Operating Systems and Unix

This document gives a general overview of the work done by an operating system and gives specific examples from UNIX.

O.S. Responsibilities

- Manages Resources:
  - I/O devices (disk, keyboard, mouse, terminal)
  - Memory
- Manages Processes:
  - process creation, termination
  - inter-process communication
  - multi-tasking (scheduling processes)

Unix

- We will focus on the Unix operating system.
- There are many *flavors* of Unix, but the libraries that provide access to the kernel are pretty standard (although there are minor some differences between different flavors of Unix).

Posix - Portable Operating System Interface

- Posix is a popular standard for Unix-like operating systems.
- Posix is actually a *collection* of standards that cover system calls, libraries, applications and more…
- Posix 1003.1 defines the C language interface to a Unix-like kernel.

Posix and Unix

- Most current Unix-like operating systems are *Posix compliant* (or nearly so).

  Linux, BSD, Solaris, IRIX

- We won't do anything fancy enough that we need to worry about specific versions/flavors of Unix (any Unix will do).

Posix 1003.1

- process primitives
  - creating and managing processes
- managing process environment
  - user ids, groups, process ids, etc.
- file and directory I/O
- terminal I/O
- system databases (passwords, etc)

System Calls

- A *system call* is an interface to the kernel that makes some request for a service.
- The actual implementation (how a program actually contacts the operating system) depends on the specific version of Unix and the processor.
- The C interface to system calls is standard (so we can write an program and it will work anywhere).

Unix Processes

- Every process has the following attributes:
  - a *process id* (a small integer)
  - a *user id* (a small integer)
  - a *group id* (a small integer)
  - a *current working directory*.
  - a chunk of memory that hold name/value pairs as text strings (the *environment variables*).
  - a bunch of other things…

User IDs

- User ids and group ids are managed via a number of functions that deal with system databases.
- The *passwsd* database is one example.
- When you log in, the system looks up your user id in the passwd database.

passwd database

- For each user there is a record that includes:
  - user's login name
  - user id (a small integer – 16 bits)
  - group id
  - home directory
  - shell
  - encrypted password
  - user's full name

C data structure for passwd DB

```
struct passwd {
  char *pw_name;    /* user login name */
  char *pw_passwd;  /* encrypted password */
  int  pw_uid;      /* user id */
  int  pw_gid;      /* group id */
  char *pw_gecos;   /* full name */
  char *pw_dir;     /* home directory */
  char *pw_shell;   /* shell */
}
```

passwd access functions

```
#include <pwd.h>

/* given a user id – get a record */
struct passwd *getpwuid( int uid);

/* given a login name – get a record */
struct passwd *getpwnam( char *name);
```

The Filesystem

- The Unix filesystem is based on *directories* and *files*.

- Directories are like folders (for you Windows Weenies).

Files and File Names

- Every file has a name.
- Unix file names can contain any characters (although some make it difficult to access the file).
- Unix file names can be long!
  - how long depends on your specific flavor of Unix

File Contents

- Each file can hold some raw data.
- Unix does not impose any structure on files
  - files can hold any sequence of bytes.
- Many programs *interpret* the contents of a file as having some special structure
  - text file, sequence of integers, database records, etc.

