what depends on what, and how to build things. Make(1) looks at timestamps and figures out how to build things that don't exist or are out-of-date.
- Each section of a makefile looks like:

```
target1: [dependency list] # empty mean always rebuild
     instructions            # MUST be TAB indented.
```

- Sections are separated by blank lines. e.g.:

```
$ cat makefile
foo: foo1.o foo2.o
     gcc -o foo foo1.o foo2.o

foo1.o: foo1.c foo.h
     gcc -c foo1.c

foo2.o: foo2.c foo.h
     gcc -c foo2.c
```

- Typing "make" causes the first target in the Makefile to be built. Typing "make foo1.o" causes a specific target to be built.



# Section #8

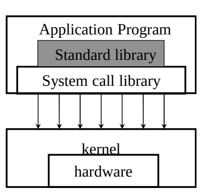
## Standard Library

"Never code something that someone else has already coded better."



# Standard Library

- There is more in the standard library than you might expect. (Read "man intro" and lookup the intro's for sections starting with 3.)
- Library contains functions, variables, and macros.
- Some library calls perform system calls, others do not. The system calls interface routines themselves are not considered part of the standard library (See "man 2 intro"). They are simply C interfaces to the system calls.






# Standard Library

- Library is divided into a number of different parts (see /usr/include)

| | |
|---|---|
| stdio | User-level buffered file I/O |
| errno | Checking return status of system calls |
| malloc | Memory allocation |
| ctype | Classifying ASCII-coded integers |
| string | Operations on null-terminated strings |
| math | Mathematical functions and macros |
| exit | Normal and abnormal termination |
| getenv | Accessing environment variables by NAME |
| qsort | Sorting |
| bsearch | Binary search |
| assert | Diagnostics used for debugging |
| stdarg | Accessing variable length function parameter lists |
| setjmp | Non-local program control jumps |
| signal | Handling UNIX signals (also called exceptions) |
| limits | Implementation-dependent information |
| float | Implementation-dependencies for floating point |
| random | Random number generation |
| time | Dealing with date and time |
| network | Accessing networks |
| encrypt | DES encryption |
| dbm | Database routines (key-content pairs) |
| dir | Directory operations |
| getopt | Parse options in argv |
| regex | Regular expression handlers |
| stty | Setting terminal driver characteristics |
| system | Performing shell commands |

… and more ...

# Standard I/O

- Designed to make performing I/O convenient and efficient.
- I/O is done independently on independent streams.
- To use:

```
#include <stdio.h>
```

which defines (among other things):

| | |
|---|---|
| FILE | Stream struct |
| NULL | No stream |
| EOF | End-of-file or error return indicator |
| stdin | Standard streams |
| stdout | |
| stderr | |



# Opening & Closing Streams

- FILE *fopen(char *filename, char *mode)
  - Opens “filename” for access according to “mode”.
  - Mode can be one of “r”, “w”, “a”, “r+”, “w+”, “a+”
- FILE *freopen(char *filename, char *mode, FILE *stream)
  - Substitutes the named file in place of the open stream. The old stream is closed.
- FILE *fdopen(int fildes, char *mode)
  - Opens a stream that refers to the given UNIX file descriptor (must currently be open).
- int fileno(FILE *stream)
  - Returns the UNIX file descriptor associated with the stream.
- int fflush(FILE *stream)
  - Causes any buffered data for the named stream to be written out.
- int fclose(FILE *stream)
  - Flushes the stream, closes the file, and deallocates the FILE data structure.
- int exit(int status)
  - Causes all open streams to “fclose”d calls _exit(status).



# Output Buffering Modes

- There are three kinds of output buffering modes for streams:
  1) Unbuffered - Characters appears on the terminal or in the file as soon as they are written.
  2) Block Buffered - Many characters are saved up and then written as a block.
  3) Line Buffered - Characters are buffered until a newline is encountered or input is read from stdin.
- Normally all files are block buffered, except terminals which normally default to line buffered for **stdout**, and **stderr** which is always unbuffered.

- int setbuffer(FILE *stream, car *buf, int size)
  - Specifies that “buf” be used rather than a malloc’d buffer on the first **getc** or **putc** and sets the buffer size to “size”. If “buf” is NULL, I/O will be unbuffered. Used after a stream is opened, but before it is read or written.
- int setbuf(FILE *stream, char *buf)
  - Same as setbuffer(stream, buf, BUFSIZ).
- int setlinebuf(FILE *stream)
  - Used to change **stdout** or **stderr** to line buffered. Can be called anytime.



# Low-Level Input

- int getc(FILE *stream)
  - Returns the next character from “stream”. (macro - beware of side effects)
- int ungetc(int c, FILE *stream)
  - Pushes the character “c” back onto “stream”. Returns c.
- int getchar()
  - Identical to getc(stdin).
- int fgetc(FILE *stream)
  - Same as getc, but not a macro.
- int getw(FILE *stream)
  - Returns the next int from “stream”. (must check for errors)
- char *gets(char *s)
  - Reads characters up to and including the next newline into “s” from stdin. The newline is replaced by a NULL character in s. Returns s. This is VERY dangerous (see Internet Worm).
- char *fgets(char *s, int n, FILE *stream)
  - Reads n-1 characters or up to and including a newline from “stream” into “s”. Adds a null character onto the end. Returns s.
- int fread(void *ptr, size_t size, int nitems, FILE *stream)
  - Reads “nitems” nto block pointed to by “ptr” from “stream”. Flushed stdout if stream is stdin. Returns # items read.

# Formatted Input

- int sscanf(char *s, char *format [, pointer] …)
  - Parses "s" according to "format" placing the results into the variables pointed to. Returns number of input items parsed and assigned.
- int fscanf(FILE *stream, char *format [, pointer] …)
  - Same as sscanf but read from "stream".
- int scanf(char *format [, pointer ] …)
  - Same as fscanf(stdin, format, …)
- "format" is composed of:
  - Blanks, tabs, newlines: Match optional white space.
  - Regular characters (not %): Must match input.
  - % [*] [maxField] [convChar]: Conversion specification.
- The conversion characters are:
  - % — Matches a % characters
  - d, D, ld, hd — Decimal interger
  - o, O, lo, ho — Octal integer
  - x, X, lx, hx — Hexadecimal integer
  - s — Character string
  - c — Single character
  - e, E, le
  - f, F, lf — Floating point number



# Low-Level Output

- int putc(char c, FILE *stream)
  - Appends "c" to "stream". Returns the character written. (macro)
- int putchar(char c)
  - Same as putc(c, stdout)
- int fputc(char c, FILE *stream)
  - Same as putc, but not a macro.
- int putw(int w, FILE *stream)
  - Appends int "w" to "stream". Returns the word written.
- int puts(char *s)
  - Appends the null-terminated string "s" to stdout, and a newline character.
- int fputs(char *s, FILE *stream)
  - Appends the null-terminated string "s" to "stream".
- int fwrite(void *ptr, size_t size, int nitems, FILE *stream)
  - Append at most "nitems" of data of type *ptr beginning at "ptr" to "stream". Returns # of items written. (returns 0 for error)



# Formatted Output

- int sprintf(char *s, char *format [, pointer] …)
  - Places "format" expanded using "args" into the string "s".
- int fprintf(FILE *stream, char *format [, pointer] …)
  - Same as sprintf but appends to "stream".
- int printf(char *format [, pointer ] …)
  - Same as fprintf(stdin, format, …)
- "format" is composed of:
  - Regular characters that are copied verbatim
  - Conversion specifications of the form
    - % [flags] [fieldWidth] [.] [precision] [l] [type]
- Flags are:
  - # — Alternate form
  - - — Left alignment
  - + — Include a sign if appropriate
  - space — blank should be left before a positive number (i.e. leave space for the +)
- Types are:
  - % — Print a % character
  - d, o, x — Decimal, octal, or hex integer
  - f — Float or double
  - e — Float or double with exponent
  - g — Style d, f, or e whichever simplest gives full precision.
  - c — character
  - s — string
  - u — unsigned integer



# Positioning a Stream Pointer

- int fseek(FILE *stream, long offset, int whence)
  - Sets the position of the next I/O on "stream". The new position is at a signed "offset" from the beginning, current position, or the end-of-file, according as "whence" is 0 (SEEK_SET), 1 (SEEK_CUR), or 2 (SEEK_END). This undoes an **ungetc**.
- long ftell(FILE *stream)
  - Returns the current value of the file pointer for "stream"
- int rewind(FILE *stream)
  - Same as fseek(stream, 0L, 0)

# Status Enquiries

- int feof(FILE *stream)
  - Returns 0 iff no end-of-file was encountered.
- int ferror(FILE *strream)
  - Returns 0 iff no error has occurred while readng or wrting this stream.
- void clearerr(FILE *stream)
  - Resets the end-of-file and error indicators for this stream.



