Linux System Overview: From Boot To Panic

Boot process

Under BIOS-based systems:

1. Power-on self-test (POST) and peripheral initializations
2. Jump to the boot code in the first 440 bytes of Master Boot Record (MBR)
3. MBR boot code locates and launches boot loader – ex) GRUB, Syslinux
4. Boot loader reads its configuration and possibly presents menu
5. Boot loader loads the kernel and launches it
6. Kernel unpacks the `initramfs` (initial RAM filesystem)
    - initial temporary root filesystem
    - contains device drivers needed to mount real root filesystem
7. Kernel switches to real root filesystem
8. Kernel launches the first user-level process `/sbin/init` (pid == 1)
    - `/sbin/init` is the mother of all processes
    - traditional SysV-style init reads `/etc/inittab`
    - in Arch, `/sbin/init` is a symlink to `systemd`

User session

1. `init` calls `getty`
    - `getty` is called for each virtual terminal (normally 6 of them)
    - `getty` checks user name and password against `/etc/passwd`
2. `getty` calls `login`, which then runs the user’s shell – ex) `bash`
    - `login` sets the user’s environment variables
    - the user’s shell is specified in `/etc/passwd`

Process control

Displaying process hierarchy

```
ps axf     # display process tree

ps axfj    # with more info

ps axfjww  # even if lines wrap around terminal
```

Creating processes

Simple shell program from APUE3, 1.6:

```
#include "apue.h"
#include <sys/wait.h>

int
main(void)
{
        char    buf[MAXLINE];   /* from apue.h */
        pid_t   pid;
        int     status;

        printf("%% ");  /* print prompt (printf requires %% to print %) */
        while (fgets(buf, MAXLINE, stdin) != NULL) {
                if (buf[strlen(buf) - 1] == '\n')
                        buf[strlen(buf) - 1] = 0; /* replace newline with null */

                if ((pid = fork()) < 0) {
                        err_sys("fork error");
                } else if (pid == 0) {      /* child */
                        execlp(buf, buf, (char *)0);
                        err_ret("couldn't execute: %s", buf);
                        exit(127);
                }

                /* parent */
                if ((pid = waitpid(pid, &status, 0)) < 0)
                        err_sys("waitpid error");
                printf("%% ");
        }
        exit(0);
}
```

Questions:

- What happens if the parent process terminates before its children?
    - create this condition and see it in `ps` output
- What happens if a child process has terminated, but the parent never calls `waitpid()`?
    - create this condition and see it in `ps` output

Signals

How do you terminate the following program?

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

static void sig_int(int signo)
{
    printf("stop pressing ctrl-c!\n");
}

int main()
{
    if (signal(SIGINT, &sig_int) == SIG_ERR) {
        perror("signal() failed");
        exit(1);
    }

    int i = 0;
    for (;;) {
        printf("%d\n", i++);
        sleep(1);
    }
}
```

How about this one?

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

static void sig_int(int signo)
{
    printf("stop pressing ctrl-c!\n");
}

static void sig_term(int signo)
{
    printf("stop trying to kill me!\n");
}

int main()
{
    if (signal(SIGINT, &sig_int) == SIG_ERR) {
        perror("signal() failed");
        exit(1);
    }

    if (signal(SIGTERM, &sig_term) == SIG_ERR) {
        perror("signal() failed");
        exit(1);
    }

    int i = 0;
    for (;;) {
        printf("%d\n", i++);
        sleep(1);
    }
}
```

Signals and system calls

The following program, from APUE3 section 1.9, adds signal handling to the simple shell program we looked at before:

```
#include "apue.h"
#include <sys/wait.h>

static void sig_int(int);       /* our signal-catching function */

int
main(void)
{
        char    buf[MAXLINE];   /* from apue.h */
        pid_t   pid;
        int     status;

        if (signal(SIGINT, sig_int) == SIG_ERR)
                err_sys("signal error");

        printf("%% ");  /* print prompt (printf requires %% to print %) */
        while (fgets(buf, MAXLINE, stdin) != NULL) {
                if (buf[strlen(buf) - 1] == '\n')
                        buf[strlen(buf) - 1] = 0; /* replace newline with null */

                if ((pid = fork()) < 0) {
                        err_sys("fork error");
                } else if (pid == 0) {      /* child */
                        execlp(buf, buf, (char *)0);
                        err_ret("couldn't execute: %s", buf);
                        exit(127);
                }

                /* parent */
                if ((pid = waitpid(pid, &status, 0)) < 0)
                        err_sys("waitpid error");
                printf("%% ");
        }
        exit(0);
}

void
sig_int(int signo)
{
        printf("interrupt\n%% ");
}
```

Does it work as expected in Linux? What about other UNIX platforms like Mac OS X? What’s going on here?

X session

Let’s take a look at the processes in your X session:

```
  145 ?        Ss     0:00 login -- jae
  406 tty1     Ss     0:00  \_ -bash
  421 tty1     S+     0:00      \_ xinit /etc/xdg/xfce4/xinitrc -- /etc/X11/xinit/xs
  422 ?        Ss     0:03          \_ /usr/bin/X -nolisten tcp :0 vt1
  425 tty1     S      0:00          \_ sh /etc/xdg/xfce4/xinitrc
  430 tty1     Sl     0:00              \_ xfce4-session
  448 tty1     S      0:00                  \_ xfwm4 --display :0.0 --sm-client-id 2
  450 tty1     Sl     0:00                  \_ Thunar --sm-client-id 2ed5a75a5-2f67-
  452 tty1     Sl     0:01                  \_ xfce4-panel --display :0.0 --sm-clien
  494 tty1     S      0:00                  |   \_ /usr/lib/xfce4/panel/wrapper /usr
  498 tty1     S      0:00                  |   \_ /usr/lib/xfce4/panel/wrapper /usr
  454 tty1     Sl     0:00                  \_ xfdesktop --display :0.0 --sm-client-
  456 tty1     Sl     0:01                  \_ xfce4-terminal --geometry=100x36 --di
  505 tty1     S      0:00                      \_ gnome-pty-helper
  507 pts/0    Ss+    0:00                      \_ bash
  518 pts/1    Ss     0:00                      \_ bash
 2330 pts/1    R+     0:00                      |   \_ ps afx
 1218 pts/2    Ss+    0:00                      \_ bash
```

Explorations:

- Identify the function of each process
- Understand the network client-server architecture of X window system
- Try running a simpler X session – `twm` or even just `xterm`

Kernel module

Here is a skeleton kernel module from OSCE2, Chapter 2, Programming Project:

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>

/* This function is called when the module is loaded. */
int simple_init(void)
{
       printk(KERN_INFO "Loading Module\n");

       return 0;
}

/* This function is called when the module is removed. */
void simple_exit(void) {
        printk(KERN_INFO "Removing Module\n");
}

/* Macros for registering module entry and exit points. */
module_init( simple_init );
module_exit( simple_exit );

MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("Simple Module");
MODULE_AUTHOR("SGG");
```

Here is the Makefile:

```
obj-m += simple.o
all:
        make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules
clean:
        make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

Can you modify the code to cause the kernel panic?

References

- Source code for this lecture
- Arch Boot Process

Last updated: 2014–01–27