Basic UNIX Concepts

What is UNIX good for?

- Supports many users running many programs at the same time, all sharing (transparently) the same computer system
- Promotes information sharing
- More than just used for running software … geared towards facilitating the job of creating new programs. So UNIX is “expert friendly”
- Got a bad reputation in business because of this aspect

History (Introduction)

- 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 …
- See Wang, page 1 for a brief discussion of UNIX variations
- POSIX (potable operating system interface) - IEEE, ANSI

Some Terminology

- *Program*: executable file on disk
- *Process*: executing instance of a program
- *Process ID*: unique, non-negative integer identifier (a handle by which to refer to a process)
- *UNIX kernel*: a C program that implements a general interface to a computer to be used for writing programs (p6)
- *System call*: well-defined entry point into kernel, to request a service
- *UNIX technique*: for each system call, have a function of same name in the standard C library
  - user process calls this function
  - function invokes appropriate kernel service

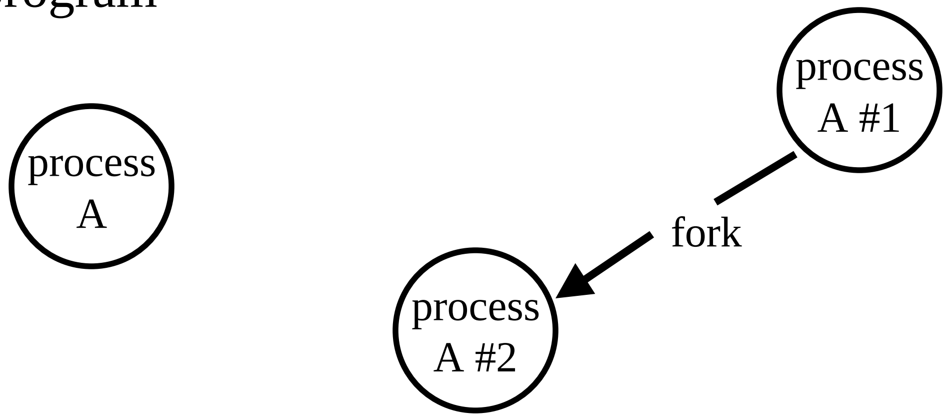
Concurrency

- Most modern developments in computer systems & applications rely on:
  - *communication*: the conveying of info by one entity to another
  - *concurrency*: the sharing of resources in the same time frame

    note: concurrency can exist in a single processor system as well as in a multiprocessor system.
- Managing concurrency is difficult, as execution behaviour (e.g. relative order of execution) is not always reproducible
- More details on this in the last 1/3 or the course

Fork (11.10)

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



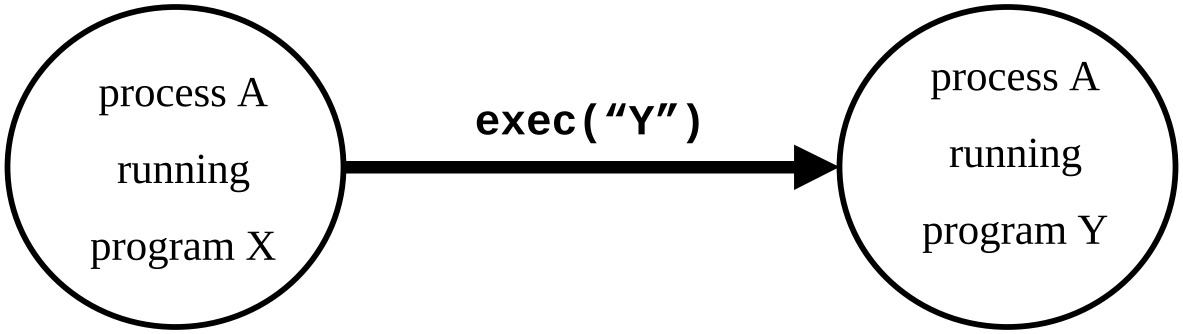
- The duplicate (*child process*) and the original (*parent process*) both proceed from the point of the fork with exactly the same data
- The only difference between the two processes is the *fork* return value, i.e. *(… see next slide)*

