Processes
Process creation and states

Paul Krzyzanowski
Rutgers University

January 30, 2012 [original September 20, 2010]

1 Introduction

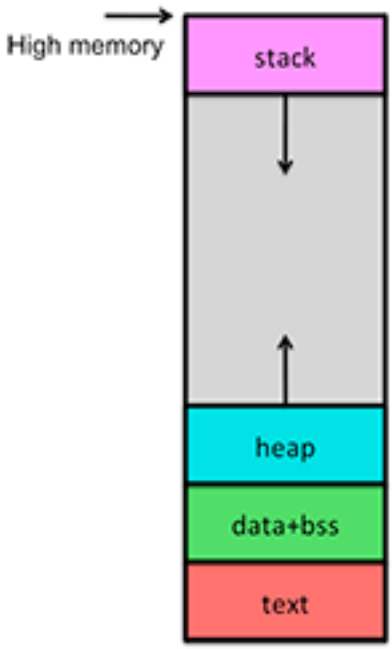

Figure 1: Process memory map

We tend to use the words *program* and *process* interchangeably much of the time. For instance, we might say *"my program died"* or *"my program is generating too much output"*. There is a distinction, however. A **program** refers to the code and static data that is stored in a file (a program can also refer to the raw, uncompiled source code). A **process**, on the other hand, is an executing (or ready to execute) program in the computer. A process is the program *plus* its execution context. The **execution context** includes the state of the processor (**process context**), which is the value of its program counter and all

of its registers. It also includes the process' **memory map**, which is the various regions of memory that have been allocated to the process. The program stored on the disk contains the compiled code (called the **text**) as well as initialized data (for example, initialized global integers, strings, and string constants). It also includes dynamically allocated components. The memory map of a process includes:

- **text**: the machine instructions (the compiled program)
- **data**: initialized static and global data
- **bss**: uninitialized static data (e.g., global uninitialized strings, numbers, structures). The size of this region is contained within the program
- **heap**: dynamically allocated memory (obtained through memory allocation requests, such as *malloc* or *new*)
- **stack**: the call stack, which holds not just return addresses but also local variables, temporary data, and saved registers

2 Process states

Figure 2: Process states

At any point in time, a process can be in one of several states:

1. A process is **running** if it currently has the CPU and is executing code.
2. A process is **ready** if it could run if only it had the CPU.
3. A process is **blocked** when it cannot run because it is waiting for some event to occur (for example, for an I/O operation to complete).

On a single processor machine, only one process may be running at a time, but several processes may be ready to run and several may be blocked. The operating system will maintain a list of ready processes and the **scheduler** is responsible for prioritizing this list based on which processes deserve to run next. The operating system maintains a separate list for blocked processes (unordered because we don't know what event will occur next).

A newly created process is placed at the back of the ready list (we will look at how this list is managed later when we consider process scheduling). Eventually it moves to the head of the list and gets a chance to run. After a while, it may perform a read system call to get some data. At this point, since the data is not ready, it becomes blocked and another process is allowed to run. When the operating system completes the read operation, the process is no longer blocked and is put back on the ready list.

If a running process does not get blocked (for example, it is compute intensive), eventually a hardware timer will interrupt the operating system and cause the operating system to stop (preempt) the process, put it on the ready list, and give someone else a chance to run. This is known as **preemptive multitasking**. Systems that do not do this, such as the old Microsoft Windows 3.1 operating systems, rely on the process to be well-behaved and decide to give up the CPU every once in a while. These systems are **non-preemptive**.

3 Processes in the operating system

The scheduler is responsible for deciding what process gets to run and for saving and restoring the state of a process as it gets stopped and when it gets to run again.

The **context** of a process is its state. As we saw, this is its text (the program code), all global variables and data structures (data and bss), all dynamic memory (heap) that was allocated to the process, the contents of the user and kernel stacks, and all machine registers.

When a process is running, the system is said to be in the context of that process. When the kernel decides to execute another process, it does a **context switch**, causing the system to execute in a different process' context. When doing a context switch, the kernel has to save enough information so that it can switch back to the earlier process and continue executing exactly where it left off. When a process executes a system call and has to change from user to kernel mode, the kernel also has to save enough information so that it can later return to user mode and continue executing the program.

