1 Processes

A running instance of a program is called a *process*.

- When you invoke a command from a shell, the corresponding program is executed in a new process; the shell process resumes when that process completes.
- Use multiple cooperating processes in a single application to enable the application to do more than one thing at once, to increase application robustness, and to make use of already-existing programs.
- There are basically two operations to create or alter a process.
    1. you can **fork** to create another process that is an exact copy of the existing process,
    2. you can use **execve** to replace the program running in a process with another program.
- Running another program usually involves both operations, possibly altering the environment in between.
- Newer, lightweight processes (threads) provide separate threads of execution and stacks but shared data segments.
- The Linux specific **clone** call was created to support threads; it allows more flexibility by specifying which attributes are shared.
- The use of shared memory allows additional control over resource sharing between processes.

1.1 Looking at Processes

1.1.1 Process IDs

- Each process in a Linux system is identified by its unique process ID, **pid**.
- Process IDs are 16-bit numbers that are assigned sequentially by Linux as new processes are created.
- Every process also has a parent process, you can think of the processes on a Linux system as arranged in a tree, with the **init** process at its root.The parent process ID, or **ppid**, is simply the process ID of the process's parent.

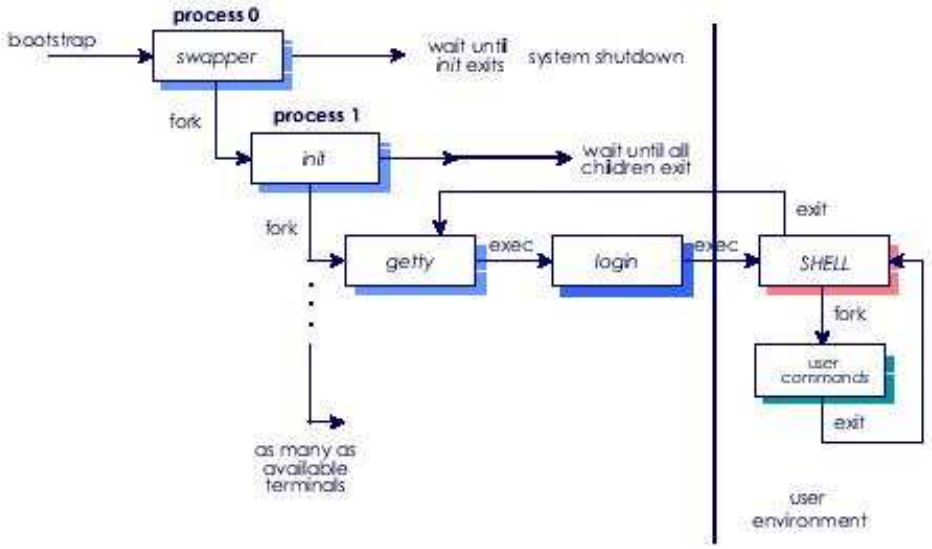

Figure 1: UNIX System initialization

- A program can obtain the process ID of the process it's running in with the *getpid()* system call, and it can obtain the process ID of its parent process with the *getppid()* system call.

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

int main ()
{
  printf ("The process id is %d\n", (int) getpid ());
  printf ("The parent process id is %d\n", (int) getppid ());
  return 0;
}
```

  if you invoke it every time from the same shell, the parent process ID (that is, the process ID of the shell process) is the same.

1.1.2 Viewing Active Processes

- The **ps** command displays the processes that are running on your system.

```
$ ps
```

- This invocation of **ps** shows two processes. The first, **bash**, is the shell running on this terminal. The second is the running instance of the **ps** program itself.