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 (11.11)

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



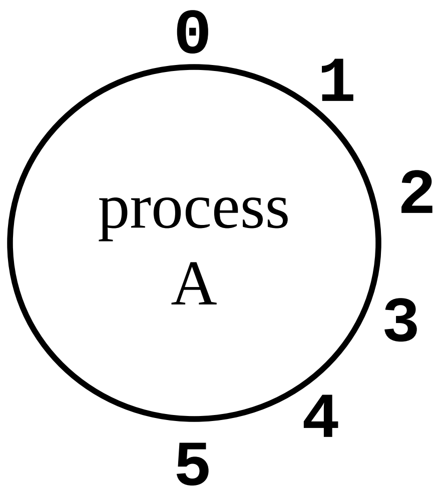
- Variations on exec: **`execl()`**, **`execv()`**, etc. which will be discussed later in the course
- On *success*, exec never returns; on *failure*, exec returns with value -1

Exec example

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

exec( “Y” );

i = 6;
printf( “%d\n”, i );

PROGRAM Y
printf( “hello” );
```

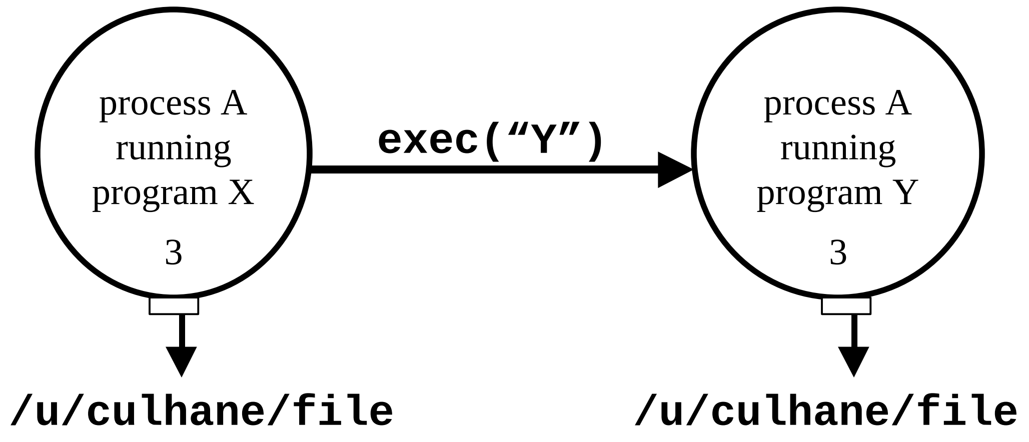
Processes and File Descriptors

- File descriptors (11.1) belong to processes, not programs
- They are a process' link to the outside world



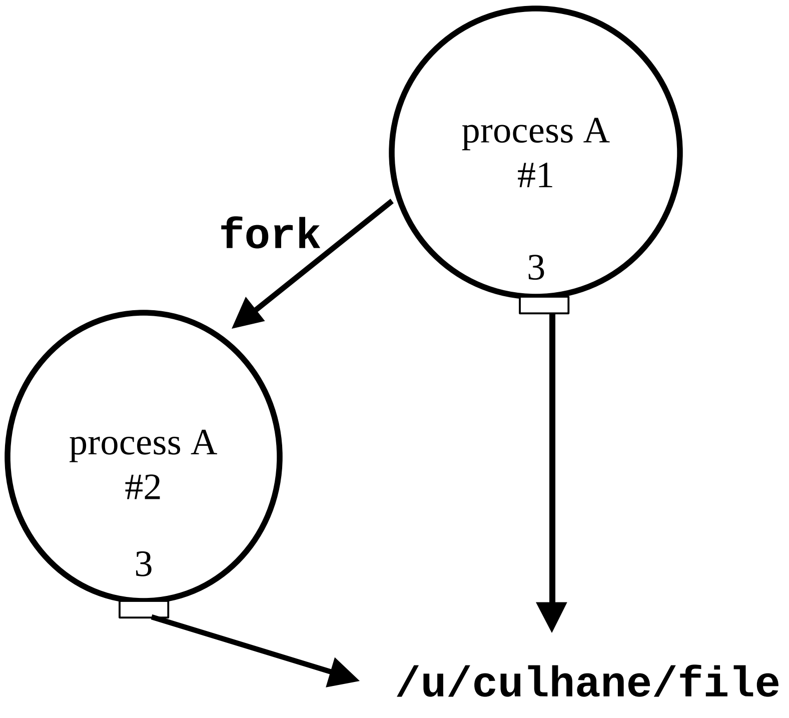