4 Process list & Process Control Blocks

To keep track of processes, the operating system maintains a process table (or list). Each entry in the process table corresponds to a particular process and contains fields with information that the kernel needs to know about the process. This entry is called a **Process Control Block** (**PCB**) Some of these fields on a typical Linux/Unix system PCB are:

- Machine state (registers, program counter, stack pointer)
- Parent process and a list of child processes
- Process state (ready, running, blocked)
- Event descriptor if the process is blocked
- Memory map (where the process is in memory)
- Open file descriptors
- Owner (user identifier). This determines access privileges signaling privileges
- Scheduling parameters

- Signals that have not yet been handled
- Timers for accounting (time resource utilization)
- Process group (multiple processes can belong to a common group)

A process is identified by a unique number called the **process ID** (**PID**). Some operating systems (notably UNIX-derived systems) have a notion of a process group. A process group is just a way to lump related processes together so that related processes can be signaled. Every process is a member of some process group. This group membership is inherited from the parent. If a process wishes to become a member of a new group, it can run the *setpgrp* system call, which returns a new group ID (if it's not already a group leader) and sets the process' group to that ID. When a process is a group leader, its process ID is the same as its process group ID.

5 Process states: more detail

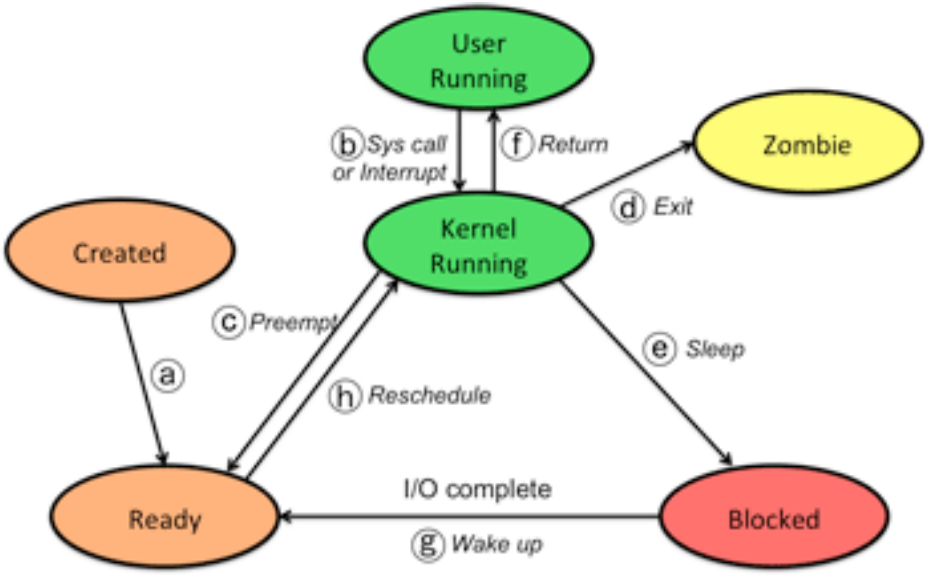

Figure 3: Process states and transitions in more detail

Created (a) Temporary state when a process is created. When the initial memory is allocated, stack is set up, and the PCB is fully initialized, then the process is ready to run and is moved to the *ready* state and placed on the ready list.

Running in user mode Running in user mode is the normal execution of a program: these are the instructions that constitute the program: your compiled code and the various libraries.

(b) Performing a system call invokes a *mode switch*, causing the process' flow of control to switch to the kernel and causing the processor to run in kernel mode. At this point,

the operating system kernel code is executing but the operating system is still aware that the process is currently active and the flow of control is on behalf of the process.

Note that a hardware interrupt, such as that from the system timer, will also cause the state transition to kernel mode. This, however is, is not something that is part of the flow of control of the process. It represents a context switch into an **interrupt context**: a temporary context for servicing interrupts.

Running in kernel mode (c) The kernel may decide that it's time to schedule another process to execute and causes the current process to be preempted. The process is goes to the *ready* state and its process ID is put on the ready list.

(d) A process death (an exit or an abnormal termination) causes it to become a **zombie**. Otherwise, it lingers in this state until the parent process executes a *wait* system call to get the state of this dead process (e.g., exit value). If there is no active parent process, then the zombie is eventually cleaned up by the *init* process (the very first process) and disappears.

(e) If a process needs to wait on an event (e.g. read data, sleep), it will go to the *blocked* state since it is not ready to run at this point.

(f) Once the process is rescheduled to run again, the kernel code restores the user state of the process and returns the process to user-mode running.

