CIS 452 - Operating Systems Concepts

Nathan Bowman

Images taken from Silberschatz book

Process Lifecycle

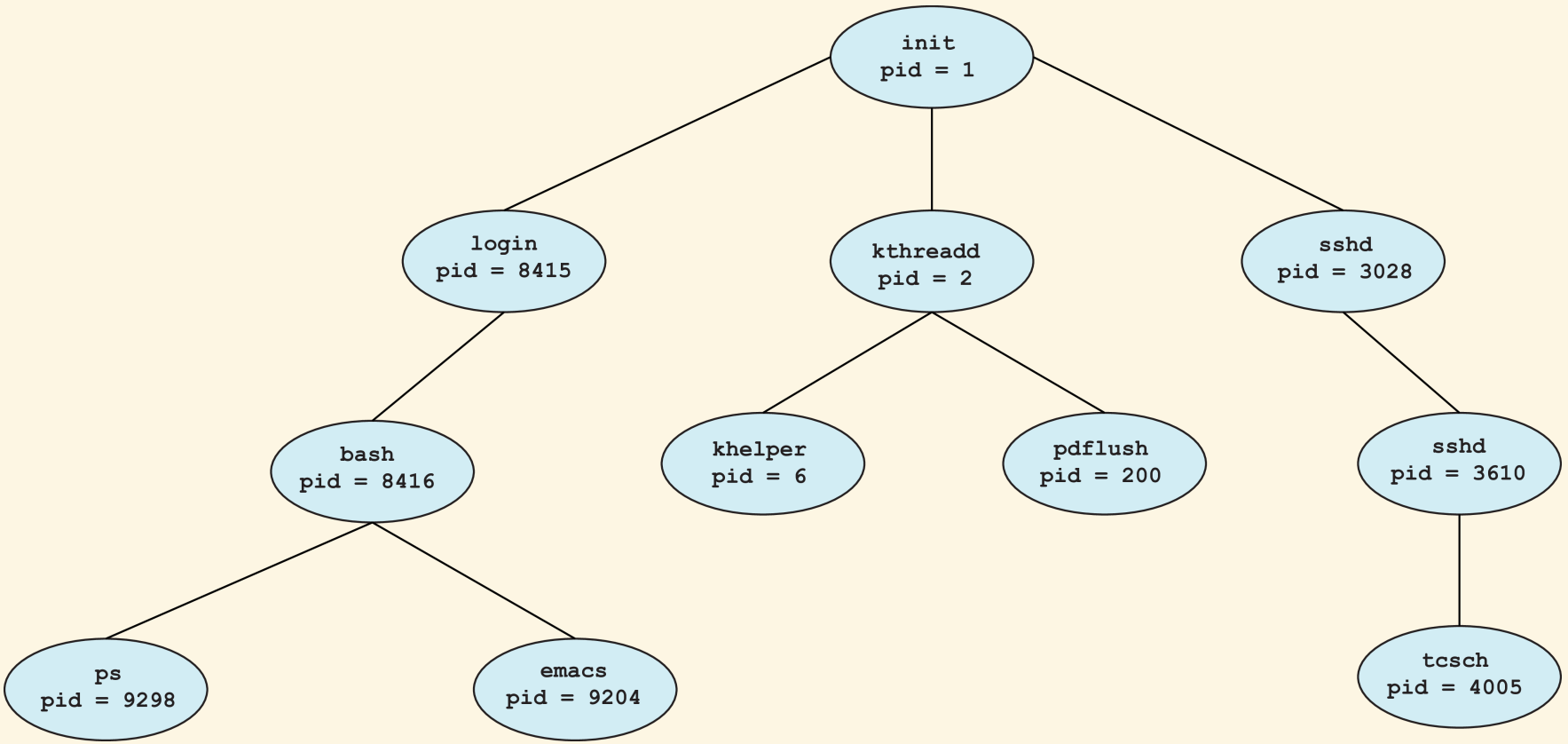
How are processes created and destroyed?

How can we specify what work they should do?

Processes create other processes (through a system call)

Creating process is "parent", created process is "child"

Structure of relationships results in a tree


init
pid = 1
login
pid = 8415
kthreadd
pid = 2
sshd
pid = 3028
bash
pid = 8416
khelper
pid = 6
pdflush
pid = 200
sshd
pid = 3610
ps
pid = 9298
emacs
pid = 9204
tcsch
pid = 4005

OS assigns each process a Process IDentifier (PID)

- unique, nonnegative integer representing process

Goals of spawning new process:

Typically want a new process to execute some command with some parameters we will pass it

Parent might want to stop running until child is finished

Process should clean up after itself when finished

We can accomplish all of this in Linux, but the procedure for executing a given program will feel roundabout at first

In Linux, primary system calls involved in process creation, usage, and removal are

- `fork`
- `exec`
- `wait`
- `exit`

New processes are created via `fork`

OS creates copy of calling process (memory, registers, PC, files, ...pretty much everything)

Creates new PCB, and new process is ready to be scheduled just like any other process on the system

How is having a duplicate helpful?