# String/Character Handling

- All "str" functions require input strings be terminated with a null byte.
- Some of the most common ones:
  **`strlen`**, **`strcpy`**, **`strcmp`**, **`strcat`**
- **`memcpy`** not just for strings!
- Some function for testing/converting single characters (ctype.h):
  **`isalpha`**, **`isdigit`**, **`isspace`**
  **`toupper`**, **`tolower`**
  **`atoi, atol`**



# Storage Allocation

- Dynamic memory allocation (heap storage!):
  **`malloc`**, **`calloc`**, **`free`**, **`realloc`**
- An example:

```
#include <stdio.h>
#include <malloc.h>
struct xx *sp;
main() {
    sp = (struct xx *) malloc( 5 * sizeof(struct xx)
      );
    if( !sp ) // if (sp == NULL)
    {
       fprintf(stderr, "out of storage\n" );
       exit( -1 );
    }
}
```



# Date and Time Functions

- Most UNIX time functions have evolved from various sources, and are sometimes inconsistent, referring to time as one of:
  - the number of seconds since Jan 1, 1970 (or Jan 1, 1900)
  - the number of clock ticks since Jan 1, 1970 (or Jan 1, 1900)
  - the broken down structure "**`struct tm`**" (see **`/usr/include/time.h`**)
  - the broken down structure "**`struct timeval`**" (see **`/usr/include/sys/time.h`**)
- Some are intended for time/date, whereas others are intended for measuring elapsed time

## Environment Interfacing

- Reading environment variables:
  char * getenv(char *envname);
- Adding environment variables:
  int putenv(char *string);
  where string is of the form `name=value`.

- Executing a shell command:
  system(“egrep 128 /etc/hosts | wc”);
  (What are the disadvantages of running a command this way?)



## Convenient Subshells

- You can also execute a command via the shell and have its output sent to a pipe instead of stdout:

```
FILE *rpipe, *wpipe;
rpipe = popen( “ls -atl”, “r” );
... // read stuff from rpipe ...
pclose( rpipe );
wpipe = popen (“cat > foo”, “w”);
... // write stuff to wpipe ...
pclose( wpipe);
```

- Note that popen(3) is a standard library call that provides a convenient method of taking advantage of the pipe(2) system call.



# Section #9

## UNIX System Calls



## UNIX System Calls

- Kernel primitives
  - Processes and protection
  - Memory management
  - Signals
  - Timing and statistics
  - Descriptors
  - Resource controls
  - System operation support

- System Facilities
  - Generic operations
  - File system
  - Interprocess communications
  - Terminals and devices
  - Process control and debugging

## Host & Process Identifiers

- A *HOST* refers to the name of the UNIX installation on which a program runs.
- Each UNIX host associated with it a 32-bit host-id, and a host name. These can be set (by the superuser) and returned by the calls:
  - int status = sethostid(long hostid);
  - long hostid = gethostid();
  - int status = sethostname(char *name, int len);
  - int len = gethostname(char *buf, int buflen);
- On each host runs a set of *processes*, each of which is identified by an integer called the *process id*.
  - int pid = getpid();



## Process Creation & Termination

- A new process is created by making a logical duplicate of an existing process:
  int pid = fork();
- The **fork** call returns twice, once in the parent process, where *pid* is the process identifier of the child, and once in the child process where the *pid* return value is 0.
- A process can overlay itself with the initial memory image of another program, passing the newly started program a set of parameters:
  int status = execve(char *name, char **argv, char **envp);
  (Note that including the types above like "char **" are not correct syntax.)
- A process may terminate by executing:
  void exit(int status);
  returning 8 bits (low-order) of exit status to its parent.
- A process may also terminate *abnormally*.



## Termination Reporting

- When a child process terminates, the parent process may elect to receive information about the event which caused termination of the child process.
  int wait(union waitstatus *waitstatus)
- There are three possibilities:
  1) No children
     - ERROR
  2) One or more dead children (zombies)
     - Call returns immediately with the status of one of the zombies chosen at random (thus burying it).
  3) No dead children
     - Call blocks until there is one, then does #2.
- An additional non-blocking call returns the same information as wait, but also includes information about resources consumed during the child's lifetime.
  int wait3(union waitstatus *astatus, int options, struct ruasge *arusage);



## User & Group ID's

- Each process in the system has associated with it a:
  - *real user id*
  - *effective user id*
  - *real accounting group id*
  - *effective accounting group id*
  - *set of access group ids*
- These are returned by:
  int ruid = getuid();
  int euid = geteuid();
  int rgid = getgid();
  int egid = getegid();
  int ngrps = getgroups(int gsetsize, int gidset[gsetsize]);
- The user and group id's are assigned at login time using:
  int status = setreuid(int ruid, int euid);
  int status = setregid(int rgid, int egid);
  int status = setgroups(int gsetsize, int gidset[gsetsize]);
- Unless the caller is superuser, ruid/gid must be equal to either the current real or effective user/group id.
- The **setgroups** call is restricted to the superuser.

## Process Groups

- Each process in the system is normally associated with a *process group*.
- The group of processes in a process group is referred to as a *job*, and manipulated by system software (such as the shell).
- The process group of a process is returned by:
  int pgrp = getpgrp(int pid);
- When a process is in a specific process group it may receive software interrupts affecting the group (causing it to suspend or resume execution, to be interrupted, or to be terminated).
- The process group associated with a process may be changed by:
  int status = setpgrp(int pid, int pgrp);
- Newly created processes are assigned process id's distinct from all processes and process groups, and inherit pgrp.
- A non-superuser process may set its process group equal to its process id.
- A superuser process may set the process group of any process to any value.

CS146 133

## Memory Management

- Each process begins with three logical areas of memory called text, data, and stack.
  - The text area is read-only and shared.
  - The data and stack areas are private to a process.
- The stack area is automatically extended as needed.
- The data area is extended and contracted on program request by the call:
  void *newBreak = sbrk(int incr);
- The size is actually changed by units of pagesize, whose CPU-dependent value is returned by:
  int pagesize = getpagesize();

CS146 134

## Time Zones

- The system's notion of the current UTC (Universal Coordinated Time, formerly Greenwich Mean Time), and current time zone is set and returned by:

```
#include <sys/time.h>
 struct timeval {
     long tv_sec; /* seconds since Jan 1, 1970 */
     long tv_usec; /* and microseconds */
 };
 struct timezone {
     int tz_minuteswest; /* of UTC */
     int tz_dsttime; /* type of dst correction */
 };
 int status = settimeofday(struct timeval *tvp, struct timezone *tzp);

 int status = gettimeofday(struct timeval *tvp, struct timezone *tzp);
```

CS146 135

## Inter-Process Communication (IPC)

- Data exchange techniques between processes:
  - Data stream exchange: files, pipes, sockets
  - Shared-memory model
  - Signals
- Limitations of *files* for inter-process data exchange:
  - Slow!
  - One typically must finish writing a file before the other process reads it.
  - Could create LARGE temporary files.
- Limitations of *pipes*:
  - Two processes must be running on the same machine
  - Two processes communicating must be "related"
- Sockets overcome these limitations but are more complicated(*we'll cover sockets later*).

CS146 136

## dup(2) and dup2(2)

```
newFD = dup( oldFD );
if( newFD < 0 ) { perror("dup"); exit(1); }
```

or, to force the newFD to have a specific number:

```
returnCode = dup2( oldFD, newFD );
if(returnCode < 0) { perror("dup2"); exit(1);}
```

- In both cases, **`oldFD`** and **`newFD`** now refer to the same file
- For **`dup2()`**, if **`newFD`** is open, it is first automatically closed
- Note that **`dup()`** and **`dup2()`** refer to fd's and *not* streams
  - A useful system call to convert a stream to a fd is
    **`int fileno( FILE *fp );`**



## `pipe()`

- The **`pipe()`** system call creates an internal system buffer and two file descriptors: one for *reading* and one for *writing*
- Pipes are FIFO(First In, First Out) constructs.
- With a pipe, typically you want the **stdout** of one process to be connected to the **stdin** of another process … this is where **`dup2()`** becomes useful.
- Usage:
  **`int fd[2], status;`**
  **`status = pipe( fd );`**
  /* **`fd[0]`** for reading; **`fd[1]`** for writing */
  If(status < 0) perror("pipe");



## pipe()/dup2() example

```
/* equivalent to "sort < file1 | uniq" */
int fd[2];
FILE *fp = fopen( "file1", "r" );
dup2( fileno(fp), fileno(stdin) );
fclose( fp );
pipe( fd ); // populates both fd[0] and fd[1]
if( fork() != 0 ) { // Parent
   dup2( fd[1], fileno(stdout) );
   close( fd[0] );  close( fd[1] ); // DON'T FORFGET THIS!
   execl( "/usr/bin/sort", "sort", (char *) 0 );  exit( 2 );
} else { // child
   dup2( fd[0], fileno(stdin) );
   close( fd[0] );  close( fd[1] );
   execl( "/usr/bin/uniq", "uniq", (char *) 0 );  exit( 3 );
}
```



# Section #10

## Debugger (gdb)

## Debugging

- A *debugger* is a program that runs other programs in a controlled environment so that you can execute the program line-by-line, view and modify variables, set *breakpoints* to stop execution at specified points in the code, and *watchpoints* which will stop execution anywhere when the value of a variable changes. As such, a debugger is perhaps more aptly called a *bug finder*.
- By default, an a.out file contains the symbol tables of all the object files it was made from.
- More info, like line numbers and variable types, can be inserted into an object(.o) file by compiling with debugging turned on (the -g flag for most UNIX compilers). These extra symbols are conveyed from the object file to the a.out executable.



