UNIX/Linux and Processes

• In UNIX and Linux, everything is done within the context of a process.
  – A terminal checking to see if someone is trying to log in is a process.
  – Logging is another process.
  – The shell running on a terminal window is a process.
Process IDs

- Every process running on a Linux system has its own process ID number, known as a **pid**.
- Pids are 16-bit numbers that are assigned sequentially.
- Every process (with one exception) has a parent process, whose process id number is known as **ppid**.

printpid.c

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

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

SIEGFRIE@panther:~$ cat printpid.c
SIEGFRIE@panther:~/c$ printpid
The process ID is 12301
The parent process ID is 12226
SIEGFRIE@panther:~/c$ printpid
The process ID is 12302
The parent process ID is 12226
SIEGFRIE@panther:~/c$ printpid
The process ID is 12303
The parent process ID is 12226
SIEGFRIE@panther:~/c$

ps

- The ps command displays the processes that are running on the computer.
- The standard way of running only lists the processes belonging to the user's terminal.
- There are other variations that run on other variants of UNIX.
**ps – An Example**

SIEGFRIE@panther:~/c$ ps

<table>
<thead>
<tr>
<th>PID</th>
<th>TTY</th>
<th>TIME</th>
<th>CMD</th>
</tr>
</thead>
<tbody>
<tr>
<td>12226</td>
<td>pts/7</td>
<td>00:00:00</td>
<td>bash</td>
</tr>
<tr>
<td>12448</td>
<td>pts/7</td>
<td>00:00:00</td>
<td>ps</td>
</tr>
</tbody>
</table>

SIEGFRIE@panther:~/c$

**ps – More Example**

SIEGFRIE@panther:~/c$ ps -f

<table>
<thead>
<tr>
<th>UID</th>
<th>PID</th>
<th>PPID</th>
<th>C</th>
<th>STIME</th>
<th>TTY</th>
<th>TIME</th>
<th>CMD</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIEGFRIE</td>
<td>12226</td>
<td>12225</td>
<td>0</td>
<td>22:00</td>
<td>pts/7</td>
<td>00:00:00</td>
<td>-bash</td>
</tr>
<tr>
<td>SIEGFRIE</td>
<td>12457</td>
<td>12226</td>
<td>0</td>
<td>22:19</td>
<td>pts/7</td>
<td>00:00:00</td>
<td>ps -f</td>
</tr>
</tbody>
</table>
| SIEGFRIE@panther:~/c$ ps -l

<table>
<thead>
<tr>
<th>F S UID PID PPID C PRI NI ADDR SZ WCHAN TTY TIME CMD</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 S 16131 12226 12225 0 80 0 - 3713 wait pts/7 00:00:00 bash</td>
</tr>
<tr>
<td>0 R 16131 12458 12226 0 80 0 - 2112 - pts/7 00:00:00 ps</td>
</tr>
</tbody>
</table>

SIEGFRIE@panther:~/c$ ps -j

<table>
<thead>
<tr>
<th>PID PGID SID TTY TIME CMD</th>
</tr>
</thead>
<tbody>
<tr>
<td>12226 12226 12226 pts/7 00:00:00 bash</td>
</tr>
<tr>
<td>12461 12461 12226 pts/7 00:00:00 ps</td>
</tr>
</tbody>
</table>

SIEGFRIE@panther:~/c$
**ps – Another Example**

SIEGFRIE@panther:~/c$ ps -eo pid -eo ppid

<table>
<thead>
<tr>
<th>PID</th>
<th>PPID</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>30238</td>
<td>30228</td>
</tr>
<tr>
<td>30239</td>
<td>30238</td>
</tr>
<tr>
<td>30771</td>
<td>1</td>
</tr>
<tr>
<td>31095</td>
<td>1</td>
</tr>
<tr>
<td>31174</td>
<td>1</td>
</tr>
<tr>
<td>31178</td>
<td>1</td>
</tr>
</tbody>
</table>

SIEGFRIE@panther:~/c$

---

**ps – Another Example**

SIEGFRIE@panther:~/c$ system

total 268
-rw------- 1 root root 126976 Aug 15 09:50 aquota.user

```
drwxr-xr-x 2 root root 4096 Nov 6 09:46 bin
drwxr-xr-x 3 root root 4096 Nov 1 15:35 boot
drwxr-xr-x 16 root root 4260 Oct 1 09:02 dev
drwxr-xr-x 145 root root 12288 Nov 7 15:39 etc
```

```
drwxr-xr-x 2 root root 4096 Nov 1 15:33 lib64
drwx------ 2 root root 16384 May 21 2011
lost+found
drwxr-xr-x 3 root root 4096 May 21 2011 media
drwxr-xr-x 2 root root 4096
```
fork and exec

- Linux uses two different processes to create a state.
- **fork()** makes an exact copy of the process.
  - Both parent and child processes are given the pid for their child.
  - The child process, having no children yet has a child pid of 0.

