CPSC 457

Processes - part 1

Contains slides from Mea Wang, Andrew Tanenbaum and Herbert Bos

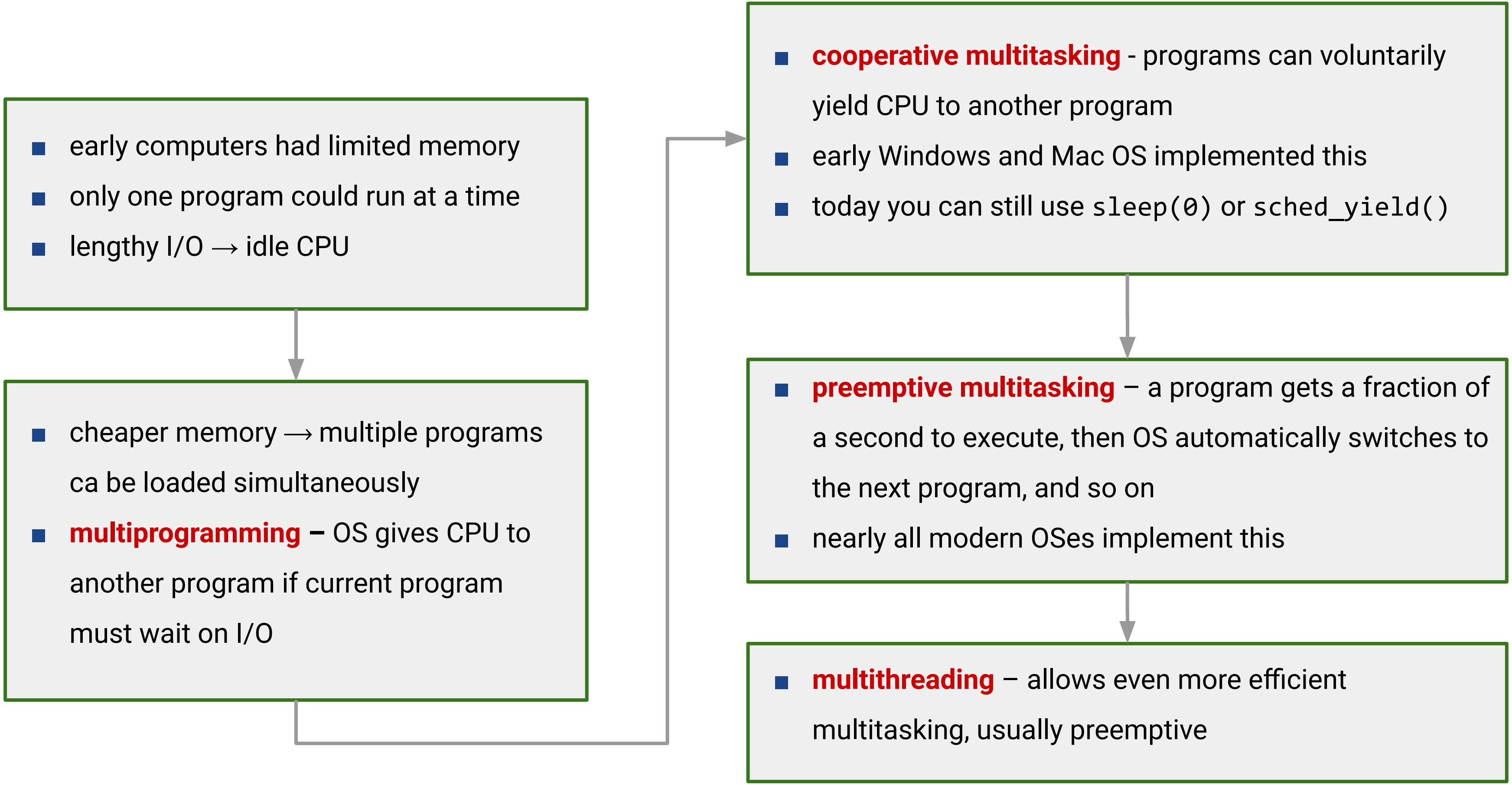
Multitasking

- multitasking = concurrent execution of multiple programs
- allows computer to run **N** processes with **M** CPUs, even if **N** > **M**
- works even with a single CPU:

```
repeat forever - using a timer interrupt
    give CPU to program (i) for a short time
    give CPU to program (i+1) for a short time
    give CPU to program (i+2) for a short time
    ...
```

