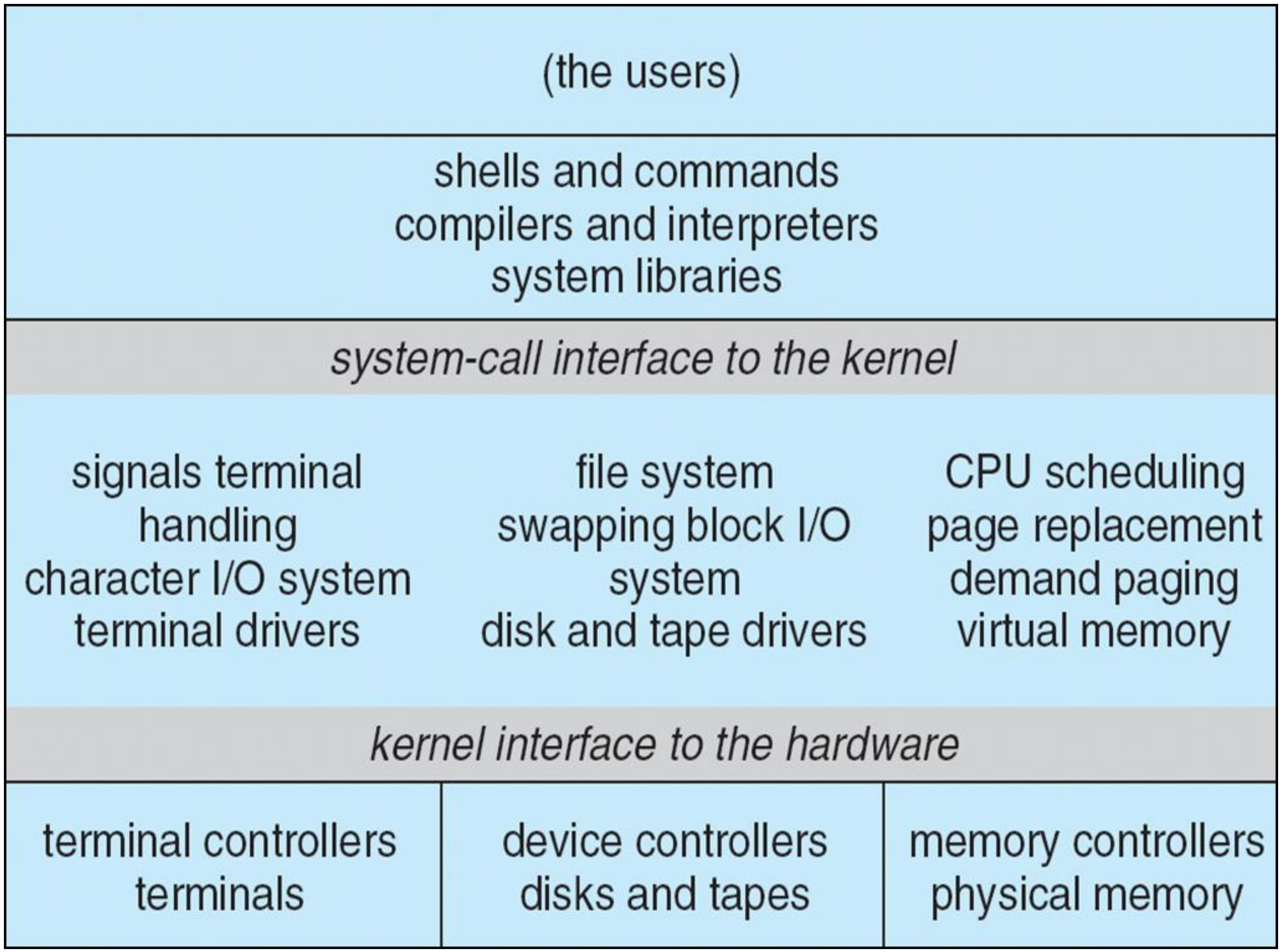
Process Management / Shell

Programmer Interface

- Kernel: Everything below the system-call interface and above the physical hardware
  - Provides file system, CPU scheduling, memory management, and other OS functions through system calls
- Systems programs: Use the kernel-supported system calls to provide useful functions, such as compilation and file manipulation