Killing a Process

- A process can be killed by issue the correct signal.
- The kill command sends a process a **SIGTERM** signal
Creating Processes

- There are two methods for creating a new process:
  - Using the `system` call – inefficient and has security risks
  - Using fork and exec – more complex but provides better flexibility

**system**

- system provides an easy way to execute a command from within a program, as if the command were typed within its shell.
- system creates a subprocess that runs the Bourne shell and then hands the commands to the subprocess to run it.
- If the shell cannot be run, it returns 127. If its any other abnormal termination, it returns 0.
fork()

- The `fork()` system call is used to create a copy of the process that issues the call.
- `fork()` returns the process identification number (`pid`) of the child process; the child process gets 0 from `fork()` while the parent gets the child's `pid`.

fork.c

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

int main(void)
{
    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);

Running fork.c

SIEGFRIE@panther:~/c$ gcc -o ../bin/myfork fork.c
SIEGFRIE@panther:~/c$ myfork
The main program process ID is 5169
This is the parent process, with id 5169
The child's process ID is 5170
This is the child process, with id 5170
SIEGFRIE@panther:~/c$
**execv()**

- `execv()` is one of a family of system calls that replaces the image of the process running with that of another process that is specified.
- The form of the system call is
  
  ```
  int execv(const char *path, 
            char *const argv[]);
  ```
  
  where `path` is the path of the process to replace the current one and `argv` are the command line parameters, with a NULL at the end.

**runhello.c**

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

int  main(void) {
    char *temp[] = {NULL,"hello","world",NULL};

    temp[0]="hello";
    execv("hello",temp);
    printf("error");
}
```
Running `runhello`