Directories

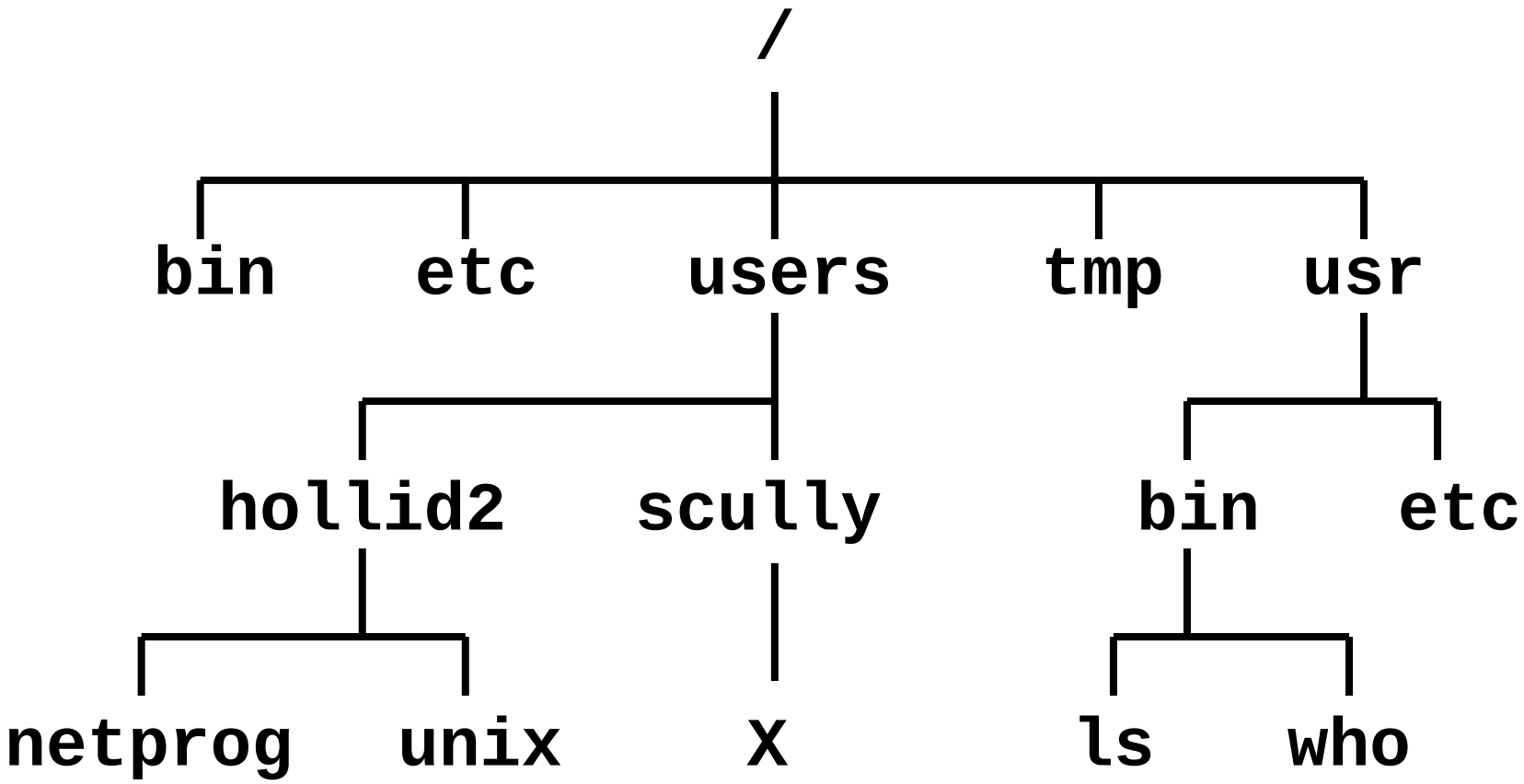
- A directory is a special kind of file - Unix uses a directory to hold information about other files.

- We often think of a directory as a container that holds other files (or directories).

More about File Names

- Review: every file has a name.
- Each file *in* the same directory must have a unique name.
- Files that are in different directories can have the same name.

The Filesystem



Unix Filesystem

- The filesystem is a hierarchical system of organizing files and directories.
- The top level in the hierarchy is called the "root" and *holds* all files and directories.
- The name of the root directory is **/**