UNIX Layer Structure



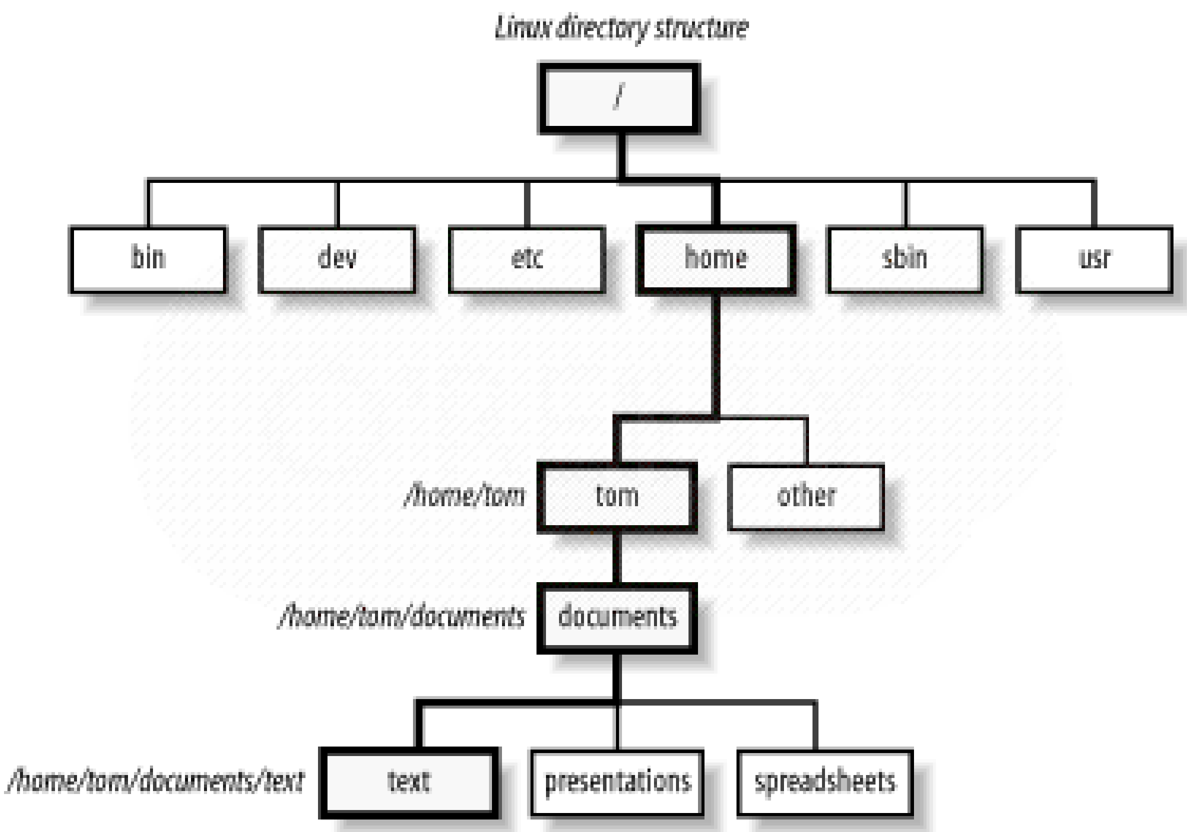
System Calls

- System calls define the programmer interface to UNIX
- The set of systems programs commonly available defines the user interface
- The programmer and user interface define the context that the kernel must support
- Roughly three categories of system calls in UNIX
  - File manipulation (same system calls also support device manipulation)
  - Process control
  - Information manipulation