```
$ ps -e -o pid,ppid,command
```

1.1.3 Killing a Process

- You can kill a running process with the **kill** command. Simply specify on the command line the process ID of the process to be killed.
- The **kill** command works by sending the process a **SIGTERM** or termination, signal. This causes the process to terminate, unless the executing program explicitly handles or masks the **SIGTERM** signal.

1.2 Process Control

1.2.1 Attributes

- Table 1 attempts to summarize how process attributes are shared, copied, replaced, or separate for the four major ways to change a process.
- Instead of actually copying memory, a feature known as "copy-on-write" is frequently used in modern OSes like Linux.
- The mappings between virtual and physical memory are duplicated for the new process, but the new mappings are marked as read-only. When the process tries to write to these memory blocks, the exception handler allocates a new block of memory, copies the data to the new block, changes the mapping to point to the new block with write access, and then resumes the execution of the program.
- This feature reduces the overhead of forking a new process.

1.3 Creating Processes

Two common techniques are used for creating a new process.The first is relatively simple but should be used rarely because it is inefficient and has considerably security risks. The second technique is more complex but provides greater flexibility, speed, and security.

Table 1: Process Attribute Inheritance

Attribute	fork	thread	clone	execve
Virtual Memory				
Code Segment	copy	shared	CLONE_VM	replaced
Const Data Segment	don't care	shared	CLONE_VM	replaced
Variable Data Segment	copy	shared	CLONE_VM	replaced
stack	copy	separate	CLONE_VM	replaced
mmap	copy	shared	CLONE_VM	replaced
brk	copy	shared	CLONE_VM	replaced
command line	copy	shared	CLONE_VM	replaced
environment	copy	shared	CLONE_VM	replaced
Files				
chroot, chdir, umask	copy	shared	CLONE_FS	copy
File descriptor table	copy	shared	CLONE_FILES	copy
file locks	separate	separate	CLONE_PID	same
Signals				
Signal Handlers	copy	shared	CLONE_SIGHAND	reset
Pending Signals	separate	separate	separate	reset
Signal masks	separate	separate	separate	reset
Process Id	different	different	CLONE_PID	same
timeslice	separate	shared	CLONE_PID	same

1.3.1 Using system and popen

- The **system** and **popen** functions in the standard C library provides an easy way to execute a command from within a program, much as if the command had been typed into a shell.
- For lazy programmers, the **system** and **popen** functions exist. These functions must not be used in any security sensitive program.
- These functions **fork** and then **exec** the user's login shell that locates the command and parses its arguments.
- They use one of the **wait** family of functions to wait for the child process to terminate.
- The **popen** function is similar to **system**, except it also calls **pipe** and creates a *pipe* to the standard input or from the standard output of the program, but not both.

```
#include <stdlib.h>

int main ()
{
  int return_value;
  return_value = system ("ls -l /");
  return return_value;
}
```

- The **system** function returns the exit status of the shell command. If the shell itself cannot be run, **system** returns 127; if another error occurs, system returns –1.
- Because the system function uses a shell to invoke your command, it's subject to the features, limitations, and security flaws of the system's shell.

1.3.2 Using fork and exec

- The DOS and Windows API contains the **spawn** family of functions.These functions take as an argument the name of a program to run and create a new process instance of that program.
- Linux doesn't contain a single function that does all this in one step. Instead, Linux provides one function, **fork**, that makes a child process that is an exact copy of its parent process.
- Linux provides another set of functions, the **exec** family, that causes a particular process to cease being an instance of one program and to instead become an instance of another program.
- Under Linux, **vfork** is the same as **fork**. Under some operating systems, **vfork** is used when the **fork** will be immediately followed by an **execve**, to eliminate unneeded duplication of resources that will be discarded; in order to do this, the parent is suspended until the child calls **execve**.
- **Calling fork**
  - When a program calls **fork**, a duplicate process, called the *child process*, is created.
  - The parent process continues executing the program from the point that **fork** was called. The child process, too, executes the same program from the same place.

- The **fork** function provides different return values to the parent and child processes
  - one process "goes in" to the **fork** call, and two processes "come out," with different return values.
  - The return value in the **parent** process is the process ID of the child.
  - The return value in the **child** process is zero.

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

int main ()
{
  pid_t child_pid;

  printf ("the main program process id is %d\n", (int) getpid ());

  child_pid = fork ();
  if (child_pid != 0) {
    printf ("this is the parent process, with id %d\n", (int) getpid ());
    printf ("the child's process id is %d\n", (int) child_pid);
  }
  else
    printf ("this is the child process, with id %d\n", (int) getpid ());

  return 0;
}
```