## ptrace

- Debugging is initiated by the **ptrace** system call.
- Generally, the debugger does a **fork**, and the child enables itself to be debugged by calling ptrace. Without this, the parent would not be allowed to debug the child. Then the child exec's the program to be debugged.
- Using ptrace, the parent can examine and modify any memory location of the child. By looking at the child's symbol table (in the a.out file), the parent can examine the child's memory that corresponds to variable names.



## How ptrace works

- A process that has executed ptrace(0) (e.g. the child of the debugger before it exec's the program) treats signals differently than a normal process.
- It also has a writable text segment (text segment is usually readonly)
- It executes normally until it receives a signal, at which time it stops, and the parent is notified via the **wait** system call.
- The parent may then use **ptrace** to examine and modify the child's memory (including the text segment).
- The child remains stopped until the parent orders it to continue by calling **ptrace**. The parent can clear the signal before continuing the child, so the child never actually "sees" the signal unless the parent wishes it.



## Breakpoints

- Since the parent can modify any memory location, it can change the code (text segment) of the child.
- For example, before (re)starting the child, the parent can insert code to generate a SIGSEGV at a specific location, for the sole purpose of stopping the child at the location.
- This called "inserting a breakpoint."
- When the child executes that code, it gets a SIGSEGV, causing it to stop. The parent can then examine the child. To clear a breakpoint, the parent re-writes the original code before ordering the child to continue.

## Examples

```
$ gcc -g foo.c # using “-ggdb” adds even more info
$ gdb a.out
(gdb) break main
Breakpoint 1 at 0x10453; file foo.c; line 9
(gdb) cond 1 (argc > 1)
(gdb) run bar
<break in function main(), line 9 of foo.c; argc=2,
   argv=<“a.out”, “bar”>
(gdb) print argc
$1 = 2
(gdb) print argv[1]
$2 = “bar”
(gdb) whatis argc
type = int
(gdb) cont
(continuing)
```



## Stack Frames

- A stack frame contains all the information pertinent to a function call - local (automatic) variables, parameters, return address, etc.
- A new stack frame is created each time a function is called at run time and discarded when the function returns.
- After hitting a breakpoint, the debugger can examine the current stack frame (using ptrace), or any stack frame “above” it.
- The stack frame above the current one belongs to the function that called the current one, etc.
- The debugger can identify the function that called the current function by searching for the function that contains the return address in the stack frame.



## Other debugger commands

- **backtrace** - show the current list of stack frames
- **step** - execute a single piece of code (could be part of a line), descending into functions.
- **next** - execute a single line, call but do not descend into functions.
- **[return]** - re-execute the previous debugger command.
- **help** - get online help.

- gdb commands have shortforms(bt, s, n, b, p) which save on typing.

- Note that **gdb** is the GNU Debugger used for debugging programs written using **gcc/g++** (the GNU C & C++ compilers). The classic compiler program **cc** (usually pre-ANSI K&R C) uses the **dbx** debugger. **dbx** has a different set of commands. Some systems have cc configured to point to gcc or some other vendor compiler.



# Section #11

## Concurrency I
## Concepts

## Why Concurrency?

- Speed
  - Concurrent programs can often overlap activities such as disk or network I/O with CPU processing.
  - Threads can take advantage of multiprocessor machines.
- Structure
  - Sometimes the code is cleaner when concurrent programming methods are used because the high-level structure is more apparent.
  - The tasks being performed could be mostly independent activities. For example, a simulation of boats entering a harbour. Each boat could be an independent process that acts of its own accord but still must interact with other boats/processes.
  - Another example is a spreadsheet: a computationally intensive re-calculation could be happening in the background, while the user interface continues to respond (does Excel do this?)



## Concurrency Topics

- Asynchronous Activities
- Concurrency vs Parallelism
- Uniprocessor vs Multiprocessor
- Threads
- Thread-safe code and Reentrancy
- Race conditions
- Deadlock
- Dining Philosophers
- Starvation
- Concurrency Control (Execution Context, Scheduling, Synchronization)
- Critical Sections
- Synchronization Primitives (Mutexes, Semaphores, Monitors, and Message Passing)
- Programming Approaches (Pipes, Work Crew, Client/Server)



## Asynchronous Activities

- Operations which can proceed independently of each other are *asynchronous*.
- Life is asynchronous. Except for the occasional synchronization events.
  - You are breathing, listening, reading, and writing at the same time.
- In contrast, most contemporary CPUs employ generally synchronous designs at the electrical level.
  - CPU instructions proceed along a fixed timetable dictated by the clock speed of the CPU.
  - Memory access happens in a precise manner governed by the clock and the times to fetch memory values.



## Concurrency vs Parallelism

- Concurrency describes the *appearance* of simultaneous operations.
- The CDF workstations are single-processor systems. Only one process can be running on the cpu at any given time. Yet, the multitasking ability of UNIX allows it to appear as though many processes are running at the same time.
- Often times concurrency can provide benefits even on a single cpu machine because processes can be waiting on external events such as disk I/O.
- *Parallelism* can occur on a multi-processor system. You have true parallelism when operations are actually proceeding simultaneously.
- Concurrency is logical parallelism or the illusion of parallelism.

# Uniprocessor vs Multiprocessor

- A *uniprocessor* machine is a computer with a single programmer-visible execution unit. Modern processors employ superscalar designs that allow multiple instructions to be executed at the same time. However, a computer with one of those processors is still a uniprocesor machine.
- *Multiprocessor* machines come in various shapes and sizes.
- The most common is the Symmetric MultiProcessor (SMP). These machines have more than one processor on the same type sharing access to a common pool of memory. It is symmetric because all of the cpus have equal access to all of the memory.
- Another type of multiprocessor computers are the Non-Uniform Memory Access(NUMA) machines. They have unequal access to memory usually arranged in some form of hierarchy so that local memory is fast and memory local to other cpus is slower.
- Cluster machines are gaining in popularity. They are composed of several computers on a network.



# Threads

- Threads are conceptually just independent processes. However, they can share address spaces (ie code & data).
- Multiple threads can be sharing data and code, running on an SMP machine, and each thread could be working independently on different sections of the data.

- Threads are often called *lightweight processes*. This is because the UNIX kernel does not have to change everything when switching between two threads of the same program. The code&data caches can be left alone as well as things like the memory management unit content and file descriptor table. By avoiding all of that work, it is possible for UNIX to provide less overhead for context switches between related threads.



# Thread-safe code and Reentrancy

- *Thread safety* refers to the ability of multiple threads being able to call the thread-safe code without damaging results.
- Thread-safe does not imply anything about efficiency. Only correctness.
- The best thread-safe code is *reentrant*.
- A reentrant routine is coded in such a way that independent threads can call it without getting in each other's way. This is usually done by eliminating the use of static data, and arranging for the caller to pass in a context structure which keeps track of things.
- An example of this would be strtok_r(3). strtok(3) is a useful routine for simple token parsing of C strings. However, it is not thread-safe because it uses static data between calls to remember where it left off.

  ```
  char * strtok(char *s1, const char *s2);
  ```
- The strtok_r(3) is a reentrant version of strtok(3) that allows the caller to pass in the context.

  ```
  char * strtok_r(char *s1, const char *s2, char **lasts);
  ```



# Race Conditions

- Race Conditions are situations where the outcome depends on the order that processes run.
- This is the situation when using fork(). Code that depends on whether the parent runs first after the fork() or the child runs first will create a race condition.
- Unexpected results can occur when an unforeseen execution order takes place because a race went the “wrong” way.

- We usually like to have a *deterministic* result. That is, a process produces the same result on the same input.

- A process whose output cannot be predicted is *non-deterministic*. This is usually not desirable.

## Race Condition Example

| Thread Observer | Thread Reporter |
|---|---|
| loop | loop |
| observe event | print count |
| count := count + 1 | count := 0 |
| end loop | end loop |

| Scenario #1: | Scenario #2 |
|---|---|
| observer sets count = 6 | observer loads 21 from count |
| reporter prints 6 | reporter prints 21 |
| observer sets count = 7 | reporter sets count = 0 |
| reporter sets count = 0 | observer add 1 to local copy in cpu register getting 22 |
| *Lost one event!* | observer sets count = 22 |
| | *22 Extra events!* |

**Moral: Avoid updating shared data! Think about the possible chaos of updating a linked list from multiple threads!**



## Critical Sections

- A *critical section* is an area of code or data that depends on there being only one process inside at any one time for correct operation to take place.
- In UNIX, we generally do not have self-modifying code (Why?) so critical sections are critical data regions.
- A critical section must have a way to ensure that only one process is inside at any time. This is called **Mutual Exclusion**.
- In general, we desire **Progress**. If no process is in its critical section, then only processes wishing to enter their critical sections participate in the decision as to whom will next enter their critical section, and this decision must occur in finite time.
- We also desire **Bounded Waiting**. No process should ever be denied access to its critical section forever while other processes enter and exit their critical sections. If this occurs, it is called **starvation**.
- No assumptions are made about the relative speeds of the processes.
- We will discuss how to implement mutual exclusion later.