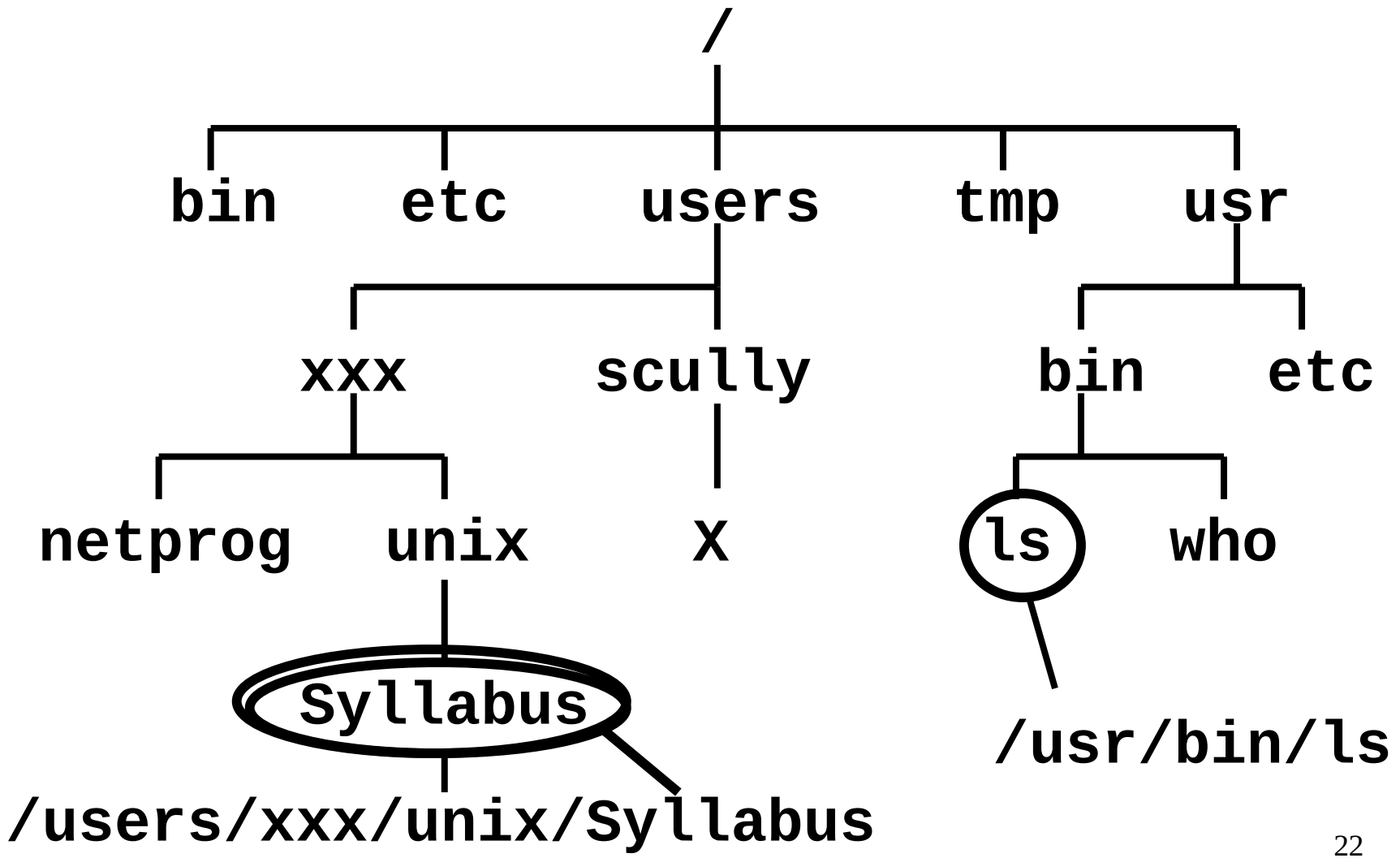
Pathnames

- The *pathname* of a file includes the file name and the name of the directory that holds the file, and the name of the directory that holds the directory that holds the file, and the name of the … up to the root
- The pathname of every file in a Unix *filesystem* is unique.

Pathnames (cont.)

- To create a pathname you start at the root (so you start with "/"), then follow the path down the hierarchy (including each directory name) and you end with the filename.
- In between every directory name you put a "/".

Pathname Examples



Absolute Pathnames

- The pathnames described in the previous slides start at the *root*.
- These pathnames are called "absolute pathnames".
- We can also talk about the pathname of a file *relative* to a directory.