- **Using the exec Family**
  - The **exec** functions replace the program running in a process with another program.
  - When a program calls an **exec** function, that process immediately ceases executing that program and begins executing a new program from the beginning, assuming that the *exec* call doesn't encounter an error.
  - The library functions **execl**, **execlp**, **execle**, **execv**, and **execvp** are simply convenience functions that allow specifying the arguments in a different way, use the current environment instead of a new environment, and/or search the current path for the executable.

* Functions that contain the letter **p** in their names (**execvp** and **execlp**) accept a ***program name*** and search for a program by that name in the current execution path; functions that don't contain the **p** must be given the full path of the program to be executed.
* Functions that contain the letter **v** in their names (**execv**, **execvp**, and **execve**) accept the ***argument list*** for the new program as a NULL-terminated array of pointers to strings.
* Functions that contain the letter **l** (**execl**, **execlp**, and **execle**) accept the ***argument list*** using the C language's *varargs* mechanism.
* Functions that contain the letter **e** in their names (**execve** and **execle**) accept an ***additional argument***, an array of environment variables. The argument should be a NULL-terminated array of pointers to character strings. Each character string should be of the form "VARIABLE=value".

```
#include <unistd.h>
int execve (const char *filename, char *const argv [],
char *const envp[]);
extern char **environ;
int execl( const char *path, const char *arg, ..., NULL);
int execlp( const char *file, const char *arg, ..., NULL);
int execle( const char *path, const char *arg , ..., NULL,
            char * const envp[]);
int execv( const char *path, char *const argv[]);
int execvp( const char *file, char *const argv[]);
```

- Because **exec** replaces the calling program with another one, it never returns unless an error occurs.
- Only rarely will you want to use these routines by themselves. Normally, you will want to execute a **fork** first to **exec** the program in a child process.

- **Using fork and exec Together**
  - Allows the calling program to continue execution in the parent process while the calling program is replaced by the subprogram in the child process.

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

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

/* Spawn a child process running a new program.  PROGRAM is the name
   of the program to run; the path will be searched for this program.
   ARG_LIST is a NULL-terminated list of character strings to be
   passed as the program's argument list.  Returns the process id of
   the spawned process.  */

int spawn (char* program, char** arg_list)
{
  pid_t child_pid;

  /* Duplicate this process.  */
  child_pid = fork ();
  if (child_pid != 0)
    /* This is the parent process.  */
    return child_pid;
  else {
    /* Now execute PROGRAM, searching for it in the path.  */
    execvp (program, arg_list);
    /* The execvp function returns only if an error occurs.  */
    fprintf (stderr, "an error occurred in execvp\n");
    abort ();
  }
}

int main ()
{
  /* The argument list to pass to the "ls" command.  */
  char* arg_list[] = {
    "ls",     /* argv[0], the name of the program.  */
    "-l",
    "/",
    NULL      /* The argument list must end with a NULL.  */
  };

  /* Spawn a child process running the "ls" command.  Ignore the
     returned child process id.  */
  spawn ("ls", arg_list);

  printf ("done with main program\n");
```

```
  return 0;
}
```

1.3.3 Using clone

- The Linux specific function call, **clone**, is an alternative to **fork** that provides more control over which process resources are shared between the parent and child processes.