## Deadlocks

- A concurrent program is in deadlock if all processes are waiting for some event that will never occur
- Typical deadlock pattern:
  - Process 1 is holding resource X, waiting for Y
  - Process 2 is holding resource Y, waiting for X

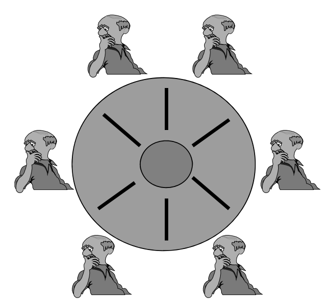
Process 1 will not get Y until Process 2 releases it

Process 2 will not release Y until it gets X, which Process 1 is holding, waiting for …



## Dining Philosophers

- **N** philosophers are seated in a circle, one chopstick between each adjacent pair.
- Each philosopher needs two chopsticks to eat, a left chopstick and a right chopstick.
- A typical philosopher alternates between eating and thinking.



## Philosopher Process

loop
<get one chopstick>
<get other chopstick>

<eat>

<release one chopstick>
<release other chopstick>

<think>

endloop



## Deadlock Example

- For **N**=2, call philosophers P1 and P2, and chopsticks C1 and C2
- Deadlocking sequence:
  P1 requests; gets C1
  P2 requests; gets C2
  P1 requests; WAITS for C2
  P2 requests; WAITS for C1

  ** DEADLOCK **

- Can avoid deadlock if the philosopher processes request both chopsticks at once, and then the get both or wait until both are available.



## Comments on Deadlock

- In practice, deadlocks can arise when waiting for some reusable resources. For example, an operating system may be handling several executing jobs, none of which has enough room to finish (and free up memory for the others)
- Operating systems may detect/avoid deadlocks by:
  – checking continuously on requests for resources
  – refusing to allocate resources if allocation would lead to a deadlock
  – terminating a process that is responsible for deadlock
- One can have a process that sits and watches, and can break a deadlock if necessary. This process may be invoked:
  – on a timed interrupt basis
  – when a process wants to queue for a resource
  – when deadlock is suspected (i.e.: CPU utilization has dropped to 0)
- Deadlock can be detected with a simple graph algorithm (Prof. Ric. C. Holt).



## Process Starvation

- *Starvation* occurs when a process is blocked waiting for an event that can, but will not occur in some future execution sequence.
- This may arise because other processes are “ganging up” on a process to “starve” it.
- During starvation (or indefinite postponement), the overall system does not grind to a halt, but treats some of its processes unfairly.

- Starvation can be avoided by having priority queues which serve concurrent processes on a first-come, first-served basis.

- UNIX *semaphores* do this, using a FIFO (first-in, first-out) queue for all requests.

# Section #12

## X Window System



## What is X?

- The X Window System (it can correctly be called X11 or X) is all of these:
  - a protocol between two processes
  - a system that defines window operations, low-level graphical rendering commands, and input request commands
  - a device-independent, portable window system
  - a network-transparent window system



## X History

- At one time there was the “W” windowing package developed at Stanford (Paul Asente).
- X was developed jointly by MIT’s Project Athena and Digital Equipment Corporation, with others also contributing.
- X Version 10 Release 4 (X10.4) was released in 1986 but was soon superceded.
- X11R1 was released in Sept 1987.
- The current version is X11R6 but many are still using X systems based on X11R4 or X11R5.

- X is a network-based windowing system. It was designed to work between many different computers.



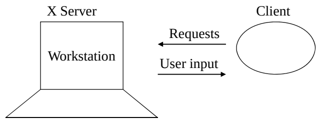
## X Servers

- The **X Server** is program that controls some of the “limited resources” on a machine: the display, keyboard, and pointer(eg mouse).
- A server:
  - Allows access to the display by multiple clients
  - Interprets network messages from clients
  - Forwards user input to clients
  - Handles [graphics] requests
  - Allocates resources
  - Maintains complex data structures (windows, cursors, fonts, graphics contexts)

- An X Server is somewhat unusual because it defines a *display* to have one or more *screens*.

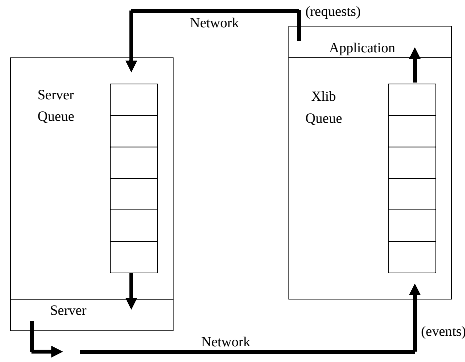
# X Clients

- An X Client is any application that connects to the X server. Any program that uses the screen or gets information from the user is an X client.
- A client:
  - Makes requests to the server (eg draw a line)
  - Processes messages from the server (usually user events)





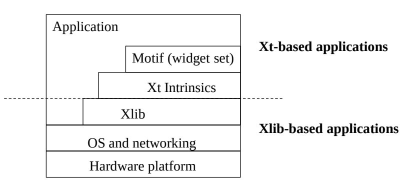

# X Client/Server Model






# X Application Architecture

- Xt-based applications can deal directly with all the layers of X.
  - the X library
  - the Intrinsics
  - and the widget set.
- Xlib-based applications can deal directly with the X library layer of X only.
- Motif is an Xt-based widget toolkit.

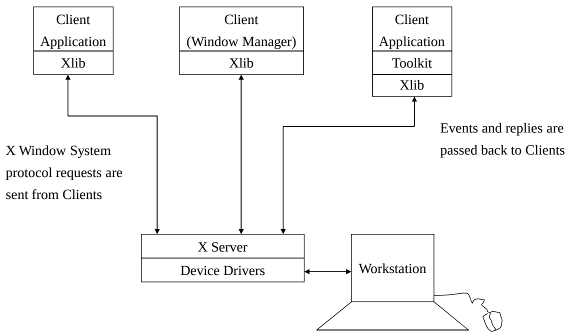





# Widgets

- When using the Xt Toolkit, the ‘things’ in the toolkits are widgets.
  - A widget is an interface object that conforms to the Toolkit Intrinsics API.
- It is a user interface building block; it has a particular job and knows how to do it.
- Examples of widgets:
  - List
  - Button
  - Form/Layout
  - Text Box
  - Scrollbar
  - Label

## X Window System Architecture






## Window Manager

- The window manager is just another X Client written using the X library. It is given special authority by convention to control the layout of windows on the display.



## XTerm

- Xterm is just another client app. It is **NOT** a shell.
- An Xterm creates a virtual terminal that a shell believes to be a character terminal like any other physical terminal hooked up via a serial cable.



# Section #13

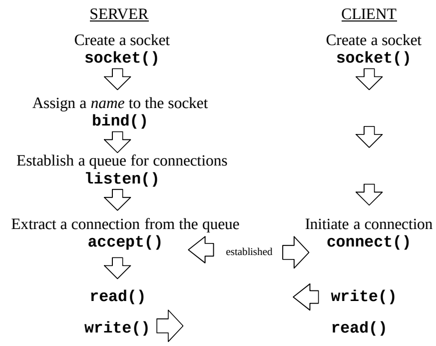
## Sockets, select(2), misc.

## Sockets

- Sockets are an extension of pipes, with the advantages that the processes don't need to be related, or even on the same machine.
- A socket is like the **end point** of a pipe -- in fact, the UNIX kernel implements pipes as a pair of sockets.
- Two (or more) sockets must be connected before they can be used to transfer data.
- Two main categories of socket types … we'll talk about both:
  - the UNIX domain: both processes on same machine
  - the INET domain: processes



## Connection-Oriented Paradigm






## Multiplexed I/O

- Consider a process that reads from multiple sources without knowing in advance which source will provide some input first
- Three solutions:
  - alternate non-blocking reads on input sources (wasteful of CPU)
  - fork a process for each input source, and each child can block on one specific input source (can be hard to coordinate/synchronize)
  - use the **select()** system call …



## select(2)

- Usage:

```
#include <sys/time.h>
#include <sys/types.h>
int select( int nfds,
            fd_set *readfds,
            fd_set *writefds,
            fd_set *exceptfds,
            struct timeval *timeout );
```

- where the three fd_set variables are file descriptor *masks*
- fd_set is defined in <sys/select.h>, which in included by <sys/types.h>

## select(2) cont.

- The first argument (**nfds**) represents the number of bits in the masks that will be processed. Typically, this is 1 + the value of the highest fd
- The three **fd_set** arguments are bit masks … their manipulation is discussed on the next slide
- The last argument specifies the amount of time the select call should wait before completing its action and returning:
  - if **NULL**, select will wait (*block*) indefinitely until one of the file descriptors is ready for i/o
  - if **tv_sec** and **tv_usec** are zero, select will return immediately
  - if timeval members are non-zero, the system will wait the specified time *or* until a file descriptor is ready for i/o
- **select()** returns the number of file descriptors ready for i/o