Relative Pathnames

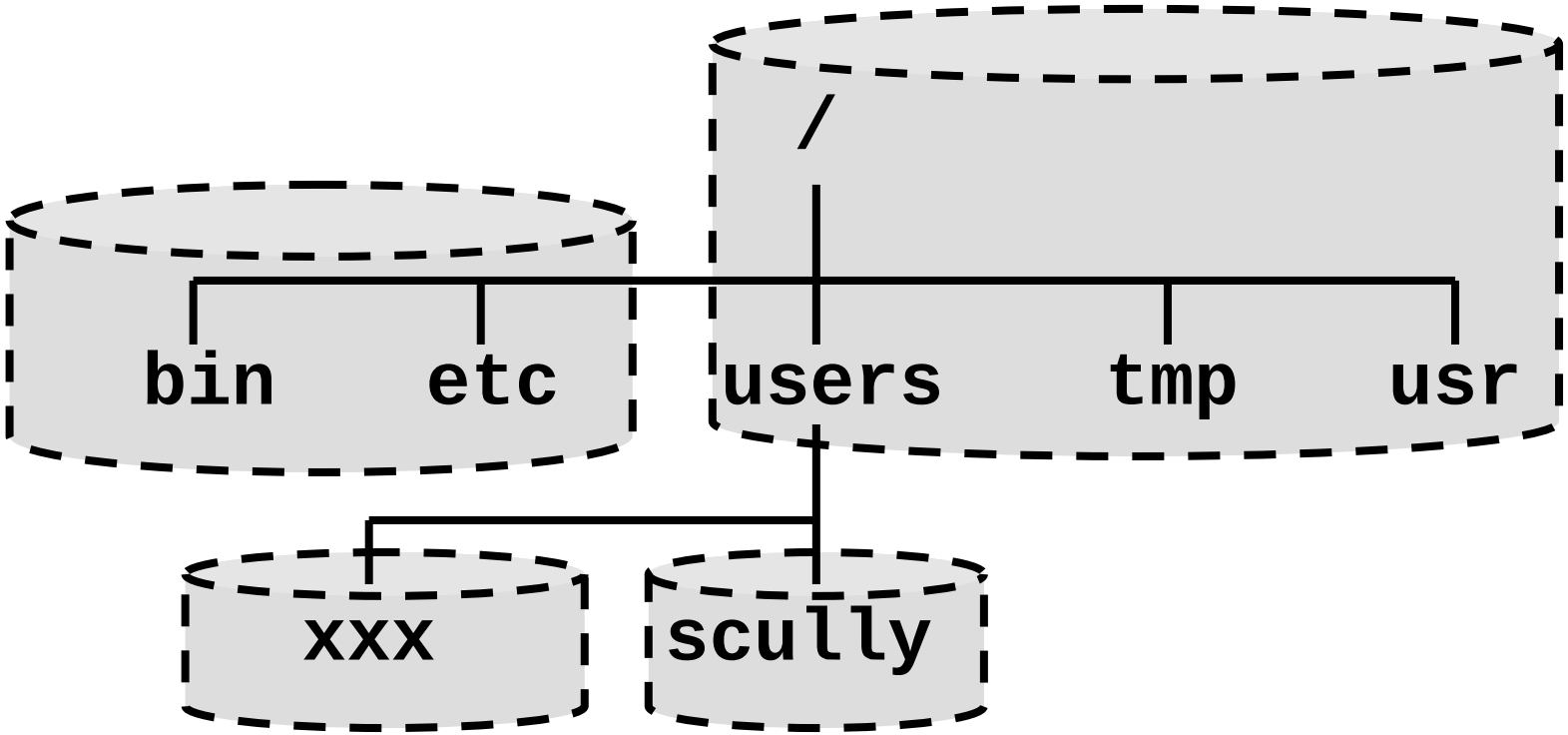
- If we are *in* the directory /users/hollid2, the relative pathname of the file **`Syllabus`** is:

`unix/Syllabus`

- Most unix commands deal with pathnames!
- We will usually use relative pathnames when specifying files.

Disk vs. Filesystem

- The entire hierarchy can actually include many disk drives.
  - some directories can be on other computers



Processes and CWD

- Every process runs *in a directory*.

- This attribute is called the

  current working directory

Finding out the CWD

```
char *getcwd(char *buf, size_t size);
```

Returns a string that contains the *absolute* pathname of the current working directory.

There are functions that can be used to change the current working directory (**`chdir`**).