Blocked (g) If whatever it was that caused a process to become blocked occurs, the process is moved to the *ready* state and put on the ready list. For example, the process may have requested a *read* of data from the disk and the data has now been transferred to memory.

Ready to run (h) When it is time for another process to run, the next process is taken off the ready list. If it is time for a preempted process to run again, the kernel completes the return from the interrupt that caused it to preempt the process.

6 Programming processes under UNIX/Linux/SunOS/OS X

Please consult the C/UNIX Programming Tutorials for programming examples.

6.1 How do I find the process' ID and group?

A process can find its process ID with the *getpid* system call. It can find its *process group number* with the *getpgrp* system call, and it can find its parent's process ID with *getppid*. For example:

Download this file

7 Creating a process

The *fork* system call clones a process into two processes running the same code. *Fork* returns a value of 0 to the child and a value of the process ID number (pid) to the parent. A value of –1 is returned on failure.

Download this file

Algorithm for fork

- Check for available resources.
- Allocate a slot in the process table/list.
- Assign a unique process ID number.
- Check whether the user is running too many processes.
- Set child state to *created*.
- Copy data from parent process control block to child block.
- Increment counts on current directory.
- Increment counts on open files.
- Copy parent context in memory (this can be a *copy on write*).
- Push a system level context on child, allowing it to recognize itself when it runs.
- Change child state to *ready to run*.

8 Running other programs

fork is fine, but all It gives us is a set of essentially identical clones to the original process. How do we run new programs?

The *execve* system call replaces the current process with a new program. The syntax to *execve* is:

```
execve(char *filename, char **argv, char **envp)
```

This system call returns only if the named program could not run. *argv* is the argument list that is passed to main. The last item on the list is 0 and the first, *argv[0]*, is ignored and expected to contain the command name. *filename* contains the name of the program to load and run. *envp* contains character strings representing the environment in the form *name=value*. *envp* is also stored in the global variable *char **environ* (*getenv* is a library function to search the environment for a particular name).

In addition to the system call *execve*, there are a number of library routines that are front-ends to *execve*. These are described in the *exec* manual page. A couple of useful ones are:

- *execlp*, which allows you to specify all the arguments on the command line (the first parameter is the command and the last must be a null pointer) and uses the search path set by the *PATH* environment variable to find the command. For example,

  execlp("ls", "ls", "-al", "/usr/paul", 0)

- *execvp*, which is almost like *execve* except that the *PATH* environment variable is user as a search path for the command and the default environment is used. Here's an example:

  char **av[] = { "ls", "-al", "/home/paul", 0}; execvp("ls", av);

9 Exiting a process

A process exits by default after it finished executing *main*. However, the *exit* call may be used to force a program to exit and/or to specify a return value on exit (a number in the range of 0..255).

Download this file

After compiling and running this program, run the shell command:

```
echo $?
```

to see the exit code from the program.

10 Algorithm for exit

1. Ignore all signals
2. If the process is associated with a controlling terminal:
   a) Send a hang-up signal to all members of the process group
   b) Reset process group for all members to 0
3. Close all open files.
4. Release current directory.
5. Release current changed root, if any.
6. Free memory associated with the process.
7. Write an accounting record (if accounting).
8. Make the process state zombie.
9. Assign the parent process ID of any children to be 1 (init).
10. Send a death of child signal to parent process.
11. Context switch.

11 Signals

Signals inform processes of the occurrence of asynchronous events. Processes can send signals to other processes owned by the same user (*root*, also known as the *superuser*, can send signals to anyone). The UNIX kernel may also send signals to a process to notify it of certain events.

12 Sending a signal

A process can send a signal to another process with the kill system call. The syntax for this call is:

```
kill(int pid, int signal_number)
```

where *pid* is the process ID and *signal_ number* is the signal that you want to send. Symbolic names for the signals are defined in the include file *signal.h* and may be obtained via the manual page (*man signal*). If the process ID is 0, then the signal is sent to all processes in the same process group. It pid is 1, the signal is sent to all processes with the same user ID.

Here is an example. Let's write a program to fork itself into two processes: the parent will just look forever, printing *"I'm the parent"*. The child will wait three seconds, kill the parent, and exit. The signal that one sends to kill a process is SIGKILL (signal 9), which forces the process to terminate. The program looks like this:

Download this file

13 Detecting signals

One of three things can happen when receiving a signal:

- The process may be able to ignore it (some signals, like KILL, cannot be ignored.
- The process can take whatever the system decided was the default action for the signal, which is usually killing the program.
- The process can ask that a user-defined function be run when the signal is detected.