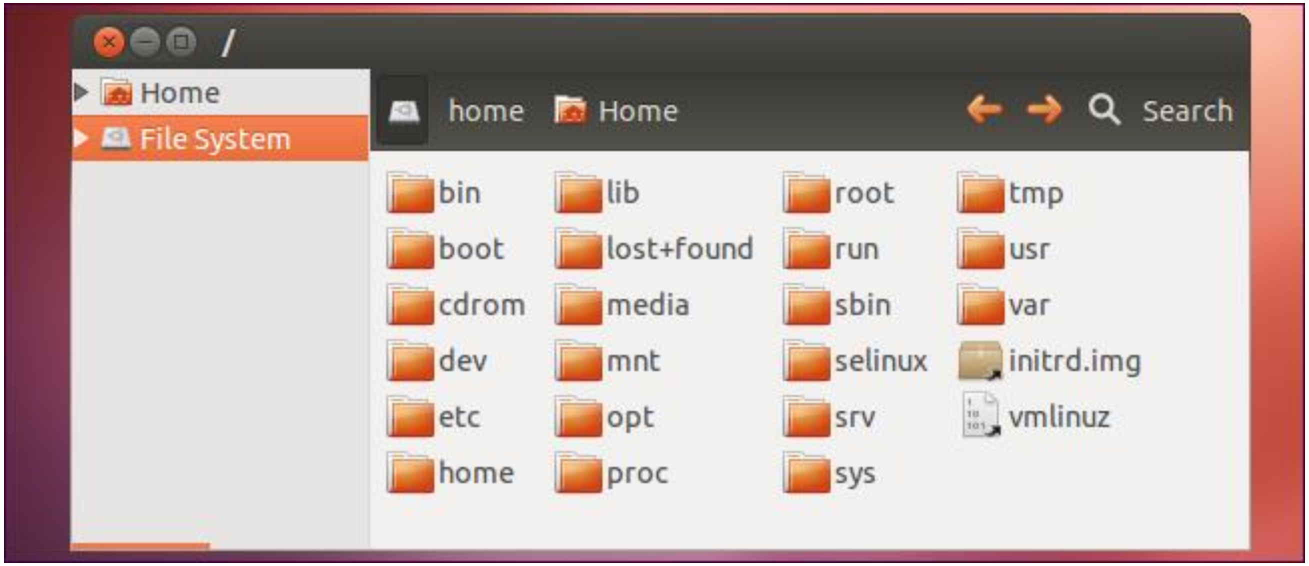
File Manipulation

- A **file** is a sequence of bytes; the kernel does not impose a structure on files
- Files are organized in tree-structured **directories**
- Directories are files that contain information on how to find other files
- **Path name**: identifies a file by specifying a path through the directory structure to the file
  - Absolute path names start at root of file system
  - Relative path names start at the current directory
- System calls for basic file manipulation: `creat`, `open`, `read`, `write`, `close`, `unlink`, `trunc`

Typical UNIX Directory Structure



Ubuntu Directory Structure



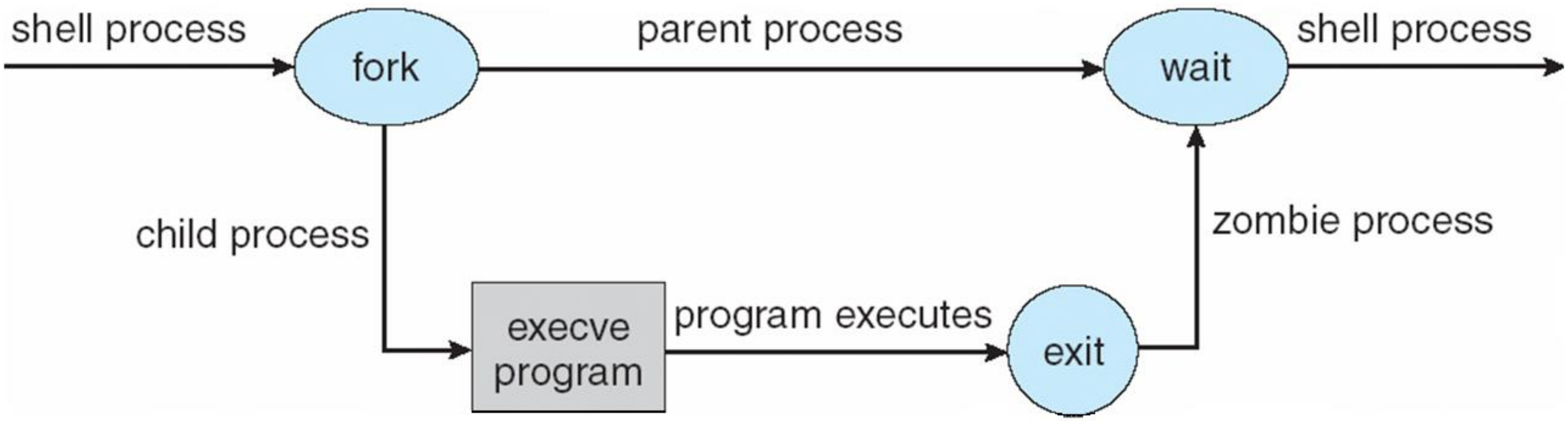
Process Control

- A process is a program in execution
- Processes are identified by their process identifier, an integer
- Process control system calls
  - `fork` creates a new process
  - `execve` is used after a fork to replace on of the two processes's virtual memory space with a new program
  - `exit` terminates a process
  - A parent may `wait` for a child process to terminate; `wait` provides the process id of a terminated child so that the parent can tell which child terminated

Process Control (cont.)

- A **zombie** process results when the parent of a **defunct** child process exits before the terminated child.
  - When a process dies it is not all removed from memory immediately.
  - The process status becomes EXIT_ZOMBIE and the process's parent is notified that its child process has died.
  - Parent process is supposed to execute the `wait()` system call to read the dead process's exit status.
  - Zombie children processes will stick around in memory until they are cleaned up.
  - The top command will display information about running, sleeping, stopped, and zombie processes in Linux.

Illustration of Process Control Calls



Process Control (Cont.)

- Processes communicate via pipes; queues of bytes between two processes that are accessed by a file descriptor.
- All user processes are descendants of one original process, `init`
- `init` forks a *getty* (get tty) process: initializes terminal line parameters and passes the user's *login name* to *login*.
  - *login* sets the numeric *user identifier* of the process to that of the user.
  - executes a **shell** which forks subprocesses for user commands.