PIDs and FDs across an exec

- File descriptors are maintained across *exec* calls:



PIDs and FDs across a fork

- File descriptors are maintained across *fork* calls:

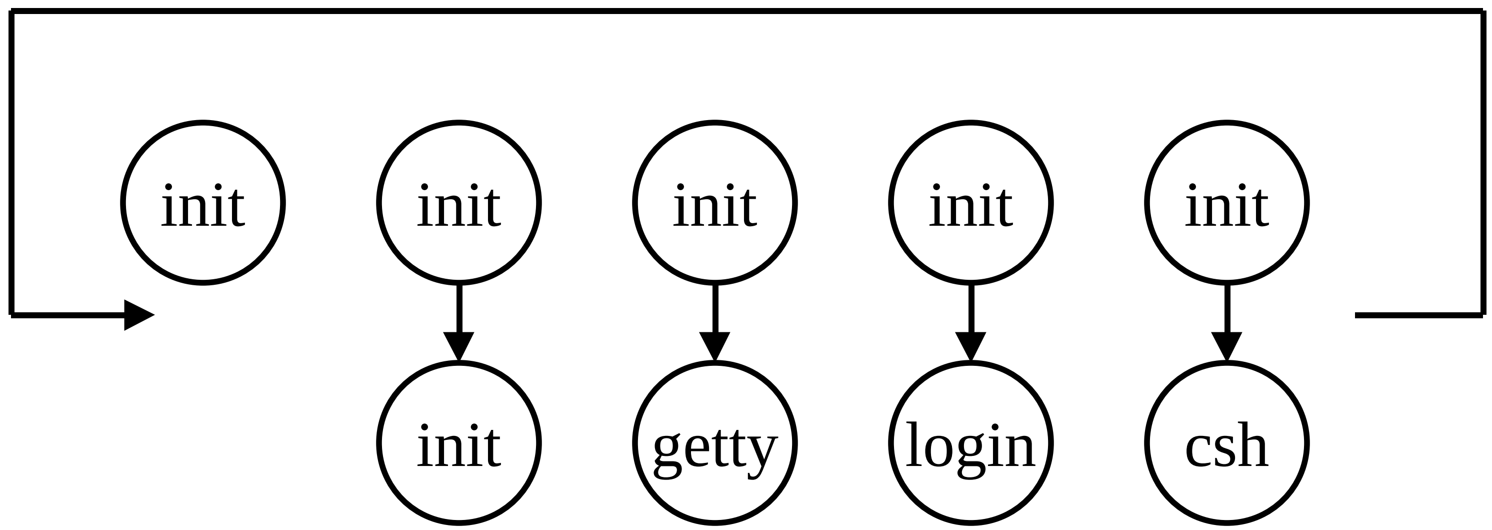


More UNIX Concepts

Initializing UNIX

- The first UNIX program to be run is called “**`/etc/init`**” (11.17)
- It *forks* and then *exec*s one “**`/etc/getty`**” per terminal
- 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 shell, the process dies. Init finds out about it (*wait* system call), and *forks* another process for that terminal

Initializing UNIX

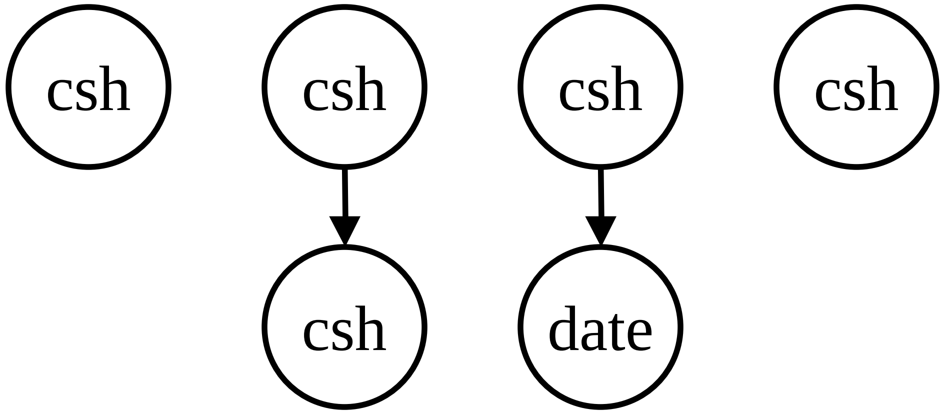


- See “**`top`**”, “**`ps -aux`**”, etc. to see what’s running at any given time