- just an illusion of parallelism – programs do not have to execute their instructions exactly at the same time, as long as it appears that way
- just like multiprogramming:
  - multitasking allows us to reduce CPU idling during I/O,
    CPU can be given to another program rather than remain idle
  - multitasking is only practical when memory is big enough to hold multiple running programs

Multiprogramming and multitasking



- early computers had limited memory
- only one program could run at a time
- lengthy I/O → idle CPU

- cheaper memory → multiple programs ca be loaded simultaneously
- **multiprogramming** – OS gives CPU to another program if current program must wait on I/O

- **cooperative multitasking** - programs can voluntarily yield CPU to another program
- early Windows and Mac OS implemented this
- today you can still use `sleep(0)` or `sched_yield()`

- **preemptive multitasking** – a program gets a fraction of a second to execute, then OS automatically switches to the next program, and so on
- nearly all modern OSes implement this

- **multithreading** – allows even more efficient multitasking, usually preemptive

Process

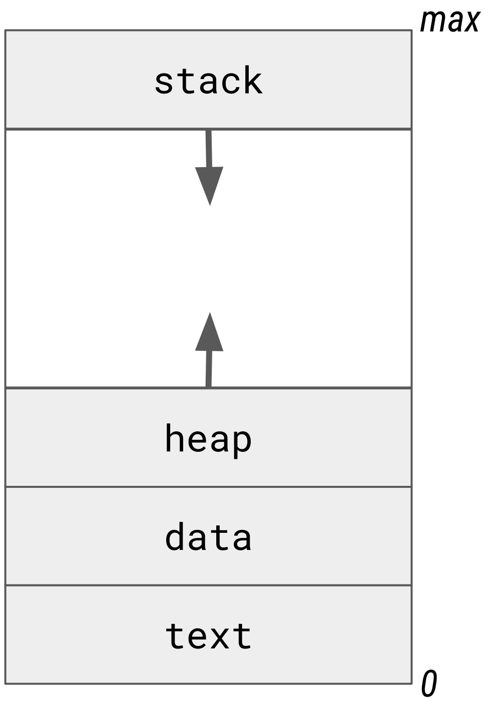
- modern OSes run multiple programs at the same time (multitasking)
- OS needs to keep track and of all running programs
- OS needs to prevent accidental or malicious interference between programs
- OS maintains some information about each running program – we call this a **process**

Program != Process

- a program is a *passive* entity – executable file containing a list of instructions, eg. stored on disk
- a process is an *active* entity – associated with a unique program counter and other resources
- a program *becomes* a process when it is loaded into memory for execution
- a single program can be used to start multiple processes
    - eg. running multiple terminals or shells

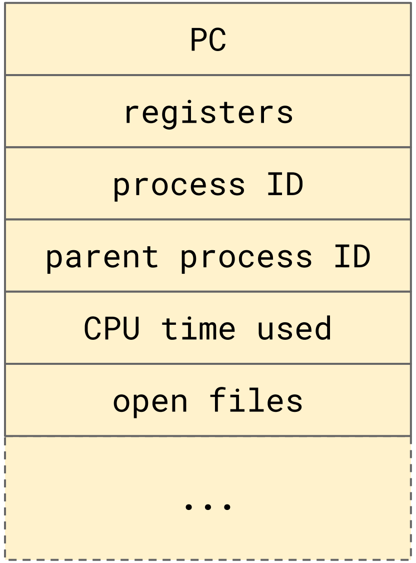
A process in memory

- each process gets its own address space
  - part of memory available to a process, decided by OS
  - on modern OSes it is a *virtual* address space (0 - max), isolated from other processes
- examples of things in address space of a process:
  - **text section**: the program code
  - **data section**: global variables, constant variables
  - **heap**: memory for dynamic allocation during runtime
  - **stack**: temporary data (parameters, return address, local variables)
  - plus many other bits of information needed by the OS for management
- OSes often group all information needed to run a process in a data-structure that we will call **Process Control Block**



Process control block (PCB)

- typical parts of PCB:
  - process state
  - program counter, CPU registers
  - priority, pointers to various queues
  - memory management info: eg. page tables, segment tables, etc.
  - accounting info: eg. CPU time, timeout values, process numbers, etc.
  - I/O status info: open files, I/O devices, etc.
- PCB in practice is not a single data structure
- a **process table** is a collection of all PCBs



More examples of fields of a PCB

Process management

program counter
registers
stack pointer
process state
priority
scheduling parameters
process ID
parent process
process group
signals
process start time
CPU time used
children's CPU time used
time of next alarm
...