Information Manipulation

- System calls to set and return an interval timer:
  `getitimer/setitimer`
- Calls to set and return the current time:
  `gettimeofday/settimeofday`
- Processes can ask for
  - their process identifier: `getpid`
  - their group identifier: `getgid`
  - the name of the machine on which they are executing:
    `gethostname`

Library Routines

- The system-call interface to UNIX is supported and augmented by a large collection of library routines.
- Header files provide the definition of complex data structures used in system calls.
- Additional library support is provided for mathematical functions, network access, data conversion, etc.

User Interface

- Programmers and users mainly deal with already existing systems programs: the needed system calls are embedded within the program and do not need to be obvious to the user.
- The most common systems programs are file or directory oriented.
  - Directory: `mkdir, rmdir, cd, pwd`
  - File: `ls, cp, mv, rm`
- Other programs relate to editors (e.g., `emacs, vi`) text formatters (e.g., `troff, TeX`), and other activities.

Standard I/O

- Most processes expect three file descriptors to be open when they start:
  - *standard input* – program can read what the user types.
  - *standard output* – program can send output to user's screen.
  - *standard error* – error output.
- Most programs can also accept a file (rather than a terminal) for standard input and standard output.
- The common shells have a simple syntax for changing what files are open for the standard I/O streams of a process — *I/O redirection.*

Standard I/O Redirection

Command	Meaning of command
% ls > filea	Direct output of ls to file filea.
% pr < filea > fileb	Input from filea and output to fileb.
% lpr < fileb	Input from fileb.
% ./fork1 > errs &	Save output in a file.

The Shell

- The command interpreter accepts commands from the user. It surrounds the kernel.
- The shell refers to the command interpreter being executed.
- Commands that we type in a Unix-based system such as “ls” or “ps” are not actually considered part of the operating system.
- The commands make use of system calls. They are executable binary object files.
- A system call allows a command to request a service from the operating system.
- A shell is a process.
  - A process e.g., a program being executed.

The Shell

- There are different shells that you can use in a Unix-based system including:
  - bourne shell
  - C shell
  - bash shell
  - tcsh shell
  - and many more
- A list of various shells may be found at:
  - https://en.wikipedia.org/wiki/Comparison_of_command_shells
  - https://en.wikipedia.org/wiki/Unix_shell

The Shell

- When a user logs in, a shell is started up.
- The shell has the terminal as standard input and display as standard output
- The shell starts out by typing the prompt, a character such as a dollar sign or percentage sign e.g.,

  jackson$

- User enters a command e.g.,

  jackson$ date

The Shell

- The user can specify that standard output be redirected to a file e.g.,

```
date >file
```

- Standard input can also be redirected e.g.,

```
sort <file1 >file2
```

- The output of one program can be used as the input to another program:

```
cat file1 file2 file3 | sort  >/dev/lp &
```

High-Level View of Shell Code

```
while (1)
{
  Get a line from the user.
  Execute command found in line.
}
```

Details Not Highlighted

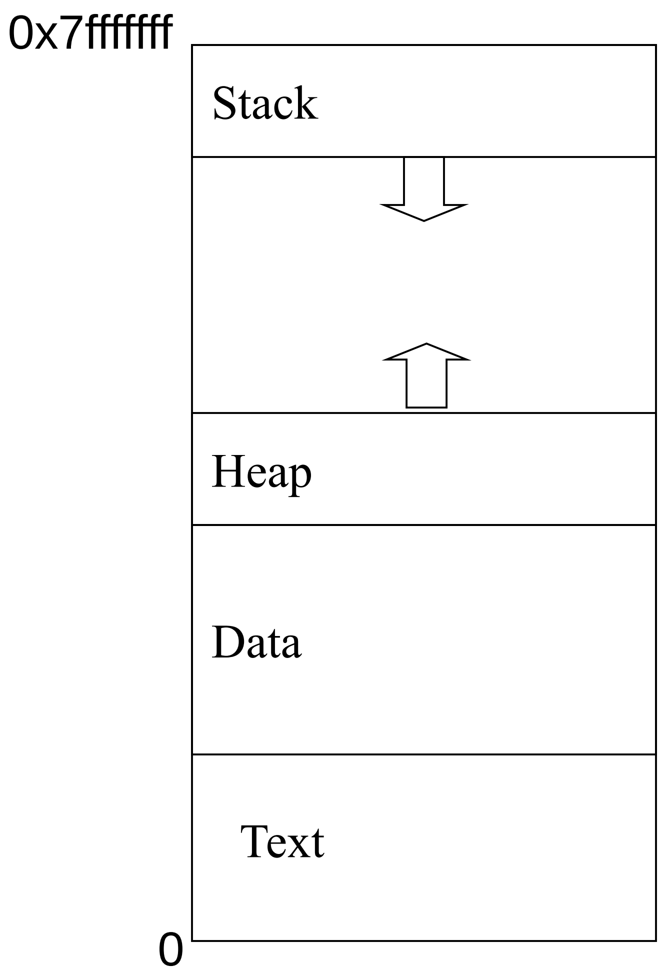
- Making sure that the line from the user is correct.
- How is shell termination handled?