- The only way to create a new process is to duplicate an existing process, therefore the ancestor of ALL processes is **`init`**, with pid=1

How csh runs commands

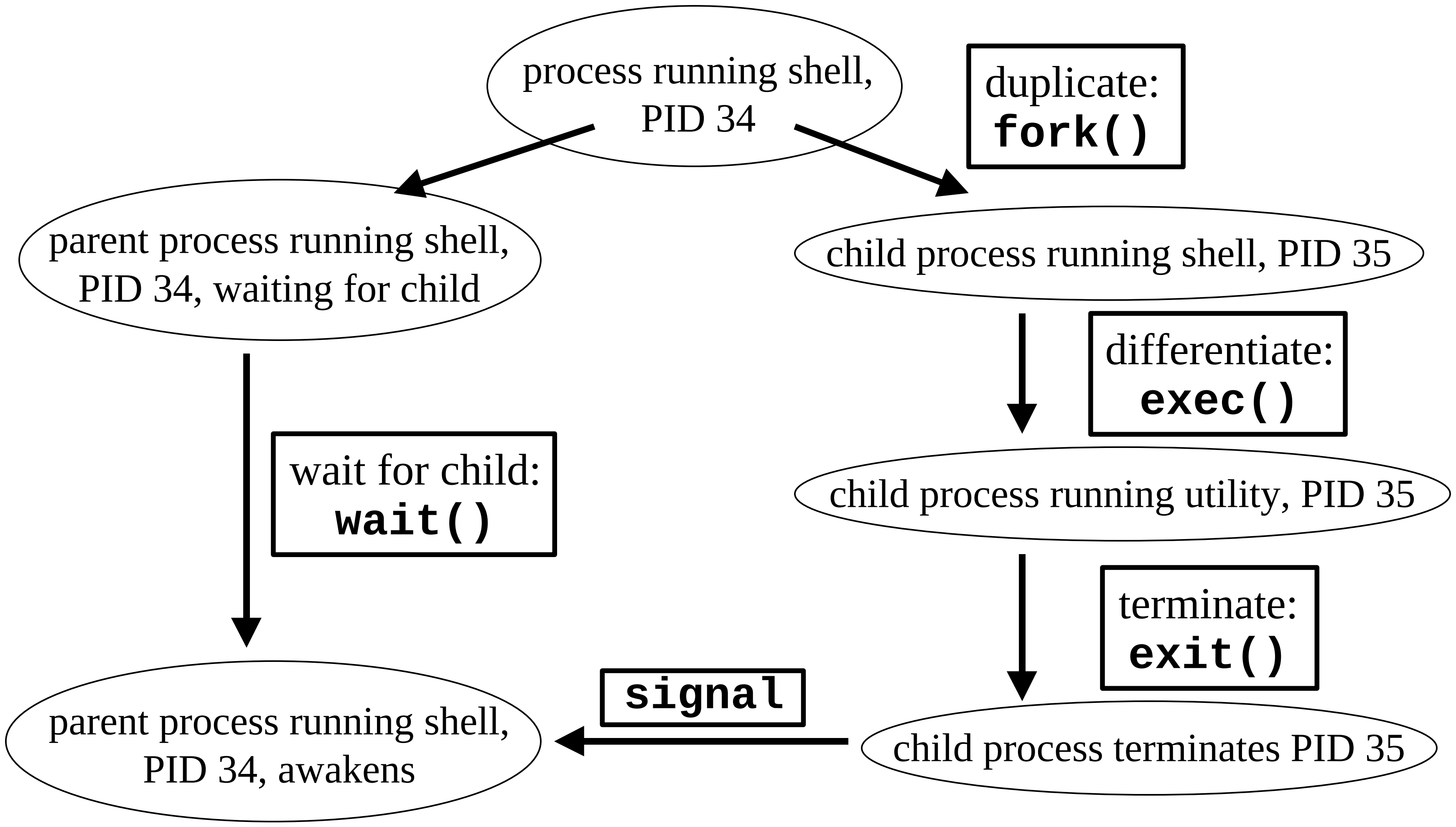
```
> date
Sun May 25 23:11:12 EDT 1997
```

- 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 csh runs (cont.)


process running shell, PID 34
duplicate: fork()
parent process running shell, PID 34, waiting for child
child process running shell, PID 35
differentiate: exec()
wait for child: wait()
child process running utility, PID 35
terminate: exit()
signal
parent process running shell, PID 34, awakens
child process terminates PID 35

Fork: PIDs and PPIDs (11.10)

- 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( 1 );
}
```