Memory management

pointer to text segment
pointer to data segment
pointer to stack segment
...

File management

root directory
working directory
file descriptors
user ID
group ID
...

Look for “`task_struct`” in Linux kernel sources

https://github.com/torvalds/linux/blob/master/include/linux/sched.h

Operations on processes

- processes need to be created and deleted dynamically
- any multitasking OS must provide mechanisms (APIs) for this
  - in UNIX *process creation* is accomplished by using `fork()`
    - parent process – the process that is creating a new process calls `fork()`
    - child process – the newly created process
    - processes in the system form a process tree
    - each process gets PID - a unique process identifier, usually an `unsigned int`
  - *process execution*, eg. `fork()` in Unix
  - *process termination*, eg. `exit()` or `kill()`
    - to let the OS know it can delete the process and free up resources
    - termination can be (typically) only requested by the process itself, its parent, or an unrelated process provided it's owner has the right permissions
  - many other operations exist as well, such as synchronization, communication, ...

A multiprocess program in C

```
$ man fork

pid_t fork(void);

fork()  creates  a new process by duplicating the calling process.  The
new process is referred to as the child process.  The  calling  process
is referred to as the parent process.

The child process and the parent process run in separate memory spaces.
At the time of fork() both memory spaces have the same content.  Memory
writes,  file  mappings (mmap(2)), and unmappings (munmap(2)) performed
by one of the processes do not affect the other.

The child process is an exact duplicate of the  parent  process  except
for the following points:
...
```

A multiprocess program in C

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

int main()
{
    printf("Hello\n");
    /* create & run child process - a duplicate of parent */
    fork();
    /* both parent and child will execute the next line */
    printf("world.\n");
}
```

Possible output:

```
???
```

A multiprocess program in C

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

int main()
{
    printf("Hello\n");
    /* create & run child process - a duplicate of parent */
    fork();
    /* both parent and child will execute the next line */
    printf("world.\n");
}
```

Possible output:

```
Hello
world.
world.
```

Are other outputs possible?

A multiprocess program in C

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

int main()
{
    printf("Hello\n");
    /* create & run child process - a duplicate of parent */
    fork();
    /* both parent and child will execute the next line */
    printf("world.\n");
}
```

https://repl.it/@pfederl/fork-hello-world
https://repl.it/@pfederl/fork-hello-world-one-char-at-a-time

Possible output:

```
Hello
world.
world.
```

Another possible output:

```
Hello
worwold.
rld.
```

assuming `printf()` is unbuffered and outputs 1 character at a time

Are other outputs possible?

A multiprocess program in C

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

int main()
{
    printf("Hello\n");
    /* create & run child process - a duplicate of parent */
    fork();
    /* both parent and child will execute the next line */
    printf("world.\n");
}
```

If `fork()` fails, it returns **-1**. You should always check the return value of a system call.

Possible output:

```
Hello
world.
world.
```

Another possible output:

```
Hello
worwold.
rld.
```

Another possible output:

```
Hello
world.
```

A multiprocess program in C

```
int main()
{
    /* create & run child process - a duplicate of parent
     * and remember the return value */
    pid_t pid = fork();
    /* both parent and child will execute the next line,
     * but will have different value for pid:
     * 0 for child, positive integer for parent, or -1 for error */
    printf("fork returned %d.\n", pid);
}
```

Possible output:

```
fork returned 7.
fork returned 0.
```

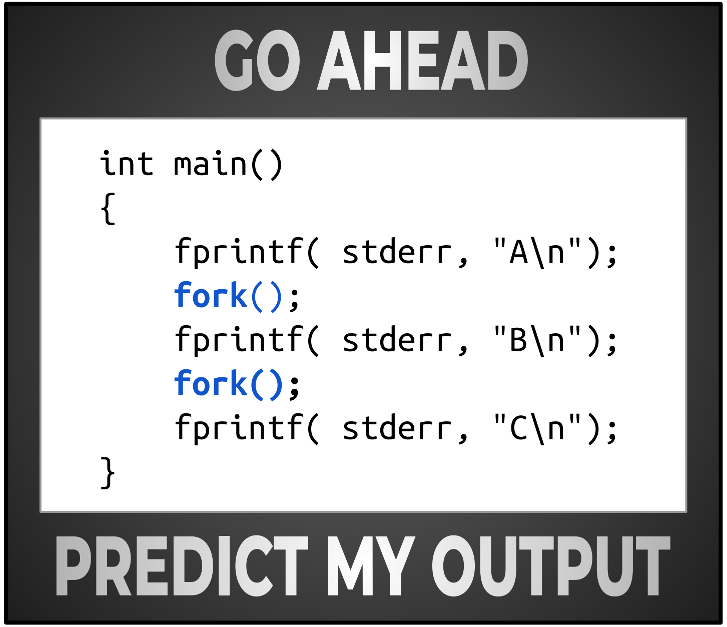
Possible output:

```
fork returned 0.
fork returned 7198.
```