- The execution of a command is done by a separate process (child process) from the shell process.
- For simple commands, the shell process waits for the child process to terminate so that it can print the prompt.
- If a child process is put in the background (using & the ampersand symbol) then the shell process can continue without waiting for the child process to terminate.

Unix Process Creation

- The Unix system call for process creation is called fork().
- The fork system call creates a child process that is a duplicate of the parent.
  - Child inherits state from parent process.
    - Same program instructions, variables have the same values, same position in the code.
  - Parent and child have separate copies of that state.
  - Child has the same open file descriptors from the parent.
    - Parent and child file descriptors point to a common entry in the system open file table.

Memory Image of a Unix Process



- Processes have three types of memory segments:
  - Text: program code.
  - Data:
    - Statically declared variables.
  - Heap
    - Areas allocated by malloc() or new (heap).
  - Stack
    - Automatic variables.
    - Function and system calls.
- Invoking the same program multiple times results in the creation of multiple distinct address spaces.

Unix Process Management

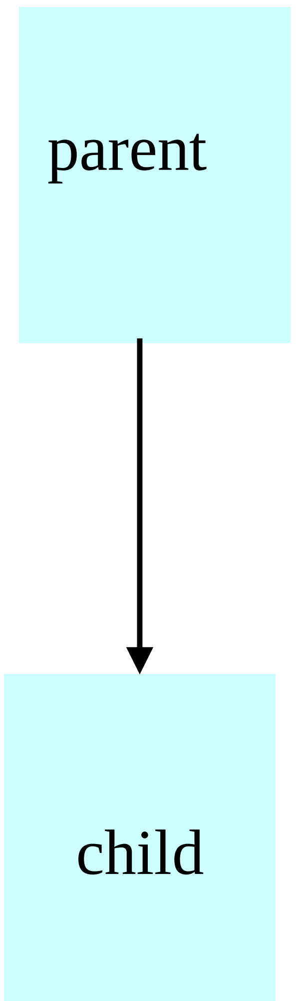
- Unix fork()
  - Creates a new address space.
  - Copies text, data, and stack into new address space.
  - Provides child with access to opened files.
- Unix exec()
  - Allows a child to run a new program.
  - There is no system call or library function with the name exec. There is a family of functions, and we refer to them as exec.
- Unix wait()
  - Allow a parent to wait for a child to terminate.

Why Create a New Process?

- Run a new program:
  - E.g., shell executing a program entered at command line.
  - Or, even running an entire pipeline of commands
  - Such as "**`wc -l * | sort | uniq -c | sort -nr`**"
- Run a new thread of control for the same program:
  - E.g., a Web server handling a new Web request
  - While continuing to allow more requests to arrive.
  - Essentially time sharing the computer.
- Underlying mechanism:
  - A process executes **`fork()`** to create a child process.
  - (Optionally) child does **`exec()`** of a new program.

Creating a New Process

- Cloning an existing process.
  - Parent process creates a new child process.
  - The two processes then run concurrently.
- Child process inherits state from parent.
  - Identical (but separate) copy of virtual address space.
  - Copy of the parent's open file descriptors.
  - Parent and child share access to open files.
- Child then runs independently.
  - Executing independently, including invoking a new program.
  - Reading and writing its own address space.



Fork System-Level Function

- `fork()` is called once.
  - But returns twice, once in each process.
- Telling which process is which.
  - Parent: `fork()` returns the child's process ID.
  - Child: `fork()` returns 0.

```
pid = fork();
if (pid > 0){
  /* in parent */   …
}else if(pid == 0){
  /* in child */ …
else {
    /* Error */ …
}
```

Fork and Process State

- Inherited
  - User and group IDs
  - Signal handling settings
  - stdio
  - File pointers
  - Root directory
  - File mode creation mask
  - Resource limits
  - Controlling terminal
  - All machine register states
  - Control register(s)
  - ...
- Separate in child
  - Process ID
  - Address space (memory)
  - File descriptors
  - Parent process ID
  - Pending signals
  - Time signal reset times
  - ...