```
#include <sched.h>
int __clone(int (*fn) (void *arg), void *child_stack, int flags,
            void *arg)
```

- This function exists to facilitate the implementation of **pthreads**. It is generally recommended that you use the portable **pthreads_create()** to create a thread instead, although **clone** provides more flexibility.

  1. The first argument is a pointer to the function to be executed.
  2. The second argument is a pointer to a stack that you have allocated for the child process.
  3. The third argument, flags, is **CLONE_*** flags (shown in Table 1).
  4. The fourth argument, arg, is passed to the child function; its function is entirely up to the user.

- The call returns the process ID of the child process created. In the event of an error, the value -1 will be returned and *errno* will be set.

1.3.4 Process Scheduling

- The following calls manipulate parameters that set the scheduling algorithm and priorities associated with a process.

```
#include <sched.h>
int sched_setscheduler(pid_t pid, int policy,
const struct sched_param *p);
int sched_getscheduler(pid_t pid);
struct sched_param {
...
int sched_priority;
...
};
#include <unistd.h>
```

```
int nice(int inc);
#include <sys/time.h>
#include <sys/resource.h>
int getpriority(int which, int who);
int setpriority(int which, int who, int prio);
#include <sched.h>
int sched_get_priority_max(int policy);
int sched_get_priority_min(int policy);
```

- A process with a higher static priority will always preempt a process with a lower static priority.
- For the traditional scheduling algorithm, processes within static priority 0 will be allocated time based on their dynamic priority (*nice value*).
- The system calls **sched_setscheduler** and **sched_getscheduler** are used to set or get, respectively, the scheduling policy and parameters (set only) associated with a particular process.
- The scheduling policy, **policy**, is one of
  - SCHED_OTHER (the default universal time-sharing scheduler policy used by most processes),
  - SCHED_FIFO,
  - SCHED_RR.
- The latter two specify special policies for time critical applications and will preempt processes using SCHED_OTHER.
- A SCHED_FIFO process can only be preempted by a higher priority process, but a SCHED_RR process will be preempted if necessary to share time with other processes at the same priority.
- Linux schedules the parent and child processes independently; there's no guarantee of which one will run first, or how long it will run before Linux interrupts it and lets the other process (or some other process on the system) run.
- In particular; none, part, or all of the **ls** command may run in the child process before the parent completes.
- You may specify that a process is less important and should be given a lower priority by assigning it a higher *niceness* value.

- By default, every process has a niceness of zero.
- A higher niceness value means that the process is given a lesser execution *priority*; conversely, a process with a lower (that is, negative) niceness gets more execution time.

