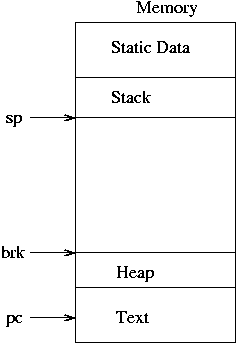
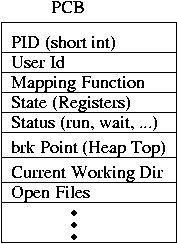
**Processes, Loaders, Fork, and Exec**

**Processes:**

This section is all about processes and the operations we perform on them. What, then, is a process? **It is a single instance of a running program**. There may be many processes running on a machine at any given time; the kernel manages them all so they don't interfere with one another. A single machine may also be running several different processes with the same program (e.g. we are all running emacs(1) on sumatra).

**What makes up a process?**

A process consists of some state. Some of this state is represented as the CPU registers (either squirreled away by the kernel if the process is waiting, or in the CPU itself if it is running). The kernel also maintains a Process Control Block (PCB) with lots of state information (Linux calls this a task\_struct; it's defined in [sched.h](http://lxr.linux.no/source/include/linux/sched.h" \l "L281)). The PCB is not accessible to user programs (question: what mechanism allows the kernel to implement this hiding?). And, of course, there is the memory that the process can use, as defined by the mapping function (the implementation of the mapping function is stored in the PCB). We might draw this all as:

The brk point is just the name UNIX uses for the top of the heap. Also, text is just the program's code in machine language. Note that, while I've included arrows for the SP, PC, and brk on the memory drawing, the SP and PC are actually stored in registers and the brk in the PCB.

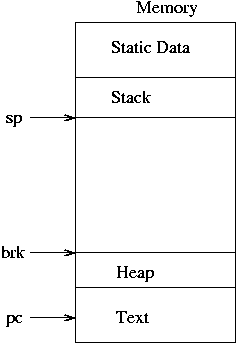
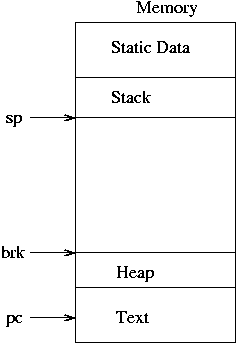
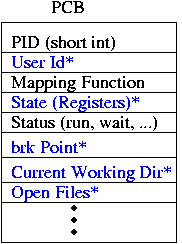
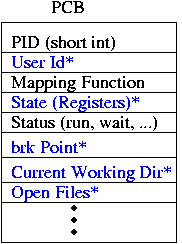
**Running a New Process**

OK, so that's what makes up a process. How do processes get created, though? In general, some other process asks the kernel to create them. For example, I might request that my shell [[1](https://courses.cs.washington.edu/courses/cse451/02sp/section/notes/fork/#shell-process)] create a new mail(1) process by typing the text 'mail\n' at a prompt. (The kernel creates init(8), the first process, at boot. The kernel never spontaneously creates any other processes.)

What happens when I request the shell create a mail(1) process? Under UNIX, two steps are required: first, the shell creates a copy of itself via the fork(2) syscall; then, this copy replaces itself with the mail program using the exec(3) syscall. Kind of strange, but it turns out to be quite useful.

**fork(2):**

The function of fork(2) is to create an almost exact duplicate of the process that calls it. So, if we had the above diagram before the process invoked the fork syscall, then the diagram afterward would look like:

Most fields of the PCB are copied from the original to the newly created PCB; they are starred and shown in blue above (we'll call the new process the child and the old the parent from now on). The memory is copied [[2](https://courses.cs.washington.edu/courses/cse451/02sp/section/notes/fork/#copy-on-write)]; the mapping function is almost the same, but points to the copied memory. Note that the PC of the child process is at exactly the same address as the PC of the parent process. In fact, if we look at the instruction that the PC points to, we'll find that it's a syscall instruction (on x86, the actual assembly instruction is called int).

Recall that it is the kernel's job to schedule lots of different processes, giving each one some amount of processing time. These two processes are no different; eventually, the kernel will get around to running one of them (which one? that's undefined; either could run first). What will happen? The same thing that always happens when a process runs. It's registers are loaded onto the CPU, the kernel changes the CPU bit from protected mode to user mode, and starts processing at the PC.

Readers may have noticed a problem at this point. The processes are both exactly the same. If the kernel runs one, it'll do some stuff (whatever the instructions after the fork syscall instruction tell it to do). Then, when the other one runs, it'll do...the same thing. That's not too useful.