Possible output:

```
fork returned -1.
```

Exercise – to stay in shape



```
int main()
{
    fprintf( stderr, "A\n");
    fork();
    fprintf( stderr, "B\n");
    fork();
    fprintf( stderr, "C\n");
}
```

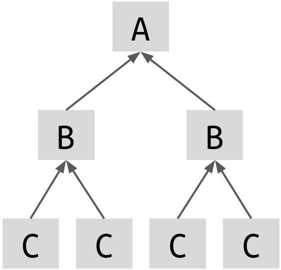
Can you predict all possible outputs assuming fork() does not fail?

Exercise – to stay in shape

```
int main()
{
    fprintf( stderr, "A\n");
    fork();
    fprintf( stderr, "B\n");
    fork();
    fprintf( stderr, "C\n");
}
```

Hint:

- find all topological orderings in the graph



- then remove duplicates
- hard problem in general, but quite easy for a small graph

Another forking exercise – predict all outputs

```
int main()
{
    for(int i=0 ; i<4 ; i++ ) {
        fork();
    }
    printf("X");
}
```

Easy version:
assume fork()
does not fail

Hard version:
assume fork()
could fail

Another forking exercise – predict all outputs

```
int main()
{
    for(int i=0 ; i<4 ; i++ ) {
        fork();
    }
    printf("X");
}
```

Hint: are these equivalent?

```
int main()
{
    fork();
    fork();
    fork();
    fork();
    printf("X");
}
```

Another forking exercise – predict all outputs

```
int main()
{
    for(int i=0 ; i<4 ; i++ ) {
        fork();
        printf("%d",i);
    }
}
```

This is actually very similar to the first exercise...

Exercise – can you predict the output?

```
int x = 10;
int main()
{
    printf("x=%d\n", x);
    fork();
    x ++;
    printf("x=%d\n", x);
}

// assume fork() does not fail
```