```
$ nice -n 10 sort input.txt > output.txt
```

- You can use the **renice** command to change the niceness of a running process from the command line.
- Only a process with *root* privilege can run a process with a negative niceness value or reduce the niceness value of a running process.

1.4 Signals

- *Signals* are mechanisms for communicating with and manipulating processes.
- A signal is a special message sent to a process.
- Signals are *asynchronous*; when a process receives a signal, it processes the signal immediately, without finishing the current function or even the current line of code.
- Each signal type is specified by its signal number, but in programs, you usually refer to a signal by its name. (In Linux, these are defined in /usr/include/bits/signum.h)
- When a process receives a signal, it may do one of several things, depending on the signal's disposition.
  - For each signal, there is a *default disposition*, which determines what happens to the process if the program does not specify some other behavior.
  - For most signal types, a program may specify some other behavior- either to ignore the signal or to call a special *signal-handler function* to respond to the signal.
  - If a signal handler is used, the currently executing program is paused, the signal handler is executed, and, when the signal handler returns, the program resumes.
- The system sends signals to processes in response to specific conditions.

  - For instance, SIGBUS (bus error), SIGSEGV (segmentation violation), and SIGFPE (floating point exception) may be sent to a process that attempts to perform an illegal operation.
  - The default disposition for these signals it to terminate the process and produce a core file.

- A process may also send a signal to another process.
  - One common use of this mechanism is to end another process by sending it a SIGTERM or SIGKILL signal.
  - Another common use is to send a command to a running program.Two "userdefined" signals are reserved for this purpose: **SIGUSR1** and **SIGUSR2**.
  - The **SIGHUP** signal is sometimes used for this purpose as well, commonly to wake up an idling program or cause a program to reread its configuration files.

- The **sigaction** function can be used to set a signal disposition.
  - The first parameter is the signal number. The next two parameters are pointers to **sigaction** structures; the first of these contains the desired disposition for that signal number, while the second receives the previous disposition.
  - The most important field in the first or second **sigaction** structure is **sa_handler**. It can take one of three values:
    * **SIG_DFL**, which specifies the default disposition for the signal.
    * **SIG_IGN**, which specifies that the signal should be ignored.
    * A pointer to a signal-handler function.The function should take one parameter, the signal number, and return void.

- A signal handler should perform the minimum work necessary to respond to the signal, and then return control to the main program (or terminate the program).

- The main program then checks periodically whether a signal has occurred and reacts accordingly.

- It is possible for a signal handler to be interrupted by the delivery of another signal.

- Even assigning a value to a global variable can be dangerous because the assignment may actually be carried out in two or more machine instructions, and a second signal may occur between them, leaving the variable in a corrupted state.
- If you use a global variable to flag a signal from a signal-handler function, it should be of the special type **sig_atomic_t**.
- The following program skeleton, for instance, uses a signal-handler function to count the number of times that the program receives SIGUSR1, one of the signals reserved for application use.

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

sig_atomic_t sigusr1_count = 0;

void handler (int signal_number)
{
  ++sigusr1_count;
}

int main ()
{
  struct sigaction sa;
  memset (&sa, 0, sizeof (sa));
  sa.sa_handler = &handler;
  sigaction (SIGUSR1, &sa, NULL);

  /* Do some lengthy stuff here.  */
  /* ...  */

  printf ("SIGUSR1 was raised %d times\n", sigusr1_count);
  return 0;
}
```

1.5 Process Termination

- Normally, a process terminates in one of two ways.

- Either the executing program calls the exit function, or the program's main function returns. The exit code is the argument passed to the **exit** function, or the value returned from **main**. The **exit** function
  - terminates execution of the current program,
  - closes any open file descriptors,
  - and returns the lower eight bits of the value status to the parent process to retrieve using the **wait** family of functions.
  - The parent process will receive a SIGCHLD signal. Any child processes' parent process id will be changed to 1 (init).
  - The **exit()** function will call the system call **_exit()**, which may be called directly to bypass the exit handlers.
- A process may also terminate abnormally, in response to a signal.
  - SIGBUS, SIGSEGV, and SIGFPE signals cause the process to terminate.
  - Other signals are used to terminate a process explicitly.
  - The SIGINT signal is sent to a process when the user attempts to end it by typing Ctrl+C in its terminal.
  - The SIGTERM signal is sent by the **kill** command.
  - The default disposition for both of these is to terminate the process.
  - By calling the abort function, a process sends itself the SIGABRT signal, which terminates the process and produces a core file.
  - The most powerful termination signal is SIGKILL, which ends a process immediately and cannot be blocked or handled by a program.

```
$ kill -KILL pid
$ man kill
```

- To send a signal from a program, use the **kill** function. Include the $< sys/types.h >$ and $< signal.h >$ headers if you use the kill function.
- With most shells, it's possible to obtain the exit code of the most recently executed program using the special $? variable.

```
$ ls /
```

```
$ echo $?
0
$ ls bogusfile
ls: bogusfile: No such file or directory
$ echo $?
1
```

- You should use exit codes only between zero and 127. Exit codes above 128 have a special meaning-when a process is terminated by a signal, its exit code is 128 plus the signal number.

1.5.1 Waiting for Process Termination

In some situations, it is desirable for the parent process to wait until one or more child processes have completed.This can be done with the **wait** family of system calls. These functions allow you to wait for a process to finish executing, and enable the parent process to retrieve information about its child's termination.