Example: What Output?

```
int main(void)
{ pid_t pid;
  int x = 1;

  pid = fork();
  if( pid > 0 ) {
     printf("parent: x = %d\n", --x); exit(0);
   } else if( pid < 0 ){
             printf("Error\n"); exit(0);
   } else {
            printf("child: x = %d\n", ++x);
            exit(0);
           }
}
```

Executing a New Program

- `fork()` copies the state of the parent process.
  - Child continues running the parent program
  - … with a copy of the process memory and registers.
- Need a way to invoke a new program.
  - In the context of the newly-created child process
- Example:

NULL-terminated array

Contains command-line arguments (to become "argv[]" of ls)

program

```
execvp("ls", argv);
fprintf(stderr, "exec failed\n");
exit(EXIT_FAILURE);
```

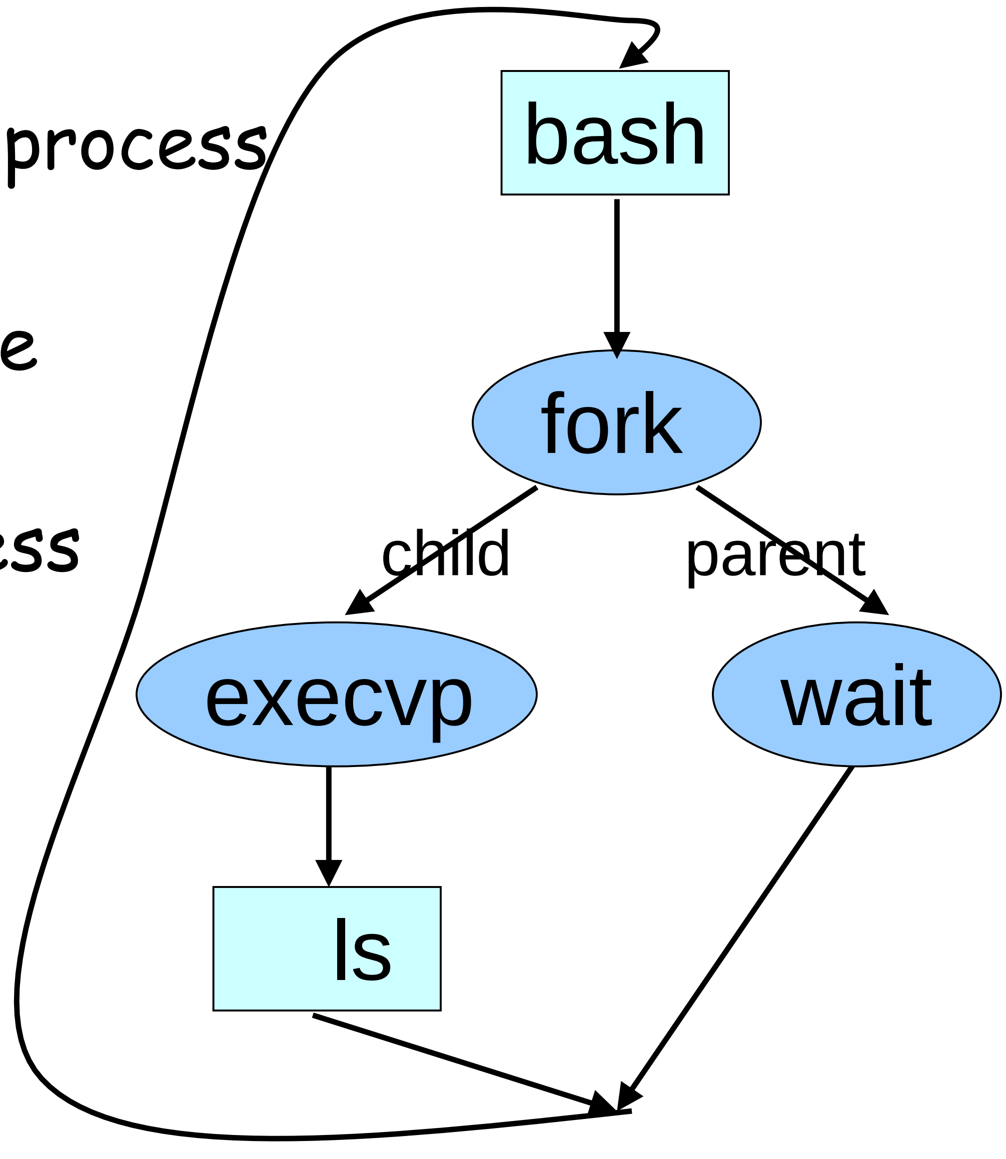
Waiting for the Child to Finish

- Parent should wait for children to finish.
  - Example: A shell waiting for operations to complete.
- Waiting for a child to terminate: **`wait()`**
  - Blocks until some child terminates.
  - Returns the process ID of the child process.
  - Or returns -1 if no children exist (i.e., already exited).
- Waiting for specific child to terminate: **`waitpid()`**
  - Blocks till a child with particular process ID terminates.