Concurrency Example

Program a:

```
#!/usr/bin/csh -f
@ count = 0
while( $count < 200 )
   @ count++
   echo -n "a"
end
```

Program b:

```
#!/usr/bin/csh -f
@ count = 0
while( $count < 200 )
   @ count++
   echo -n "b"
end
```

- When run *sequentially* (**`a;b`**) output is as expected
- When run *concurrently* (**`a&;b&`**) output is interspersed, and re-running it may produce different output

Producer/Consumer Problem

- Simple example:

```
who | wc -l
```

- Both the writing process (**`who`**) and the reading process (**`wc`**) of a pipeline execute concurrently
- A pipe is usually implemented as an internal OS buffer
- It is a resource that is concurrently accessed by the reader and by the writer, so it must be managed carefully

Producer/Consumer (cont.)

- *consumer* should be *blocked* when buffer is empty
- *producer* should be *blocked* when buffer is full
- producer and consumer should run independently so far as the buffer capacity and contents permit
- producer and consumer should never both be updating the buffer at the same instant (otherwise, data integrity cannot be guaranteed)
- producer/consumer is a harder problem if there is more than one consumer and/or more than one producer

Machine Language

- CPU interprets machine language programs:

```
1100101 11111111 11100110 00000000
1010001 00000010 01011101 00000000
1100101 00000000 11111111 00100100
```

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

```
MOVE   FFFFDC00, D0          % b = a * 2
MUL    #2, D0
MOVE   D0, FFFDC04
```