```
SIEGFRIE@panther:~/c$ runhello
Filename: hello
hello world
SIEGFRIE@panther:~/c$
```

**fork() and execv()**

- If you wish to run a subprogram within another program:
  1. fork the process and then
  2. exec the subprogram.
- This 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<unistd.h>

int     main(void)
{
    pid_t child_pid;
    char *temp[] = {NULL,"hello","world", NULL};

    /* Duplicate this process. */
    child_pid = fork();

    if (child_pid != 0)     {
        /* This is the parent process. */
        printf("This is the parent\n");
        exit(0);
    }
    else { 
        temp[0]="hello";
        execv("hello",temp);
        printf("error");
    }
}
SIEGFRIE@panther:~/c$ gcc -o ../bin/copyhello copyhello.c
SIEGFRIE@panther:~/c$ copyhello
This is the parent
SIEGFRIE@panther:~/c$ Filename: hello
hello world
SIEGFRIE@panther:~/c$

Running copyhello

SIEGFRIE@panther:~/c$ copyhello
This is the parent
SIEGFRIE@panther:~/c$ Filename: hello
hello world
←
SIEGFRIE@panther:~/c$
Using wait()

- It is often desirable for the parent process to wait until one or more child processes have completed before continuing execution.
- This can be done with the wait family of system calls; they allow you to wait for a process to finish executing, and enable the parent process to retrieve information about its child’s termination.

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

int global; /* In BSS segment, will automatically be assigned '0'*/

int main()
{
}
pid_t child_pid;
int status;
int local = 0;
/* now create new process */
child_pid = fork();

if (child_pid >= 0) {
    /* fork succeeded */
    if (child_pid == 0) {
        /* fork() returns 0 for
           the child process */
        printf("child process!\n");
        /* Increment the local and
           global variables */
        local++;
        global++;
        printf("child PID = %d, parent \
               "pid = %d\n", getpid(), getppid());
        printf("\n child's local = %d, "
               "child's global = %d\n",
               local, global);
        char *cmd[] = {"whoami", (char*)0};
        /* call whoami command */
        return execv("/usr/bin/", cmd);
    }
}
else /* parent process */ {
    printf("parent process!\n");
    printf("parent PID = %d, child \\
           "pid = %d\n", getpid(),
           child_pid);
    /* wait for child to exit, and 
       store child's exit status */
    wait(&status);
    printf("Child exit code: %d\n",
           WEXITSTATUS(status));
    /*
     * The change in local and global
     * variable in child process should
     * not reflect here in parent process.
     */

    printf("\n Parent'z local = %d," \\
            " parent's global = %d\n", 
            local,global);

    printf("Parent says bye!\n");
    exit(0); /* parent exits */
}
else /* failure */
{
    perror("fork");
    exit(0);
}
}
Running few

SIEGFRIE@panther:~/c$ few
parent process!
child process!
parent PID = 5923, child pid = 5924
child PID = 5924, parent pid = 5923

child's local = 1, child's global = 1
Child exit code: 255

Parent'z local = 0, parent's global = 0
Parent says bye!
SIEGFRIE@panther:~/c$

Process Scheduling

• Linux schedules parent and child processes independent of each other, so there is no guarantee of which process will run first or how long it will run before it is pre-empted for the other process (or some other process).
• The user can specify which process is less important by assigning it a niceness value, that can be changed subsequently if the user desires.
nice – An Example

- You can run
  `sort input.txt > output.txt`
  By writing
  `nice -n 10 input.txt > output.txt`
  you reduce its priority.
- You can use `renice` to change its nice factor.
- Only root can run a process with a negative nice factor or change it for a process that is running.

What are signals?

- Signals provide a mechanism for communicating with and manipulating processes in Linux.
- A signal is an asynchronous message sent to a process; when a process receives a signal, it processes it immediately, without finishing the current code that its about to execution.
Types of Signals

- **SIGKILL** This signal ends a process immediately and cannot be handled.

Examples of Signals

<table>
<thead>
<tr>
<th>Type of Signal</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIGHUP</td>
<td>Linux sends a process this signal when it becomes disconnected from a terminal.</td>
</tr>
<tr>
<td>SIGINT</td>
<td>Linux sends a process this signal when the user tries to end it by pressing Ctrl+C.</td>
</tr>
<tr>
<td>SIGILL</td>
<td>A process gets this signal when it attempts to execute an illegal instruction.</td>
</tr>
<tr>
<td>SIGABRT</td>
<td>The abort function causes the process to receive this signal.</td>
</tr>
<tr>
<td>SIGFPE</td>
<td>The process has executed an invalid floating-point math instruction.</td>
</tr>
<tr>
<td>SIGKILL</td>
<td>This signal ends a process immediately and cannot be handled.</td>
</tr>
</tbody>
</table>
What Do Processes Do With Signals?

• When a process receives a signal, it may do one of several things, depending on the signal’s disposition.
• Each signal has a default disposition, which determines what happens to the process if the program does not specify some behavior.

What Do Processes Do With Signals?
(continued)

• 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.
Processes Signaling Other Processes

• A process may also send a signal to another process. This mechanism is used to end another process by sending it a **SIGTERM** or **SIGKILL** signal.

Processes Signaling Other Processes (continued)

• Processes will also send a command to a running program; **SIGUSR1** and **SIGUSR2** are user-defined signals reserved for this purpose.
• 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.
**sigaction()**

- The `sigaction()` system call is used to change the action taken by a process on receipt of a specific signal.
- Usage:
  ```c
  #include <signal.h>
  int sigaction(int signum,
  const struct sigaction *act,
  struct sigaction *oldact);
  ```
  - `signum` specifies the signal and can be any valid signal except `SIGKILL` and `SIGSTOP`.
  - `act` is the new action to be performed.
  - `oldact` is the old action that had been performed.

**memset()**

- `void *memset(void *str, int c, size_t n)` copies the character `c` (an unsigned char) to the first `n` characters of the string pointed to by the argument `str`. 
**sa_handler()**

- **sa_handler** specifies the action to be associated with **signum**.
- The action may be **SIG_DFL** for the default action, **SIG_IGN** to ignore this signal, or a pointer to a signal handling function.
- This function receives the signal number as its only argument.

**siguser.c**

```c
#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 (void)
{
    int i;
    struct sigaction sa;
    memset (&sa, 0, sizeof (sa));
    sa.sa_handler = &handler;
    sigaction (SIGINT, &sa, NULL);

    for (i = 0; i < 10; i++) {
        sleep(1);
        printf("Hi there!\n");
    }
    printf ("SIGILL was raised %d times\n", sigusr1_count);
    return 0;
}
Killing a Process

- The kill command is used to pass a signal to a process.
- The format of the command:
  ```
  kill  -n pid
  ```
  where n is the signal and pid is the process id number.
- Example
  ```
  kill  -9 20401
  ```