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);
```

Example: A Simple Shell

- Shell is the parent process
  - E.g., bash
- Parses command line
  - E.g., “ls -l”
- Invokes child process
  - **`fork()`**, **`execvp()`**
- Waits for child
  - **`wait()`**



Process Creation Using Fork

```
int main ()
{
   pid_t pid;
   int status = 0;
   pid = fork();
   if (pid < 0)
     perror("fork()");
   if (pid > 0) {
        /* parent */
        printf("I am parent\n");
    } else {
       /* child */
         printf("I am child\n");
         exit(status);
    }
}
```

The fork system call returns twice: it returns a zero to the child and the child process ID (pid) to the parent.

The perror function produces a message on the standard error output describing the last error encountered during a call to a system or library function (man page).

pid is zero which indicates a child process.

Fork System Call

- If fork () succeeds it returns the child PID to the parent and returns 0 to the child.
- If fork() fails, it returns -1 to the parent (no child is created) and sets errno.
- A program almost always uses this difference to do different things in the parent and child processes.
- Failure occurs when the limit of processes that can be created is reached.
- pid_t is defined as typedef int pid_t
  - We will use pid_t and int interchangeably.
- Other calls:
  - int getpid() – returns the PID of calling process.
  - int getppid() – returns the PID of parent process.

Wait

- Parents waits for a child (system call).
  - Blocks until a child terminates.
  - Returns pid of the child process.
  - Returns -1 if no child process exists (already exited).
  - status

```
#include <sys/types.h>
#include <sys/wait.h>
   pid_t wait(int *status)
```

- Parent waits for a specific child to terminate.

```
pid_t waitpid(pid_t pid, int *status, int options)
```

Process Creation Using Fork

```
int main ()
{
  pid_t pid;
  int status = 0;
  pid = fork();
  if (pid < 0)
    perror("fork()");
  if (pid > 0) {
      /* parent */
      printf("I am parent\n");
      pid = wait(&status);
   } else {
      /* child */
      printf("I am child\n");
      exit(status);
   }
}
```

The fork syscall returns twice: it returns a zero to the child and the child process ID (pid) to the parent.

Parent uses wait to sleep until the child exits; wait returns child pid and status.

wait variants allow wait on a specific child, or notification of stops and other signals.

fork() Example

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int main()
{
    pid_t pid;
    int i;
    pid = fork();
    if( pid > 0 )
   {
      /* parent */
     for( i=0; i < 1000; i++ )
       printf("\t\t\tPARENT %d\n", i);
   }
else
{
   /* child */
   for( i=0; i < 1000; i++ ) {
     printf( "CHILD %d\n", i );
   }
   return 0;
}
}
```

What is the possible output?

fork () Example:Possible Output

```
                  PARENT 0
                  PARENT 1
                  PARENT 2
                  PARENT 3
                  PARENT 4
                  PARENT 5
                  PARENT 6
                  PARENT 7
                  PARENT 8
                  PARENT 9
CHILD 0
CHILD 1
CHILD 2
CHILD 3
CHILD 4
CHILD 5
CHILD 6
CHILD 7
CHILD 8
CHILD 9
```

fork () Example:Possible Output

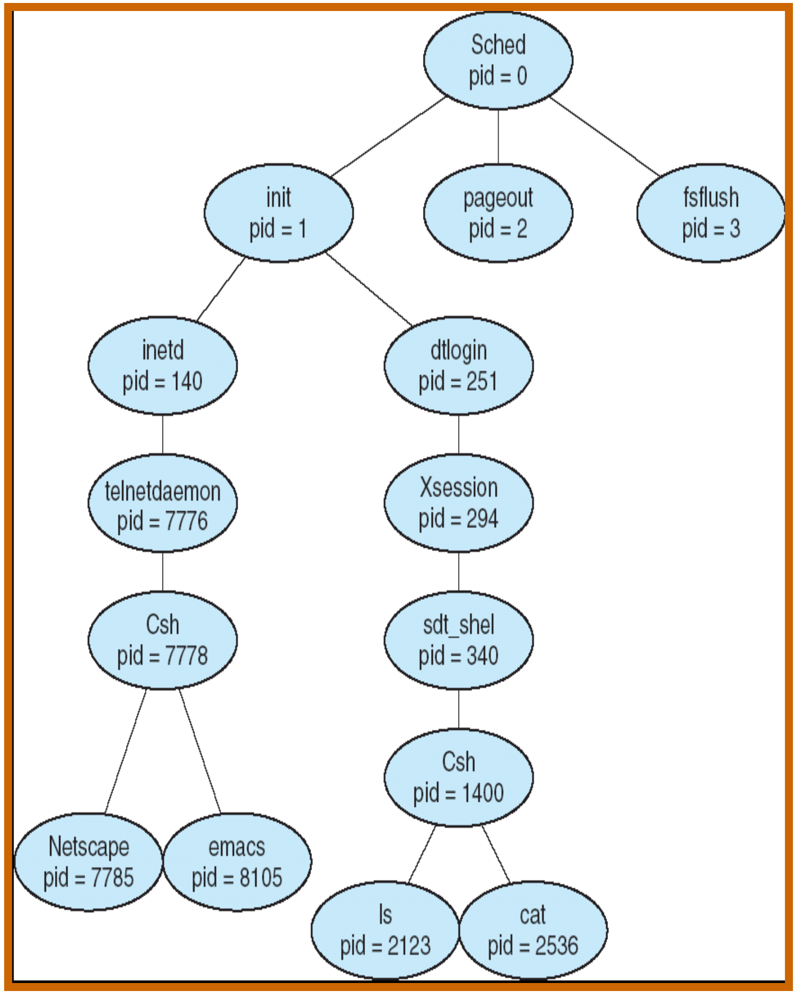
```
                 PARENT 0
                 PARENT 1
                 PARENT 2
                 PARENT 3
                 PARENT 4
                 PARENT 5
                 PARENT 6
CHILD 0
CHILD 1
CHILD 2          PARENT 7
                 PARENT 8
                 PARENT 9

CHILD 3
CHILD 4
CHILD 5
CHILD 6
CHILD 7
CHILD 8
CHILD 9
```