Compilation

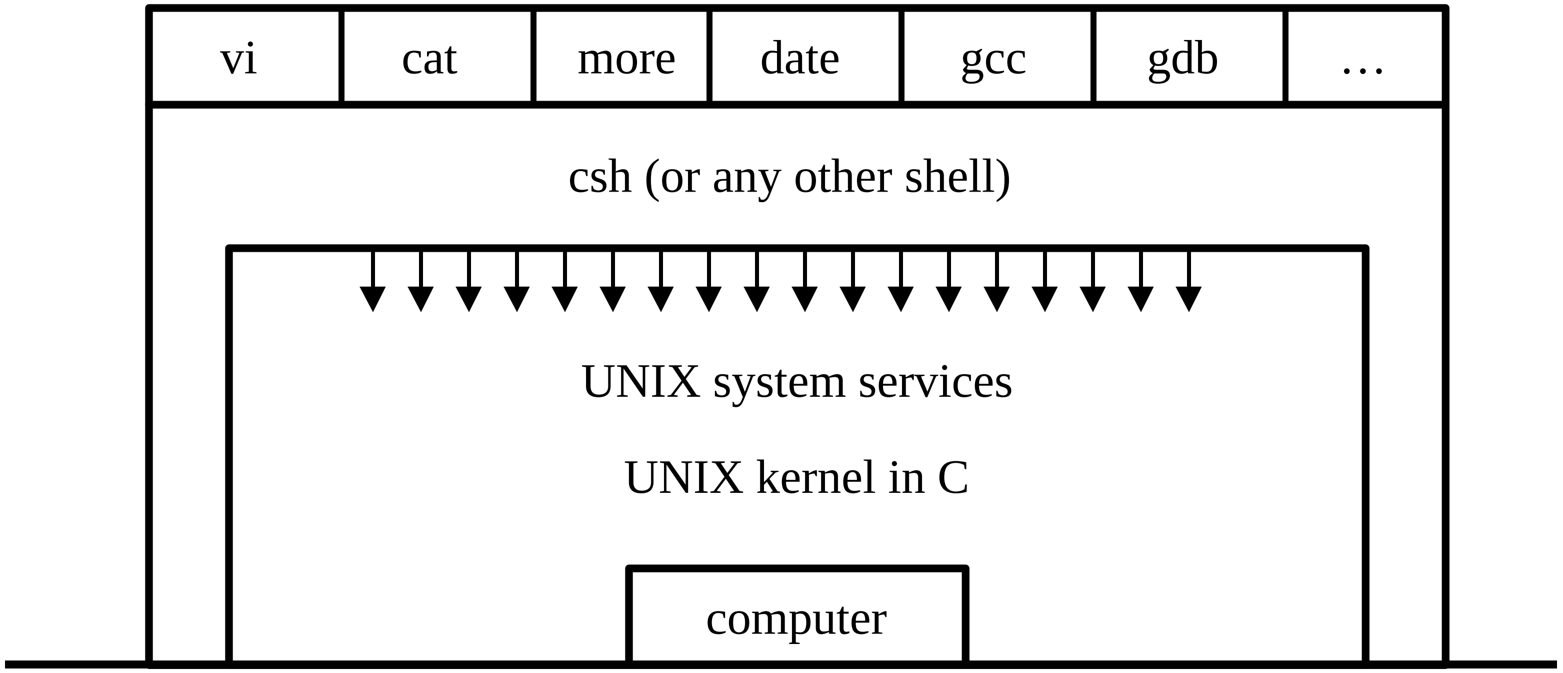
- High Level Language (HLL) is a language for expressing algorithms whose meaning is (for the most part) independent of the particular computer system being used
- A *compiler* translates a high-level language into object files (machine language modules).
- A *linker* translates object files into a *machine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

```
gcc -c fork.c -o fork.o
```