Finding out some other process attribute values

```
#include <unistd.h>
pid_t getpid(void);

uid_t getuid(void);
```

Creating a Process

- The only way to create a new process is to issue the **`fork()`** system call.

- **`fork()`** *splits* the current process in to 2 processes, one is called the *parent* and the other is called Regis.
- Actually it's called the *child*.

Parent and Child Processes

- The child process is a *copy* of the parent process.
- Same program.
- Same place in the program (almost – we'll see in a second).
- The child process gets a new process ID.

Process Inheritence

- The child process *inherits* many attributes from the parent, including:
  - current working directory
  - user id
  - group id

The `fork()` system call

```
#include <unistd.h>
pid_t fork(void);
```

`fork()` returns a process id (a small integer).

`fork()` returns twice!

In the parent – **`fork`** returns the id of the child process.

In the child – **`fork`** returns a 0.

Example

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

void main(void) {
  if (fork())
     printf(“I am the parent\n”);
  else
     printf(“I am the child\n”);
  printf(“I am the walrus\n”);
}
```

Bad Example (don't try this!)

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

void main(void) {
  while (fork()) {
    printf("I am the parent %d\n“
           ,getpid());
  }
  printf("I am the child %d\n“
         ,getpid());
}
```

fork bomb!

I told you so…

- Try pressing Ctrl-C to stop the program.
- It might be too late.
- If this is your own machine – try rebooting.
- If this is a campus machine – run for your life. If they catch you – deny everything.

Try listening next time…

Switching Programs

- **`fork()`** is the only way to create a new process.
- This would be almost useless if there was not a way to switch what *program* is associated with a process.
- The **`exec()`** system call is used to start a new program.

exec

- There are actually a number of exec functions:
  `execlp execl execle execvp execv execve`
- The difference between functions is the parameters… (how the new program is identified and some attributes that should be set).

The exec family

- When you call a member of the exec family you give it the pathname of the executable file that you want to run.
- If all goes well, exec will never return!
- The process *becomes* the new program.

execl()

```
int execl(char *path,
          char *arg0,
          char *arg1, …,
          char *argn,
          (char *) 0);

execl(“/home/chrisc/reverse”,
        “reverse”, “Hidave”,NULL);
```

A complete `execl` example

```
#include <unistd.h>  /* exec, getcwd */
#include <stdio.h>   /* printf */

/* Exec example code */
/* This program simply execs "/bin/ls" */

void main(void) {
  char buf[1000];

  printf(“Here are the files in %s:\n",
         getcwd(buf,1000));
  execl("/bin/ls","ls","-al",NULL);
  printf("If exec works, this line won't be
   printed\n");
}
```

`fork()` and **`exec()`** together

- Program does the following:
  - **`fork()`** - results in 2 processes
  - parent prints out it's **`PID`** and waits for child process to finish (to exit).
  - child prints out it's **`PID`** and then **`exec`**s "**`ls`**" and exits.

execandfork.c part 1

```
#include <unistd.h>     /* exec, getcwd */
#include <stdio.h>      /* printf */
#include <sys/types.h> /* need for wait */
#include <sys/wait.h>  /* wait() */
```

execandfork.c part 2

```
void child(void) {
  int pid = getpid();

  printf("Child process PID is %d\n",pid);
  printf("Child now ready to exec ls\n");
  execl("/bin/ls","ls",NULL);
}
```

execandfork.c part 3

```
void parent(void) {
  int pid = getpid();
  int stat;

  printf("Parent process PID is %d\n",pid);
  printf("Parent waiting for child\n");
  wait(&stat);
  printf("Child is done. Parent now
  transporting to the surface\n");
}
```

execandfork.c part 4

```
void main(void) {
  printf("In main - starting things with a
  fork()\n");
  if (fork()) {
     parent();
  } else {
     child();
  }
  printf("Done in main()\n");
}
```

execandfork.c output

```
> ./execandfork
In main - starting things with a fork()
Parent process PID is 759
Parent process is waiting for child
Child process PID is 760
Child now ready to exec ls
exec            execandfork    fork
exec.c          execandfork.c  fork.c
Child is done. Parent now transporting to
  the surface
Done in main()
>
```