## "FD_" macros

- Useful macros defined in <sys/select.h> to manage the masks:

```
void FD_ZERO( fd_set &fdset );
void FD_SET( int fd, fd_set &fdset );
void FD_CLR( int fd, fd_set &fdset );
int  FD_ISSET( int fd, fd_set &fdset );
```

- Note that each macro is passed the *address* of the file descriptor mask



## select(2) example

```
#include <sys/types.h>
fdset rmask;
int fd;        /* a socket or file descriptor */

FD_ZERO( &rmask );
FD_SET( fd, &rmask ); FD_SET( fileno(stdin), &rmask );

while(1) {
   select( fd+1, &rmask, NULL, NULL, NULL );
   if( FD_ISSET( fileno(stdin), &rmask)
      /* read from stdin */
   if( FD_ISSET( fd ), &rmask)
      /* read from descriptor fd */
   FD_SET( fd, &rmask); FD_SET( fileno(stdin), &rmask
   );
}
```



# Section #13 A

Miscellaneous

(can be skipped if short on time)

## Creation & Removal

- Directory creation and removal:
  int status = mkdir(char *path, int mode);
  int status = rmdir(char *path);
- File creation:
  #include <sys/file.h>
  int fd = open(char *path, int flags, int mode);
  mode:
  O_RDONLY 000
  O_WRONLY 001
  O_RDWR 002
  O_NDELAY 004 /* non-blocking*/
- Device creation
  int status = mknod(char *path, int mode, int dev);
- File removal(except for directories):
  int status = unlink(char *path);



## Process Priorities

- The system gives CPU scheduling priority to processes that have not used CPU time recently. Well, sort of.
- Process scheduling is a complex dance to try to second-guess the best allocation of CPU time to jobs to provide good interactive response and good throughput.
- It is possible to determine the current priority (an integer in the range -n to +n), or alter this priority by:

```
#define PRIO_PROCESS    0
#define PRIO_PGRP       1
#define PRIO_USER       2

int prio = getpriority(int which, int who);

int status = setpriority(int which, int who, int prio);
```



## Resource Utilization

- The resources used by a process are returned by:
  #include <sys/resource.h>
  int status = getrusage(int who, struct ruasge *rusage);
- The **who** parameter specifies whose resource usage is to be returned: those of the current process, or those of all terminated children of the current process.
- Resource usage information is returned concerning:
  - user time
  - system time
  - max core resident set
  - data mem size
  - page reclaims
  - page faults
  - swaps
  - block inputs
  - signals received
  - ...



## Resource Limits

- Resource usage may be controlled by:

```
#include <sys/resource.h>
    struct rlimit {
        int rlim_cur;
        int rlim_max;
    }
int status = getrlimit(int resource, struct rlimit *r);
int status = setrlimit(int resource, struct rlimit *r);
```

- Only the superuser can raise rlim_max.
- Other processes may alter rlim_cur within the range from 0 to rlim_max or (irreversible) lower rlim_max.
- The various resources whose limits may be controlled in this manner are:

| | |
|---|---|
| - milliseconds of CPU time | - maximum stack segment size |
| - maximum file size | - maximum core file size |
| - maximum data segment size | - maximum resident set size |

# System Support

- The UNIX file system name space may be extended by:
  int status = mount(char *blkdev, char *dir, int ronly);
- A device may be made available for swappng or paging by:
  int status = swapon(char *blkdev, int size);
- A file system not currently being used can be unmounted by:
  int status = unmount(char *dir);
- All system cache buffers may be scheduled to be cleaned by:
  sync();
- The system may be rebooted by:
  reboot(int how);

- The system optionally keeps an accounting record in a file for each process that exists on the system. The accounting can be enabled to a file by:
  int status = acct(char *path);



# Descriptors

- Descriptors are used to access resources such as files, devices, and communication links.

- A process access its descriptors indirectly through its own descriptor reference table, whose size is given by:
  int nds = getdtablesize();

  The entries in this tables are referred to by integers in the range 0 .. nds-1.



# Managing Descriptors

- A duplicate of a descriptor reference may be made by:
  int new = dup(int old);
  The new descriptor reference is indistinguishable from the old one.

- A copy of a descriptor reference may be made in a specific slot by:
  int status = dup2(int old, int new);
  This causes the system to deallocate the descriptor reference count occupying slot new, if any, replacing it with a reference to the same descriptor as old.

- A descriptor reference deallocation may also be performed by:
  int status = close(int old);



# Reading File Attributes

- Detailed information about the attributes of a file may be obtained wit the call:
  #include <sys/stat.h>
  int status = stat(char *path, struct stat *stb);
  int status = fstat(int fd, struct stat *stb);
- The stat structure includes:
  - file type
  - protection
  - ownership
  - access times
  - size
  - hard link count
- If the file is a symbolic link, the status of the link itself may be found by:
  int status = lstat(char *path, struct stat *stb);

# Modifying File Attributes

- Newly created files are assigned the user ID of the process that created it, and the group ID of the directory in which it was created.
- Ownership can be changed by:
```
int status = chown(char *path, int owner, int group);
int status = fchown(int fd, int owner, int group);
```
- The protection attributes associated with a file may be changed by:
```
int status = chmod(char *path, int mode);
int status = fchmod(int fd, int mode);
```
- The access and modify times on a file may be changed by:
```
int status = utime(char *path, struct timeval *tvp[2]);
```



# Links & Renaming

- Links allow multiple names for a file to exist. They exist independently of the file linked to.
- Two types of links exist:
- Hard Links
  - A reference counting mechanism that allows files to have multiple names within the same file system.
  - A hard link insures the target file will always be accessible even after its original directory entry is removed.
```
int status = link(char *path1, char *path2);
```
- Symbolic Links
  - Cause string substitution during the path name interpretation process.
  - A symbolic link does not insure that the target file will be accessible. In fact, a symbolic link to a non-existent file can be created.
```
int status = symlink(char *path1, char *path2);
int len = readlink(char *path, char *buf, int size);
```
- Atomic renaming of file system resident objects is done by:
```
int status = rename(char *old, char *new);
```



# Extension & Truncation

- Files are created with zero length and may be extended by writing to them.
- While a file is open the system maintains a pointer into the file indicating the current location in the file associated with the descriptor. This pointer may be moved by:
```
#include <sys/file.h>
int oldoffset = lseek(int fd, int offset, int whence);
#define SEEK_SET    0
#define SEEK_CUR    1
#define SEEK_END    2
```
- Files may have “holes” in them: void areas where data has never been written. Holes are treated as zero-valued bytes.
- Files may be truncated by:
```
int status = truncate(char *path, int newlen);
int status = ftruncate(int fd, int newlen);
```



# Checking Accessibility

- A process running may interrogate the accessibility of a file to the real user. This may be of particular interest to processes with different real and effective user ids.
```
#include <sys/file.h>
int accessible = access(char *path, int how);
#define F_OK    0
#define X_OK    1
#define W_OK    2
#define R_OK    3
```
- The presence or absence of advisory locks does not affect the result of **access**.

## Locking

- The file system provides basic facilities that allow cooperating processes to synchronize their access to shared files.
- The system does not force processes to obey the locks; they are of an advisory nature only.
- Locking is performed after on **open** call by:

```
#include <sys/file.h>
int status = flock(int fd, int how);
#define LOCK_SH     1 /* shared lock */
#define LOCK_EX     2 /* exclusive lock */
#define LOCK_NB     3 /* non-blocking */
#define LOCK_UN     4 /* unlock */
```

- If an object is currently locked by another process when an **flock** call is made, the called will be blocked until the current lock owner releases the lock.



# Section #14

## Signals



## Signals

- The system defines a set of signals that may be delivered to a process.
- A process may do one of three things with a signal:
  - Handle
    - The process specifies a handler function that is to be called on receipt of the signal. When the function returns, control is returned to the point in the program at which the signal occurred.
  - Block
    - Set mask to prevent delivery of signal until unmasked.
  - Ignore
    - If the signal occurs, no action is taken.
  - Default
    - If the signal occurs, the UNIX default action (which varies from signal to signal) is taken. This may be one of:
      - Do nothing.
      - Process termination (with or without core dump)
      - Process suspension.