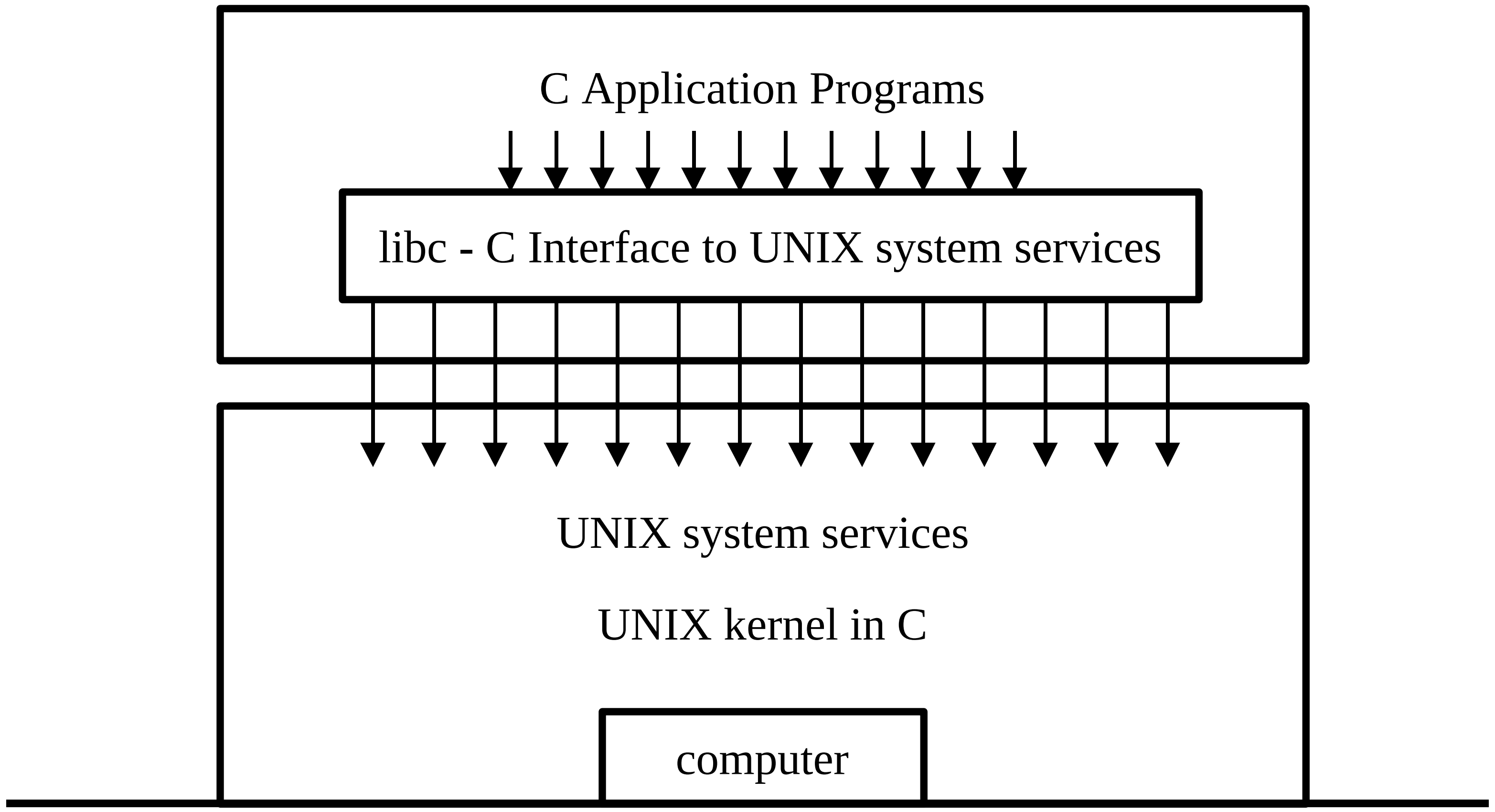
  - create executable file "**`fork`**" from object file "**`fork.o`**":

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

Tools and Applications



C and libc


C Application Programs
libc - C Interface to UNIX system services
UNIX system services
UNIX kernel in C
computer

Miscellaneous

- We haven't gone over these in any detail yet:
  - **`ln`** (*symbolic links*)
  - **`chmod`** (*permissions*)
  - **`man -k fork`** and **`man 2 fork`** (*ie: viewing specific pages*)
  - **`du`** (*disk space usage)*
  - **`quota -v username`** and **`pquota -v username`**
  - **`noglob`**
  - *… any others ?????*