Hint:
child has its own 'x' variable, different from the parent

Fork bomb

WARNING

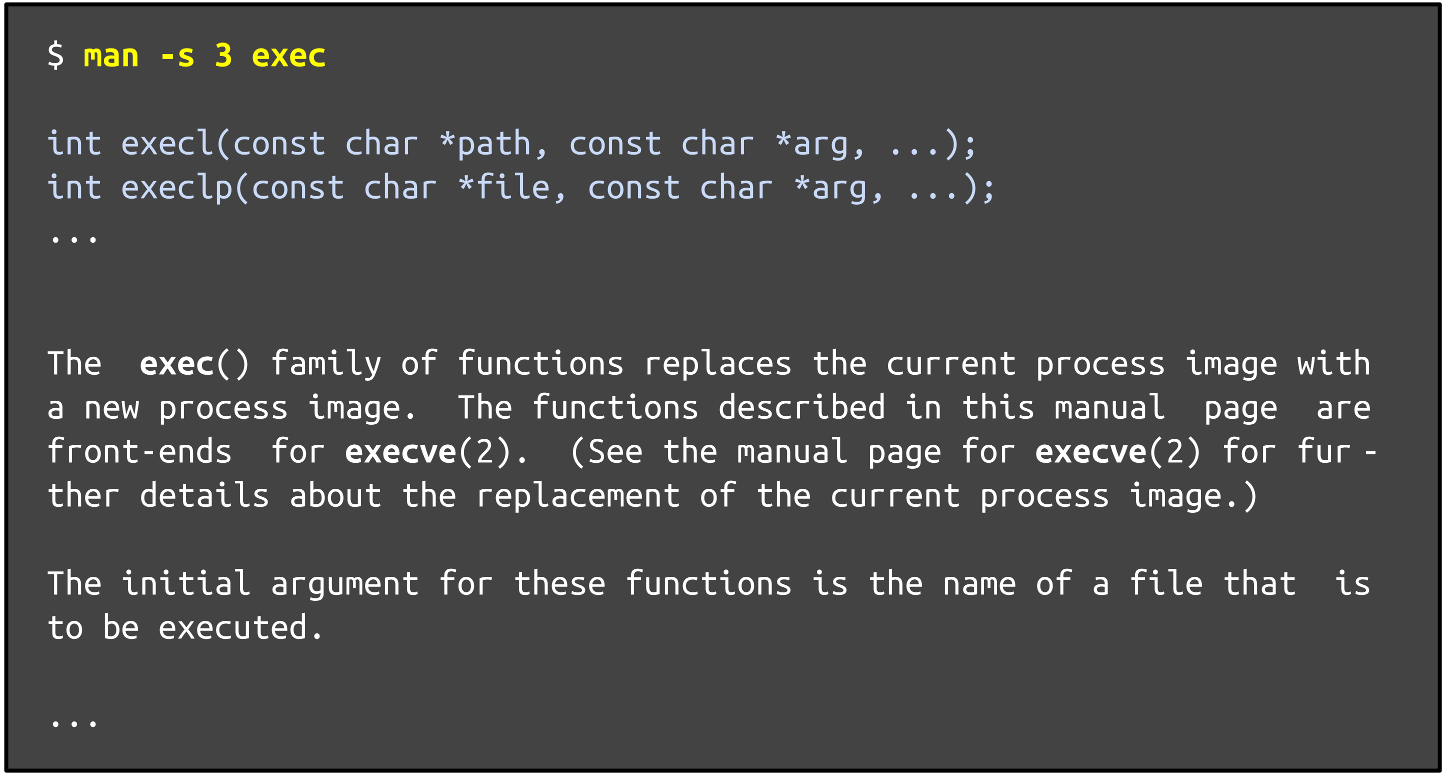
Do not run this on CPSC servers.

```c
int main() {
    while(1) {
        fork();
    }
    printf("X"); // ???
}
```

Starting an external program (programmatically)

- how do we start an external program in Unix?
- no dedicated system call for this purpose
- we have to `fork()` a new process
- then we *replace* child process with an external program using `exec()` system call

A multiprocess program in C



```
$ man -s 3 exec

int execl(const char *path, const char *arg, ...);
int execlp(const char *file, const char *arg, ...);
...


The  exec() family of functions replaces the current process image with
a new process image.  The functions described in this manual  page  are
front-ends  for execve(2).  (See the manual page for execve(2) for fur-
ther details about the replacement of the current process image.)

The initial argument for these functions is the name of a file that  is
to be executed.

...
```

A multiprocess program in C

```
$ man execve

int execve(const char *filename, char *const argv[], char *const envp[]);

execve() executes the program pointed to by filename.  filename must be
either a binary executable, or a script starting with  a  line  of  the
form:
           #! interpreter [optional-arg]

For details of the latter case, see "Interpreter scripts" below.

argv is  an  array  of argument strings passed to the new program.  By
convention, the first of these  strings  should  contain  the  filename
associated  with the file being executed.  envp is an array of strings,
conventionally of the form key=value, which are passed  as  environment
to  the  new  program.  Both argv and envp must be terminated by a null
pointer.
...
```

Starting external program with fork() + exec()

```
int main()
{
    pid_t pid = fork();
    if (pid < 0) { /* check for error */
        fprintf(stderr, "Fork failed");
        exit(-1);
    }
    else if (pid == 0) { /* child process */
        execlp("/bin/ls", "ls", "-l", NULL); /* replace process with 'ls -l' */
        /* we should be checking for errors above… */
        printf("This _should_ never print...\n");
    }
    else { /* parent process will wait for the child to complete */
        printf("Waiting for child process %d\n", pid);
        while (wait(NULL) > 0); // wait(NULL); would work in _this_ case
        printf("Child finished.\n");
        exit(0);
    }
}
```