We can specify what a function does when it gets a signal with the *signal* system call. This call accepts two parameters: the first is the signal number and the second is a pointer to a function that will be run when the signal is detected. In place of a pointer to a function, you can specify SIG_IGN to ignore the signal or SIG_DFL to take the default action. *signal* returns a pointer to the function that was the action that would have been taken if that signal would have been received before you called signal. On some versions of UNIX (SunOS in particular, not Linux or OS X), you have to reset the signal request after a signal has been received (call *signal* again).

Let's use the program from the previous example and modify it so that the parent will catch a signal that is sent by the child. This time we'll send a signal called SIGUSR1, a signal that does not correspond to any event and is reserved for use by user programs.

Download this file

The code is not a lot different. The parent calls *signal* to request that anytime the process gets the signal SIGUSR1, it should run the function *catchme*. The function *catchme* prints a message and exits. If it did not exit, then the program would continue executing where it left off after *catchme* returns, printing "*I'm the parent*" *ad infinitum*.

14 How does the kernel check for signals?

Signals may occur asynchronously and often occur when the process is not running (i.e., it may be blocked or ready). When either the kernel or another user process decides to send a signal to another process, all the kernel does is note the fact and set a bit in the process control block for that process that there is a signal. When the process is brought from the ready state to the running state, the kernel takes a look to see whether the process has to

receive a signal when it starts running. Here are the essentials of this signal detector in the kernel (in pseudocode):

```
while ("received signal" field in PCB is not zero) {
    find a signal number set to the process
    if (signal == death of child)
        if ignoring death of child
            free PCBs of zombies for this parent
        else
            return the signal
    else if not ignoring the signal
        return the signal
    turn off the "received signal" bit in the process control block
}
return "no signal"
```

15 Waiting for a process to die

When a child process dies, it sends a signal to the parent notifying it of its death. The signal is known as SIGCHLD and is defined in the include file *signal.h*. We can catch this signal with the *signal* system call that allows us to assign a function to catch this software interrupt. Once we get this interrupt, we can run the wait system call to suspend the process until we get the death notification. If we fail to do a wait and just exit, then the pending notification would get inherited by the init process. The dead child remains a zombie process until the process waits for it or it gets inherited by process 1, init (the mother of all processes), who waits for all processes.

The wait system call returns the process ID of a dead process. It also can fill an integer with the exit status that can be retrieved with the macro *WEXITSTATUS*.

```
int pid, my_pid, status;
switch (my_pid=fork()) {
case 0:         /* do child stuff */ break;
case -1:        /* do error stuff */ break;
default:        /* wait for child to exit */
    while (pid=wait(&status))
        if (pid==my_pid)
            printf("got exit of %d\n", WEXITSTATUS(status));
            break;
}
```

16 Algorithm for wait

Here's what the kernel does for a process that's sleeping doing a wait:

```
loop forever {
```

```
    if waiting process has a zombie child
        pick any zombie child
        add its CPU usage to the parent
        free child process control block
        return child ID and the exit code of the child
    if process has no children
        return error
    sleep at an interruptible priority
}
```

17 References

- Linux Multitasking[1], Linux KernelAnalysis-HOWTO, Chapter 6.
- Understanding Memory[2], AICT Linux Cluster Architecture, Academic Information Communication Technologies, University of Alberta
- Bach, Maurice J., *The Design of the UNIX Operating System.* Prentice-Hall, 1986.
- McKusick, M.K., Bostic, K., Karels, M.J., Quarterman, J.S., *The Design and Implementation of the 4.4 BSD Operating System.* Addison-Wesley Publishing Co., 1996.
- Leffer, S.J., McKusick, M.K., Karels, M.J., Quarterman, J.S., *The Design and Implementation of the 4.3 BSD UNIX Operating System.* Addison-Wesley Publishing Co., 1989.
- Sun Microsystems, *SunOS 5.8 Programmers Manual*, 2001.
- Tanenbaum, Andrew S., *Modern Operating Systems.* Prentice Hall, 1992.

[1] http://www.linux.org/docs/ldp/howto/KernelAnalysis-HOWTO-6.html
[2] http://www.ualberta.ca/CNS/RESEARCH/LinuxClusters/mem.html