## Signal Types

- The various types of signals are (/usr/include/signal.h):

| | |
|---|---|
| SIGFPE | Floating point exception |
| SIGILL | Illegal instruction |
| SIGSEGV | Attempting access to addresses outside the currently assigned areas of memory. |
| SIGBUS | Accesses that violate memory protection constraints. |
| SIGIOT | I/O trap |
| SIGEMT | Emulation trap |
| SIGTRAP | Single-step trap |
| SIGINT | Interrupt from keyboard (^C) |
| SIGQUIT | Same as SIGINT but with a core dump (^\) |
| SIGHUP | “Hang up” - for graceful process terminations. |
| SIGTERM | Terminate by user or program request. |
| SIGKILL | Same as SIGQUIT but cannot be caught, blocked, or ignored. |
| SIGUSR1,SIGUSR2 | User defined signals. |
| | |
| SIGALRM | Alarm – timeout of a timer (used by alarm(2)) |
| SIGVTALM | ??? |
| SIGPROF | Expiration of interval timers. |
| SIGIO | If requested, occurs when I/O possible to a descriptor. |
| SIGURG | Urgent condition. |
| SIGSTOP | Causes suspension. Cannot be caught. |
| SIGTSTP | Suspend by user request. |
| SIGTTIN | Suspend because input attempted from terminal. |
| SIGTTOU | Suspend because output attempted to terminal. |
| SIGCHILD | Child process’ status has changed. |
| SIGXCPU | Occurs when a process near its CPU time limit. |
| SIGXFSZ | Occurs when limit on file size creation has been reached. |

## Handling Signals

- A process changes the way a signal is delivered with:

```
#include <signal.h>
struct sigvec {
   int (*sv_handler)(int signo, long code, struct sigcontext *scp);
   int sv_mask;
   int sv_flags;
};
int status = sigvec(int signo, struct sigvec *sv, struct sigvec *csv);
```

- Possible values for *sv_handler* are a function, SIG_IGN, or SIG_DEF.
- *sv_mask* specifies which additional signals are to be masked on receipt of this one (implicitly includes *signo*).
- *Sv_flags* indicate whether system calls should be restarted if the signal handler returns, and whether the signal handler should operate on the normal stack or an alternate stack.



## Signal Delivery

- When a signal condition arises for a process, the signal is added to a set of signals pending for the process.
- If the signal is not currently *blocked* by the process then it will be delivered.
- Signal delivery involves:
  1) Adding the signal to be delivered and those signals specified in the *sv_mask* to a set of those masked (ie., blocked) for the process.
  2) Saving the current process' context
  3) Placing the process in the context of the signal handling routine.
- The context of the signal handler is so arranged that if the function returns normally the original signal mask will be restored and the process will resume execution in the original context.



## Signal example

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/signal.h>
int i=0;
void quit(int sigNum) {
   fprintf(stderr, "\nInterrupt (signal=%d,i=%d)\n",
   sigNum, i);
   exit(123);
}
void main(void) {
   signal(SIGINT, quit);
   signal(SIGTERM, quit);
   signal(SIGQUIT, quit);
   while(1)
       if ( i++ % 5000000 == 0) putc('.', stdout);
}
```



## Blocking Signals

- Blocked signals are added to the mask.
- If masked signals occur then delivery is delayed until the signals are unblocked or unmasked.
- To add a set of signals to the mask:
  – long oldmask = sigblock(long mask);
- To set the mask:
  – long oldmask = sigsetmask(long mask);
- To mask a set of signals, wait for an unmasked signal, and then restore the original mask:
  – int signo = sigpause(long mask);

## Sending Signals

- Signals may be sent either from the keyboard via the terminal driver or from another process:
  - int status = kill(int pid, int signo);
  - int status = killpgrp(int pgrp, int signo);
- Unless the process is superuser, it must have the same effective user id as the process receiving the signal.
- Signals are also sent implicitly from a terminal device to the process group associated with the terminal when certain input characters are typed.



## Signal Stacks

- For applications that change stacks periodically, signal delivery can be arranged to occur on a stack that is independent of the one in use at the time of signal delivery.

```
struct sigstack {
    void *ss_sp;
    int ss_onstack;
};
int status = sigstack(struct sigstack *ss, *oss);
```



## Interval Time

- The system provides each process with three interval times:
  - REAL - Real time intervals. SIGALRM is delivered when this timer expires.
  - VIRTUAL - Virtual time runs only when the process is executing user code. SIGVTALRM is delivered when this timer expires.
  - PROF - Profiled time runs when the process is executing user code or system code on behalf of that process. SIGPROF is delivered when this timer expires.
- A timer is set or read by:

```
struct itimerval {
        struct timeval it_interval;
        struct it_value; /* current value */
};
int status = getitimer(int which, struct itimerval *value);
int status = setitimer(int which, struct itimerval *v, struct itimerval *ov);
```



## Execution Profiling

- Execution *profiling* means gathering statistics on how long a process executes particular pieces of code.
- Profiling is turned on by:
  - int status = profil(void *buf, int bufsize, int offset, int scale);
- This begins sampling of the program counter, with statistics maintained in the user provided buffer.

# Section #15

## Concurrency II

Process Synchronization



## Concurrency Topics

- Asynchronous Activities
- Concurrency vs Parallelism
- Uniprocessor vs Multiprocessor
- Threads
- Thread-safe code and Reentrancy
- Race conditions
- Deadlock
- Dining Philosophers
- Starvation
- Critical Sections
- Concurrency Control (Execution Context, Scheduling, Synchronization)
- Synchronization Primitives (Mutexes, Semaphores, Monitors, and Message Passing)
- Programming Approaches (Pipes, Work Crew, Client/Server)



## Circular Buffers

- A *circular buffer* is a method of implementing a first-in-first-out (FIFO) queue.
- Items are inserted into the queue at position **in**, and fetched from position **out**.
- The buffer "wraps around" at the endpoints, so the position after N-1 is position 0.
- These are also referred to as bounded buffers because no more than N items can be held at one time.

```
char buffer[N];
int in=0, out=0, used=0;
void Insert(char c) {
    if (used == N)
        ERROR("buffer overflow!");
    buffer[in] = c;
    in = (in + 1) % N;
    ++used;
}
```

```
char  Fetch(void) {
    if (used == 0)
        ERROR("buffer underflow!");
    char nextc = buffer[out];
    out = (out + 1) % N;
    --used;
    return nextc;
}
```



## The Producer-Consumer Problem

- Consider what happens if two processes have concurrent read-write access to the buffer.
- The Producer process inserts things into the buffer.
- The Consumer process removes things from the buffer.
- Unless we're very lucky, there will be problems with the following.

```
/* Producer Process */

char val;
while(1) {
    val = produce_item();
    Insert(val);
}
```

```
/* Consumer Process */

while(1) {
    next_val = Fetch();
    consume_item(next_val);
}
```

# Critical Sections Again

Recall...

- A *critical section* is an area of code or data that depends on there being only one process inside at any one time for correct operation to take place. (e.g. a linked-list data structure or a circular buffer)
- Code that modifies a shared variable usually has the following form:

```
ENTRY SECTION
  Critical Section
EXIT SECTION
  Remainder Section
```

- Entry Section - The code that requests permission to modify the shared variable.
- Critical Section - The code that modifies the shared variable.
- Exit Section - The code that releases access.
- Remainder Section - The remaining code.



# Atomic Operations

- An *Atomic Operation* is an operation that, once started, completes in a logically indivisible manner. Most solutions of the critical-section problem rely on some form of atomic operation.
- On a machine with a single CPU, individual machine instructions are often atomic but necessarily so.
- Note that:

```
value = 5;
```

is a C *statement* and probably translates into several machine instructions.



# Two-Process Mutual Exclusion

(Wrong Algorithm #1)

- Assume there are two processes, 0 and 1.
- We will have a variable called turn which is -1 if it's nobody's turn, otherwise it's 0 or 1.
- When a process wants to enter its critical section, it checks to see if turn is -1, then sets turn to itself.
- Both processes execute the same code below except the have different values of id.

```
shared int turn = -1;
/* Process 0 */
while(1) {
    while(turn != -1) /* busy wait */;
    turn = 0;
    /* critical section */
    turn = -1;
    /* remainder section */
}
```

```
/* Process 1 */
while(1) {
    while(turn != -1) /* busy wait */;
    turn = 1;
    /* critical section */
    turn = -1;
    /* remainder section */
}
```



# Two-Process Mutual Exclusion

(Wrong Algorithm #2)

- Idea: Don't be greedy and take control. Be courteous by waiting for it to be given to you.

```
local const int id; /* initialized to 0 or 1, depening on which process */
shared int turn = 0; /* initialize to one of them */
```

```
/* Process 0 */
while(1) {
    while(turn != id);

    /* critical section */
    turn = 1-id;
    /* remainder section */
}
```

```
/* Process 1 */
while(1) {
    while(turn != id);

    /* critical section */
    turn = 1-id;
    /* remainder section */
}
```

## Two-Process Mutual Exclusion

(Wrong Algorithm #3)

- Idea: Check to see if the other process wants to enter its critical section. If not, then it's OK to enter.
- When you want to enter, turn on a flag.

shared int want[2] = { 0, 0 };
local const int id = /* initialized to 0 or 1 for process id*/

```
/* Process 0 */
while(1) {
    want[id] = 1;
    while(want[1-id]);
    /* critical section */
    want[id] = 0;
    /* remainder section */
}
```

```
/* Process 1 */
while(1) {
    want[id] = 1;
    while(want[1-id]);
    /* critical section */
    want[id] = 0;
    /* remainder section */
}
```



## Two-Process Mutual Exclusion

- Dekker first solved the problem in the early 1960's but his solution allowed starvation to occur in the presence of contention.
- Peterson came up with a solution in 1981 that was simpler and didn't suffer from starvation problems.
- Remember we are only assuming memory interlock for these algorithms.

- The idea combines the notions from the last two incorrect algorithms.
- When you want to enter your critical section, turn on your flag.
- Then offer turn to the other process. If it wants it, it gets it; otherwise you can take it.



## Peterson's Algorithm

shared int want[2] = { 0, 0 };
shared int turn =0;
local const int id = /* initialized to 0 or 1 for process number */