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

A multiprocess program in C

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

int main()
{
  printf("Before ls.\n");
  system("/bin/ls -l");
  printf("After ls.\n");
}
```

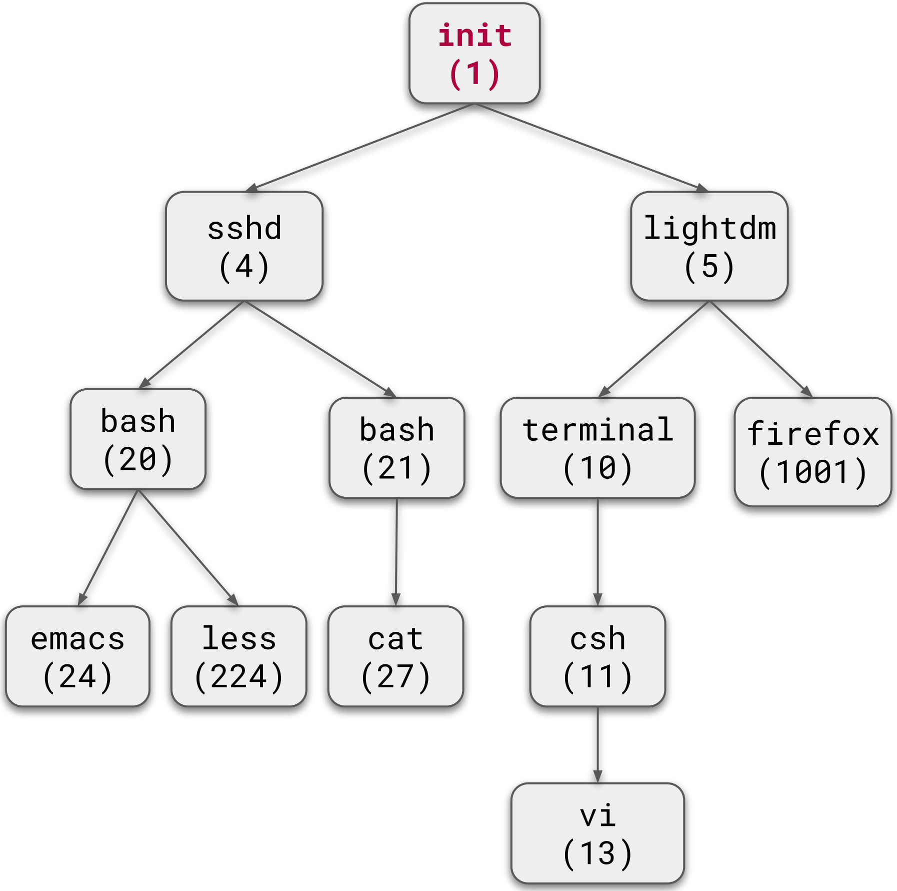
```
$ man system
...
The system() library function uses fork(2) to create a
child process that executes the shell command specified
in command using execl(3) as follows:

    execl("/bin/sh", "sh", "-c", command,
          (char *) 0);

system() returns after the command has been completed.
...
```

A multiprocess program in C

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

int main()
{
  FILE * fp = popen("/bin/ls -l", "r");
  if (fp == NULL) {
    fprintf( stderr, "popen failed.\n");
    exit(-1);
  }

  char buff[4096];
  while (fgets(buff, sizeof(buff), fp) != NULL)
    printf("%s", buff);

  pclose(fp);
}
```

```
$ man popen
...

The  popen()  function opens a process
by creating a pipe, forking, and
invoking the shell.  Since a pipe is
by definition unidirectional, the type
argument may specify only reading or
writing,  not  both;  the resulting
stream is correspondingly read-only or
write-only.
```

Process tree



- **parent process** - the creating process
- **child process** (**child**) - the newly created process
- **PID** - the unique process identifier for each process

- in Unix, parent and child processes continue to be associated, forming a process hierarchy

- in Windows, all processes are equal, the parent process can give the control of its children to any other process

init process

- `init` is the first process started after booting
  - older UNIX systems used init based systems
  - many newer Linux systems switched from init to `systemd`
- `init` is the ancestor of all user processes (direct or indirect parent), i.e. root of process tree
- `init` always has PID = 1
- orphaned processes are adopted by `init` (parent terminates before child)
- printing a process tree

```
$ pstree
$ ps axjf
```

- note: some special 'system processes' are created by kernel during bootstrap, and do not have to be descendants of init, such as `swapper` and `pagedaemon`

Questions?

Review

- Which one of the following executes in kernel mode?
  - □ A user program
  - □ A library function call
  - □ A system call
  - □ A system call wrapper function

- In C, **`printf()`** is a system call.
  - □ True
  - □ False

Review

- When 4 programs are executing on a computer with a single CPU, how many program counters are there?
- When does a program become a process?

Review

- What is the name of the PCB data structure in Linux?
- Name some of fields in a PCB.
- On UNIX systems, what is the name of the process that is the ancestor of all user processes?