To get around this, we make the processes differ in a very slight but important way. The return value of the fork syscall will be 0 in the child process, and will be greater than zero in the parent (it actually happens to be the PID of the child in the parent). (Where is this return value? It might be in a register, it might be on the stack. That's yet another thing defined by the architecture.)

Thus, if we are clever, we'll put an if (pid > 0) statement right after our fork syscall, and do one thing in the child and a different thing in the parent. In the case of the shell, we'll see that the child calls exec(3) while the parent waits around for the child to finish.

**exec(3):**

So we now have two copies of the shell. We've even figured out how to make them do different things. But they are still both running the shell program; we wanted to run the mail(1) program. So the child uses another syscall, exec(3), to replace itself with the mail program (without creating a new process, hence the need to fork before calling exec lest we no longer have a shell after we are done reading mail).

exec first wipes out the mapping function for the process that called it, getting rid of all the memory. It then goes to the disk, and finds whatever program file was requested.

Executable files are stored on disk in some executable file format; Linux uses ELF, while other operating systems may use Mach.o, COFF, a.out, or other formats. They all have a very similar purpose, though: they tell the loader what to do. Specifically, the header of the file includes instructions for loading various bytes from the file into memory at different addresses, information about how to set up the mapping function, what the initial value of the PC should be, and some other information about what libraries should be loaded. And, of course, the executable file has to include the text (remember, text is code) of the program plus any static data (e.g. string constants) that it uses.

Once the loader has read its instructions from the executable file, it goes about setting everything up, copying the bytes from the file into memory. It probably needs to alter the mapping function to do this. But it doesn't alter most of the other fields in the PCB - this is important, because it means the process calling exec(3) (the child copy of the shell, in this case) can set things up if it wants to, possibly changing the open files or other PCB fields [[3](https://courses.cs.washington.edu/courses/cse451/02sp/section/notes/fork/#through-syscalls)].

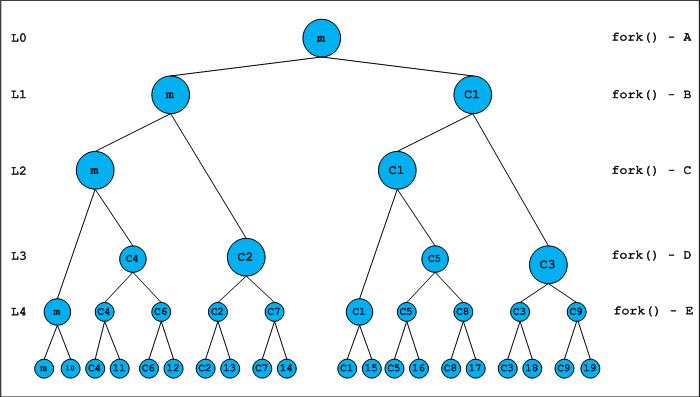
After setting up, the loader sets the initial state of the process (the register values) including the PC and SP. At this point we've still got two processes, but now one (the parent) is the shell, and the other (the child) is mail(1). The kernel can choose to run either of them, doing its normal job of making sure each one gets a bit of processing time. It is likely the case that the shell wants to wait for the mail (child) process to finish before doing anything else; it can tell the kernel this using the wait(2) syscall, which says that the process calling wait(2) doesn't want to run again until one of it's children finishes.

So that's the life cycle of a process on UNIX: clone, mutate, compute, die. Not a bad life for a process...

**fork() and Binary Tree:**

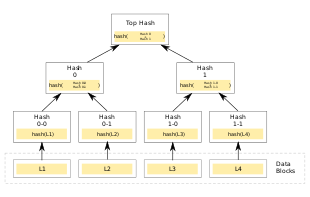
How many processes will be spawned after executing the above program?

A fork() system call spawn processes as leaves of growing binary tree. If we call fork() twice, it will spawn 22 = 4 processes. All these 4 processes forms the leaf children of binary tree. In general if we are level l, and fork() called unconditionally, we will have 2l processes at level (l+1). It is equivalent to number of maximum child nodes in a binary tree at level (l+1).



**Merkle tree:**

In cryptography and computer science, a hash tree or Merkle tree is a tree in which every leaf node is labelled with the cryptographic hash of a data block, and every non-leaf node is labelled with the cryptographic hash of the labels of its child nodes. Hash trees allow efficient and secure verification of the contents of large data structures. Hash trees are a generalization of hash lists and hash chains.

****