Lots of possible outputs!!

Execution

- Processes get a share of the CPU before giving it up to give another process a turn.
- The switching between the parent and child depends on many factors:
  - machine load, system process scheduling.
- Output interleaving is nondeterministic
  - Cannot determine output by looking at code.

More about Process Operations



- In Unix-based systems, a hierarchy of processes is formed.
- In Unix, we can obtain a listing of processes by using the ps command.
- ps -el will list complete information for all processes.

Exec

- The term exec refers to a family of functions where each of the functions replace a process's program (the one calling one of the exec functions) with a new loaded program.
- A call to a function from exec loads a binary file into memory (destroying the memory image of the program calling it).
- The new program starts executing from the beginning (where main begins).
- On success, exec never returns; on failure, exec returns -1.
- The different versions are different primarily in the way parameters are passed.

Exec

- The exec family consists of these functions: execvp, execlp, execv, execve, execl, execle.
  - Functions with *p* in their name (execvp, execlp) search for the program in current path; functions without *p* must be given full path.
  - Functions with *v* in their name (execv, execvp, execve) differ from functions with "l" (execl, execlp, execle) in the way arguments are passed.
  - Functions with *e* accept array of environment variables.

Versions of exec

- Versions of exec offered by C library:

```
int execl( const char *path, const char *arg, ... );
int execlp( const char *file, const char *arg, ... );
int execle( const char *path, const char *arg
              , ..., char *const envp[] );
int execv( const char *path, char *const argv[] );
int execvp( const char *file, char *const argv[] );
int execve( const char *filename, char *const argv [], char *const
              envp[] );
```

Exec Example

Program A:

```
int i = 5;
printf("%d\n",i);
execl("B", "B", NULL);
printf("%d\n",i);
```

Program B:

```
main()
{
    printf("hello\n");
}
```

- What is the output of program A?
  5
  hello
- Why is it not this?
  5
  hello
  5
- The exec command replaces the instructions in the process with instructions for program B. It starts at the first instruction (starts at main).

Exec Example

Program A:

```
int i = 5;
prog_argv[0] = "B";
prog_argv[1] = NULL;
printf("%d\n",i);
execv(prog_argv[0],
        prog_arg);
printf("%d\n",i);
```

Program B:

```
main()
{
   printf("hello\n");
}
```

- Same functionality as the program on the previous slide.
- Used execv instead of execl.
- execv uses an array to pass arguments.
- execl uses a list to pass arguments.
- If you use execv you must provide the full path name e.g.,

prog_argv[0] = "/eecs/courses/OS/CodeExamples/B"

Exec Example

```
int main(int argc, char *argv[])
{

  char *prog1_argv[4];
  int i = 5;

  execlp("ls","ls","-l","a.c",NULL);

  perror("execlp\n");
  printf("%d\n",i);

}
```

- In this example, note that the command is
  ls -l a.c
- Each argument is in the list.
- Question:
  - What would cause the perror function to be executed?

Exec Example

```
int main(int argc, char *argv[])
{

  char *prog1_argv[4];
  int i = 5;

  prog1_argv[0] = "ls";
  prog1_argv[1] = "-l";
  prog1_argv[2] = "a.c";
  prog1_argv[3] = NULL;

  execvp(prog1_argv[0],
     prog1_argv);

  perror("execvp\n");
  printf("%d\n",i);

}
```

- Same example as that on the previous slide but execvp is used which requires an array.

Fork and Exec

- Child process may choose to execute some other program than the parent by using one of the exec calls.
- `exec` overlays a new program on the existing process.
- Child will not return to the old program unless exec fails. This is an important point to remember.
- File descriptors are preserved.

Example

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
int main()
{
   pid_t pid;
   pid = fork();
     if (pid < 0)
        perror("fork()");
if (pid > 0)
 {
    wait(NULL);
    printf("Child Complete");
 } else{
   if (pid == 0)
     execlp("ls","ls", "-l, "a.c",
                  NULL);
 }
}
```