```
/* Process 0   (id == 0) */
while(1) {
    want[id] = 1;
    turn = 1 - id;
    while(want[1-id] && turn == 1-id);
    /* critical section */
    want[id] = 0;
    /* remainder section */
}
```

```
/* Process 1  (id == 1)*/
while(1) {
    want[id] = 1;
    turn = 1 - id;
    while(want[1-id] && turn == 1-id);
    /* critical section */
    want[id] = 0;
    /* remainder section */
}
```



## The Test-and-Set Instruction

- Things are much easier when the hardware provides a mechanism to implement mutual exclusion without the need for Peterson's algorithm.
- Test-and-Set is one such machine instruction that is available on some processors. It defined as an *atomic operation* that implements the following logical function:

```
int TestAndSet(int *p) {
     int value = *p;
    *p = 1;
    return value;
}
```

- In assembly language, entering a critical section might look like:

```
loop:   tset busy
        branch-if-zero critical section
        jmp loop
```

## Mutexes

- We have seen how two processes can ensure mutual exclusion.
- Regardless of the implementation, it is often sufficient to assume the existence of a high level locking facility with a simple call interface.
  - void MutexBegin(void); // block, or return FALSE if you're not allowed to enter your critical section
  - void MutexEnd(void);
- The above functions would be suitable for a single global lock.
- It is often better to organize things into localized locks.



## Process Synchronization

- Locking critical sections using mutexes works well for short operations. However it doesn't work well for unbounded waiting.
- Recall the Producer/Consumer problem. If the consumer finds an empty buffer, it must wait until the producer can add to the buffer. The consumer doesn't know how long it has to wait. With only MutexBegin/MutexEnd, it would have to spin in a busy loop to keep checking for more work.
- *Condition Variable*s are used to sleep for some event or condition and wake-up when that condition is fulfilled.



## Semaphores

- A semaphore provides two operations:
  - Wait (down, P, lock)
  - Signal (up, V, unlock)
- Dijkstra proposed the semaphore concept in 1965.
- P and V are from the Dutch words *passeren* (to pass) and *vrygeven* (to release).
- A semaphore, s, is a non-negative integer that is atomically updated using the P and V primitives. Note the fact that it is an integer with the special update properties.
- An analogy to marbles in a bowl. s is the number of marbles, P(s) tries to take a marble (it may have to wait), and V(s) puts one marble back (it might wake up another process doing a P(s)).



## Implementing Semaphores

```
void Signal(int *s) // up, unlock
{
    MutexBegin();
    *s = *s + 1;
    MutexEnd();
}
```

```
void Wait(int *s) // down, lock
{
    int blocked = true;
    do
    {
        MutexBegin();
        if (*s > 0)
        {
            *s = *s - 1;
            blocked = false;
        }
        MutexEnd();
    } while(blocked);
}
```

Exercise:

MutexBegin() and MutexEnd() can be implemented using semaphores just as semaphores can be implemented using mutexes.

Try to do it.

## Other Primitives

- We have seen Mutexes and Semaphores.
- Other terms you will hear are Monitors and Message Passing.
- Message Passing works by having each thread/process send messages back and forth. Receiving a message is usually a blocking operation.
- Monitors are a higher level abstraction than message passing and semaphores. They associate a set of methods to the resource or data that requires access control.



## Programming Approaches

- Pipes
  - We've seen this in the shell. It is essentially a chain of producer/consumer pairs.
- Work Crew
  - A group of worker processes grab work from a pool of jobs.
- Client/Server
  - A server process serves the requests of the client processes. (Remember the X Window System?)



# Section #16

## UNIX Memory Management



## Memory Management

- The operating system must manage the memory resources of the system. It should try to do so efficiently.
- With virtual memory systems, it is up to the operating system to manage the allocation of information(code & data) between main memory (core memory, RAM, physical memory) and secondary storage (usually disks or servers on the network).
- The memory management subsystem in the kernel works with the *Memory Management Unit* (MMU) hardware.

# Virtual Memory

- Each application is given the illusion that it has a large main memory at its disposal.
- Each process has a *process address space* which maps to the physical address space of the computer.

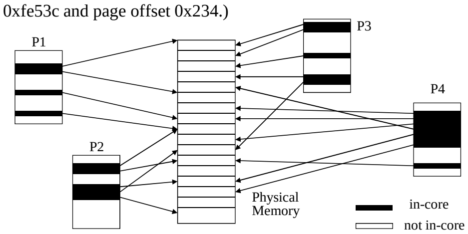
Memory management and virtual memory advantages:

- The ability to run programs larger than physical memory
- Run partially loaded programs, thus reducing program startup time.
- Allow more than one program to reside in memory at one time.
- Allow sharing. For example, two processes running the same program should be able to share a single copy of the code in memory.
- Access control. One process shouldn't be able to trample over another process' memory.

CS146 229

# Demand Paging

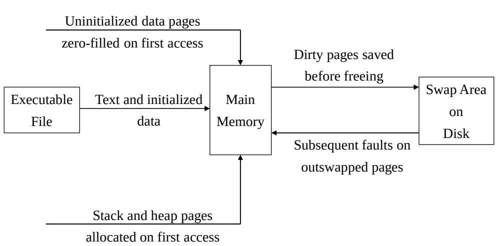
- *Demand paging* systems divide the physical and process address spaces into fixed-size pages (eg 4k or 8k).
- Each page is brought into or out of main memory as needed.
- Note that the page size is a power 2. Therefore, for any address, you can determine the page number and page offset with simple bit operations (shift or mask). (eg With 4k pages, 0xfe53c234 is page 0xfe53c and page offset 0x234.)




CS146 230

# Swapping Pages

- Swapping used to refer to swapping whole processes between disk and memory. With demand paging, we only send individuals pages of memory to the swap space (on disk).
- Dirty pages are memory pages modified so that they are irreplaceable now. Code pages are never dirty because they are read-only.




CS146 231

# Copy-On-Write

- Copy-on-Write (COW) is a technique to save work on a fork.
- Fork() is VERY often followed immediately by an exec() call.
- Therefore, it would wasteful to make a full duplicate of the process in memory when it forks.
- The idea is to share all data pages until data is changed by either the parent or child (before a page is touched, the parent and child can share the page because it is identical for both of them).
- When the page is written-to, the kernel intercepts the operation and makes a copy of the page. Now parent and child have their own copies.

- Why don't code pages undergo Copy-on-Write?

CS146 232

# mmap()

- **mmap()** maps a file (usually a disk file or /dev/zero) into a buffer in memory, so that when bytes are fetched from the buffer the corresponding bytes of the file are read
- Multiple processes can map the same file simultaneously.
- Usage:

```
caddr_t mmap( caddr_t addr, size_t len, int
       prot, int flag, int filedes, off_t off
  );
```

  - `addr` and `off` should be set to zero,
  - `len` is the number of bytes to allocate
  - `prot` is the file protection, typically (`PROT_READ|PROT_WRITE`)
  - `flag` should be set to `MAP_SHARED` to emulate shared memory
  - `filedes` is a file descriptor that should be opened previously



# mmap() example

```
char *ShareMalloc( int size )
{
   int  fd;
   char *returnPtr;
   if( (fd = open( "/tmp/mmap", O_CREAT | O_RDWR, 0666 )) < 0
   )
      Abort( "Failure on open" );
   if( lseek( fd, size-1, SEEK_SET ) == -1 )
      Abort( "Failure on lseek" );
   if( write( fd, "", 1 ) != 1 )
      Abort( "Failure on write" );
   if( (returnPtr = (char *) mmap(0, size,
   PROT_READ|PROT_WRITE,
         MAP_SHARED, fd, 0 )) == (caddr_t) -1 )
      Abort( "Failure on mmap" );
   return( returnPtr );
}
```



# Section #17

## Source Code Revision Control

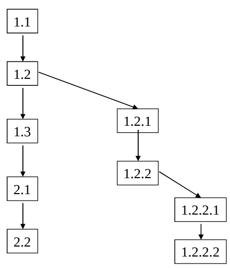


# Source Code Management

- Projects can involve many files that evolve over a long period of time.
- It is often necessary to keep track of the versions of the files and control changes from several people making updates.
- The SCCS (Source Code Control System) and RCS (Revision Control System) systems are collections of utilities to manage file revisions. They control who is allowed to make changes to files, store all changes, and any comments that accompanied the changes. All versions are accessible. The system maintains a copy of the file with differences between subsequent versions.
- Source code management can be just as useful for documentation, reports, assignments, html files, and system configuration files.
- We will discuss the RCS system in this section.

## Versions of a File

- An RCS file can have four components to its version number. They are the release, level, branch, and sequence numbers.






## RCS Commands

- There are several commands in the RCS system.

  rcs - The main utility program.

  ci - check-in

  co - check-out

  rlog - rcs file information and revision history

  rcsdiff

  - Show the differences between a revision and working file or between two revisions.



## Creating an RCS File

- To create an RCS file, use the check-in command (ci).