1.5.2 The wait System Calls

- The simplest such function is called simply **wait**. It blocks the calling process until one of its child processes exits (or an error occurs).
- It returns a status code via an integer pointer argument, from which you can extract information about how the child process exited.
  - For instance, the WEXITSTATUS macro extracts the child process's exit code.
  - You can use the WIFEXITED macro to determine from a child process's exit status whether that process exited normally (via the **exit** function or returning from **main**) or died from an unhandled signal.

```
int main ()
{
int child_status;
/* The argument list to pass to the "ls" command. */
char* arg_list[] = {
"ls", /* argv[0], the name of the program. */
"-l",
"/",
```

```
NULL /* The argument list must end with a NULL. */
};
/* Spawn a child process running the "ls" command. Ignore the
returned child process ID. */
spawn ("ls", arg_list);
/* Wait for the child process to complete. */
wait (&child_status);
if (WIFEXITED (child_status))
printf ("the child process exited normally, with exit code %d\n",
WEXITSTATUS (child_status));
else
printf ("the child process exited abnormally\n");
return 0;
}
```

- The **waitpid** function can be used to wait for a specific child process to exit instead of any child process.
- The **wait3** function returns CPU usage statistics about the exiting child process, and the **wait4** function allows you to specify additional options about which processes to wait for.

1.5.3 Zombie Processes

- If a child process terminates while its parent is calling a **wait** function, the child process vanishes and its termination status is passed to its parent via the **wait** call.
- What happens when a child process terminates and the parent is not calling **wait**?
- The information about its termination, such as whether it exited normally and, if so, what its exit status is would be lost.
- A **zombie** process is a process that has terminated but has not been cleaned up yet. It is the responsibility of the parent process to clean up its zombie children.The **wait** functions do this.
- Suppose that a program forks a child process, performs some other computations, and then calls **wait**.
  - If the child process has not terminated at that point, the parent process will block in the **wait** call until the child process finishes.

  - If the child process finishes before the parent process calls wait, the child process becomes a zombie.

- When the parent process calls **wait**, the zombie child's termination status is extracted, the child process is deleted, and the **wait** call returns immediately.

- The following program forks a child process, which terminates immediately and then goes to sleep for a minute, without ever cleaning up the child process.

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

int main ()
{
  pid_t child_pid;

  /* Create a child process.  */
  child_pid = fork ();
  if (child_pid > 0) {
    /* This is the parent process.  Sleep for a minute.  */
    sleep (60);
  }
  else {
    /* This is the child process.  Exit immediately.  */
    exit (0);
  }
  return 0;
}
```

- Run it, and while it's still running, list the processes on the system by invoking the following command in another window:

```
$ ps -e -o pid,ppid,stat,cmd
```

- What happens when the main **make-zombie** program ends when the parent process exits, without ever calling wait? Does the zombie process stay around? The **init** process automatically cleans up any zombie child processes that it inherits.

1.5.4 Cleaning Up Children Asynchronously

- An easy way to clean up child processes is by handling SIGCHLD.
- The following program is what it looks like for a program to use a SIGCHLD handler to clean up its child processes.

```
#include <signal.h>
#include <string.h>
#include <sys/types.h>
#include <sys/wait.h>

sig_atomic_t child_exit_status;

void clean_up_child_process (int signal_number)
{
  /* Clean up the child process.  */
  int status;
  wait (&status);
  /* Store its exit status in a global variable.  */
  child_exit_status = status;
}

int main ()
{
  /* Handle SIGCHLD by calling clean_up_child_process.  */
  struct sigaction sigchld_action;
  memset (&sigchld_action, 0, sizeof (sigchld_action));
  sigchld_action.sa_handler = &clean_up_child_process;
  sigaction (SIGCHLD, &sigchld_action, NULL);

  /* Now do things, including forking a child process.  */
  /* ...  */

  return 0;
}
```

- Note how the signal handler stores the child process's exit status in a global variable, from which the main program can access it. Because the variable is assigned in a signal handler, its type is **sig_atomic_t**.