Main difference between parent and child is return value of `fork`: child sees 0, parent sees pid of new child

May not seem like much, but that difference will be enough to let us do what we want with new process

`exec` system call (library routine, really) overwrites a process's memory with another image

New child that was forked (remember -- it knows it is the child because of the pid) can call `exec` to overwrite its memory

Program code lives in memory, so by overwriting child's memory with, e.g., `ls` program, child process *becomes* a process that performs `ls`

`exec` is an overwrite -- there is no going back

So, method to "create new process that does X" is two system calls: `fork` and `exec`

A disadvantage is that we must call two functions

One advantage is that it allows for simple parent-child communication before the `exec`. Passing arguments is simple when you share all of your memory.

As soon as the child is forked, the two processes can operate mostly independently of one another

Parent has some rights over child, such as ability to end child process via a `syscall`

Parent also responsible for cleaning up after child

Other than that, they operate just like any other two processes

Note about efficiency: it would be a huge waste to actually copy all of the parent's memory to the child's address space, especially if we are about to overwrite the child anyway

Memory "copied" from parent to child is copy-on-write

Okay, we can create a process and assign it a task

What about the other things: waiting and cleaning up?

Parent will by default go on normally, perform other operations, and even spawn additional children

However, it is often the case that child was created for a reason, and parent cannot proceed until child finishes

In that case, parent calls `wait`

`wait` blocks until child is finished and reports exit status of child (for example, lets parent know that child finished with an error)

Processes let the OS (and their parents) know they are done by calling `exit`

Allows OS to free up resources no longer used by process

`exit` accepts a status argument that can be read by parent with `wait`

Once parent is notified of `exit`, it is free of its `wait` and can proceed normally

Caveat: because parent may end up waiting on child's return status, OS cannot clean up after child until parent calls `wait`

Process that has exited but not been cleaned up is a *zombie*

What if parent exits before child?

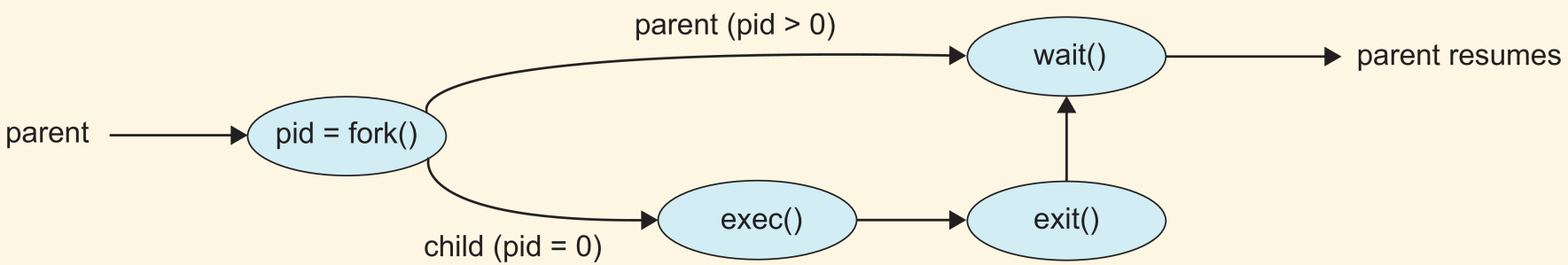
Process whose parent has exited without `wait`ing is an *orphan*

Such a process could, in theory, never be cleaned up, meaning its resources would be tied up forever even if it exited

OS gets around issue of orphans by assigning any orphan to be a child of `init`, the special process with pid 1.

`init` periodically calls `wait` to ensure processes are cleaned up

That's the process lifecyle in Linux!


parent
pid = fork()
parent (pid > 0)
wait()
parent resumes
exec()
exit()
child (pid = 0)

`fork` -- create new process by cloning parent

`exec` -- overwrite process memory with another program

`wait` -- wait for child and allow OS to clean up

`exit` -- terminate process, notify parent, and allow OS to clean up (once corresponding `wait` is called)

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

int main()
{
pid_t pid;

 /* fork a child process */
 pid = fork();

 if (pid < 0) { /* error occurred */
  fprintf(stderr, "Fork Failed\n");
  return 1;
 }
 else if (pid == 0) { /* child process */
  printf("I am the child %d\n",pid);
  execlp("/bin/ls","ls",NULL);
 }
 else { /* parent process */
  /* parent will wait for the child to complete */
  printf("I am the parent %d\n",pid);
  wait(NULL);

  printf("Child Complete\n");
 }

    return 0;
}
```

Last thing: the `fork-exec` way of spawning new processes is a design choice

Other OSs do it differently

Windows has a `CreateProcess` command that accepts arguments specifying what program the new process should run, what arguments it should see, and whether any parent resources should be shared with the child