```
$ ci mysource.c
mysource.c,v  <--  mysource.c
enter description, terminated with a single '.' or the end of file:
NOTE: This is NOT the log message!
>> C source for my network game.
>> .
initial revision: 1.1
done
$
```

- It is common to create a directory called RCS so that the mysource.c,v file goes into the RCS subdirectory instead of cluttering the current directory.
- The -u option is commonly used to keep a copy of the file available.



## Updating an RCS File

- To update an RCS file you lock the file, make your changes and then check-in the file.
- To lock use check-out with the lock flag(-l):

```
$ co -l mysource.c
RCS/mysource.c,v  -->  mysource.c
revision 1.3 (locked)
done
$
```

- To check in the changes, just use the ci command:

```
$ ci -u mysource.c
RCS/mysource.c,v  <--  mysource.c
new revision 1.4; previous revision: 1.3
enter log message, terminated with single '.' or end of file:
>> fixed socket connection subroutine
>> .
done
$
```

# Retrieving Versions

- You can check out any version of the RCS file.

```
$ co -r1.3 mysource.c
RCS/mysource.c,v  -->  mysource.c
revision 1.3
done
$
```



# RCS Historical Information

- The rlog command lets you examine information about the RCS file.

```
$ rlog mysource.c
RCS file: RCS/mysource.c
Working file: mysource.c
head: 1.4
branch:
access list:
total revisions: 2;   selected revisions: 2
description:
C source for my network game
-------------------------------
revision 1.4
date: 1998/07/21 17:26:45; author: wayne; state: Exp; lines +2 -0
fixed socket connection subroutine.
-------------------------------
revision 1.3
```



# RCS Access Controls

- RCS lets you restrict access to a set of people for updating the RCS file. Access control to look at the file is left to the standard UNIX permissions.

- RCS provides features for naming revisions or branching revisions. See “man rcs” and “man rcsintro” for more details.



# Advantages of Revision Control

- A good revision control system manages your changes for you.
- Many people make backup copies of their files or use filename conventions to handle versioning. These methods are prone to error. Note that a revision control system is NOT a replacement for a backup system!
- A revision control system keeps your changes, your comments about those changes, and the full history of your file in one place in an easily retrievable form, and does it efficiently because it can store just the differences instead of full copies of each version.

# Section #18

# Security



## Security Topics

- Computer security should be a concern of everyone. Systems programmers need to be aware of it even more than most because they are more likely to be working on servers in a network environment, etc.
- Topics to discuss:
  - Passwords
  - Root v.s. user
  - SUID
  - Detecting security breaches. Cleaning up.
  - Buffer overflows
  - Security through obscurity
  - Denial of service attacks
  - Network firewalls



## Passwords

- Passwords are stored on the system as encrypted strings.
- When you type your password, the login process encrypts your password and compares the two encrypted strings.

- Encrypted passwords can be cracked. Therefore, it is beneficial to keep the encrypted passwords in a more secure place than /etc/passwd.
- Shadow passwords are passwords kept in /etc/shadow/ instead of /etc/passwd. A shadow-aware version of login looks in /etc/shadow/passwd for passwords in addition to the usually information kept in /etc/passwd. /etc/shadow has permissions for only root. Therefore, casual users cannot look at the encrypted passwords.

- Passwords for ftp, telnet, rcp, etc, are sent over the network as plain text => use ssh instead.
- If you EVER type your password in the clear over a network, it should be changed immediately. Some systems support expiry dates on passwords.



## Root v.s. User

- If you don’t need to run a program as the superuser (root), then don’t. (same goes for Windows: don’t run as Administrator unless necessary)

- That also applies to system daemons, etc. If you install a software package that needs to run a server process, see if you can create a new user to run it.

- Novice system administrators often make the mistake of logging in as root and doing everything as root. Think what happens if you type “rm -rf *” in the wrong directory.

## Set User-ID Bit

- You can use the SUID permission on an executable to allow a program to run with the owner's access instead of user that ran the program.
- Very simply. SUID shell scripts are prone to security holes. In more ways than you can imagine.
- Binary executables can have many security problems if they are SUID root. See Buffer Overflows later.
- Programs that are designed to be SUID root should be made to minimize the part of code that is root powerful and deals with external inputs.



## Detecting & Cleaning Security Breaks

- Detecting a break-in is not always easy to do. Sometimes the intruder can be exceptionally thorough by replacing commands such as cp, sum, or diff to detect a detection attempt and thwart it.
- Using checksum programs like sum(1) are unreliable because an intruder could have carefully crafted changes to the file so that the checksum matches. Byte-by-byte comparisons are the only real test.
- You need to ensure that everything you use comes from a trusted copy (CDROMs are good for this) and you need to be aware that other hosts on the network are not trusted hosts until they have been checked and cleaned.
- Assuming you detect a break-in, how do you purge the system of back doors and viruses?



## Buffer Overflows

- The most famous buffer overflow example is the Internet Worm. The finger server, fingerd, used gets() for it's input reading. gets() does not check the length of the line read.

```
char line[512];
gets(line);
```

- If the intruder supplies a line of data longer than 512 bytes, that data will overwrite the stack frame and can cause fingerd to start running the intruder's code. You should always use fgets() instead.
- Robert T. Morris inadvertently unleashed the Internet Worm in 1988 and effectively shut down the entire Internet. The Worm didn't control its propagation well enough and it choked the networks.
- Other potential buffer overrun calls: strcpy() and sprintf().
- Fingerd did not have to be running as root. This was simply foolish.



## Security through Obscurity

- Security by Obscurity is a technique used fairly regularly but generally ineffective. The idea is to limit information. For instance, hide an oddly named publicly writeable directory under a search-only directory(i.e. no read permission). Then tell only your friends the name of that directory.
- The problem with this approach is that no information is truly private and you have no explicit control or detection that something went wrong.
- For encryption algorithms, it can be quite serious. If someone said that they have a very secure encryption algorithm but the safety of the algorithm depends on it being kept secret, then it's not very secure. Information leaks can occur and analysis usually cannot be prevented.

## Denial of Service Attacks

- A *denial of service attack* is any situation where a malicious person can overload your network or operating system to prevent legitimate users from using the system.
- Denial of service attacks can take many forms and UNIX is generally very poor about handling such attacks.
- Examples:
  - eatmem - a program that allocates and dirties more and more data pages until no more processes can run
  - network attacks - send a large volume of network packets to saturate the network bandwidth thus preventing others from communicating

CS146 253

## Firewalls

- A *firewall* isolates two regions so a fire can't spread unchecked.
- A network firewall isolates an organization's network from external networks (e.g. the Internet).
- Firewalls can be used to limit access to or from the external network. This can allow very open and free access within the organization but prevent outsiders from having that same level of access.
- Firewalls simplify security protection since you only have to concern yourself with the firewall's filter instead of every machine on your network.

CS146 254

# Section #20

## Multi-platform Development

CS146 255

## Multi-Platform Development

- Configuring software for different operating systems and programming environments.
- Separating platform dependent from platform independent source code.
- Handling conditional compilation using #ifdef based on logical characteristics vs physical/platform characteristics.
- Using abstraction layers in your programs. E.g. a single API with multiple pluggable implementations to handle different databases (Oracle, Sybase, etc).
- Testing: Test suites are important to catch errors on different platforms because not all developers will use all platforms all of the time.
- Installation will probably be different on each platform.
- Porting to new platforms should get EASIER over time.

CS146 256

# Section #21

## The Plan 9 Operating System

http://plan9.bell-labs.com/plan9



## History

- Late 1980's
- Explore a new model of computing *system*.
  - Central administration
  - Cheap local graphical terminals
  - Large central shared resources (file and compute servers)
- Clean design (All resources are like files. No ioctl() style control.)

- The networking protocol (9P) is used for accessing all resources remotely.



## Name Spaces

- Plan 9 implements the concept of per-process name spaces.
- Each process can customize its view of the system.
- All resources are accessed via the name space (network, graphics, processes, files, serial ports, etc.)
- You can choose to mount or bind a file system in front or behind the current file system.
- Union directories allow file systems to overlap.
- For instance, the concept of the PATH environment variable is unnecessary. A PATH of /bin:/usr/bin:/local/bin:$HOME/bin would be aligned as five overlapping directories at the /bin location. This allows a very nice system for multiple platforms. The /platforms/mips/bin or /platforms/solaris/bin directory can be mounted into the /bin location as appropriate.
- The ordering of file systems in a union directory govern which file is chosen for reading or executing.



## Processes as Files

- Processes are accessible as files in Plan 9.
- The /proc file system is a kernel generated file system where each file is a gateway to the process' address space.
- /proc/3241 would be the directory for process number 3241.
- /proc/3241/status would be the status for the process.
- /proc/3241/mem is the virtual memory image.
- /proc/3241/text is a link to the executable file for the process.
- /proc/3241/ctl is used to control the process (e.g. stop or kill).

# 8½ - The Plan9 Window System

- The Plan 9 Window System has a novel design. It is a special form of file server. It opens the /dev/mouse, /dev/cons, and /dev/bitblt devices and provides sets of those same files as a file server would.
- This design allows one to run 8½ as a window inside another 8½!
- Each windowing application can treat its terminal devices as if it is the only user.