Lecture 06: Process Groups, Sessions, Signals

October 11, 2016
Login Process

- init(8)
  - reads /etc/ttys
Login Process

- **init(8)**
  - reads `/etc/ttys`

- **getty(8)**
  - opens terminal
  - prints “login: ”
  - reads username
Login Process

- **init(8)**
  - reads `/etc/ttys`

- **getty(8)**
  - opens terminal
  - prints “login: ”
  - reads username

- **login(1)**
  - `getpass(3)`, encrypt, compare to `getpwnam(3)`
Login Process

- init(8)
  - reads /etc/ttys

- getty(8)
  - opens terminal
  - prints “login: ”
  - reads username

- login(1)
  - getpass(3), encrypt, compare to getpwnam(3)
  - register login in system databases
Login Process

- init(8)
  - reads /etc/ttys

- getty(8)
  - opens terminal
  - prints “login: ”
  - reads username

- login(1)
  - getpass(3), encrypt, compare to getpwnam(3)
  - register login in system databases
  - read/display various files
Login Process

- init(8)
  - reads /etc/ttys

- getty(8)
  - opens terminal
  - prints “login: ”
  - reads username

- login(1)
  - getpass(3), encrypt, compare to getpwnam(3)
  - register login in system databases
  - read/display various files
  - initgroups(3)/setgid(2), initialize environment
Login Process

- init(8)
  - reads /etc/ttys

- getty(8)
  - opens terminal
  - prints “login: ”
  - reads username

- login(1)
  - getpass(3), encrypt, compare to getpwnam(3)
  - register login in system databases
  - read/display various files
  - initgroups(3)/setgid(2), initialize environment
  - chown(2) terminal device
Login Process

- init(8)
  - reads /etc/ttys

- getty(8)
  - opens terminal
  - prints “login: ”
  - reads username

- login(1)
  - getpass(3), encrypt, compare to getpwnam(3)
  - register login in system databases
  - read/display various files
  - initgroups(3)/setgid(2), initialize environment
  - chown(2) terminal device
  - chdir(2) to new home directory
Login Process

- **init(8)**
  - reads `/etc/ttys`

- **getty(8)**
  - opens terminal
  - prints “login: ”
  - reads username

- **login(1)**
  - `getpass(3)`, encrypt, compare to `getpwnam(3)`
  - register login in system databases
  - read/display various files
  - `initgroups(3)/setgid(2)`, initialize environment
  - `chdir(2)` to new home directory
  - `chown(2)` terminal device
  - `setuid(2)` to user’s uid, `exec(3)` shell
Login Process

Let’s revisit the process relationships for a login:

\[ \text{kernel} \Rightarrow \text{init(8)} \quad \# \text{explicit creation} \]
Login Process

Let’s revisit the process relationships for a login:

\[
\text{kernel} \Rightarrow \text{init}(8) \quad \# \text{explicit creation}
\]
\[
\text{init}(8) \Rightarrow \text{getty}(8) \quad \# \text{fork}(2)
\]
Login Process

Let’s revisit the process relationships for a login:

```
kernel  ⇒  init(8)  # explicit creation

init(8)  ⇒  getty(8)  # fork(2)

getty(8)  ⇒  login(1)  # exec(3)
```
Login Process

Let’s revisit the process relationships for a login:

- `kernel` \(\Rightarrow\) `init(8)`  # explicit creation
- `init(8)` \(\Rightarrow\) `getty(8)`  # fork(2)
- `getty(8)` \(\Rightarrow\) `login(1)`  # exec(3)
- `login(1)` \(\Rightarrow\) `$SHELL`  # exec(3)
Login Process

Let’s revisit the process relationships for a login:

```
kernel  ⇒  init(8)    # explicit creation
init(8)  ⇒  getty(8)  # fork(2)
getty(8) ⇒  login(1)  # exec(3)
login(1) ⇒  $SHELL    # exec(3)
$SHELL  ⇒  ls(1)      # fork(2) + exec(3)
```
Login Process

init(8)  # PID 1, PPID 0, EUID 0
Login Process

init(8)  # PID 1, PPID 0, EUID 0

getty(8)  # PID N, PPID 1, EUID 0
Login Process

init(8)  # PID 1, PPID 0, EUID 0

getty(8) # PID N, PPID 1, EUID 0

login(1) # PID N, PPID 1, EUID 0
Login Process

init(8) # PID 1, PPID 0, EUID 0

getty(8) # PID N, PPID 1, EUID 0

login(1) # PID N, PPID 1, EUID 0

$SHELL # PID N, PPID 1, EUID U
Login Process

init(8) # PID 1, PPID 0, EUID 0

getty(8) # PID $N$, PPID 1, EUID 0

login(1) # PID $N$, PPID 1, EUID 0

$SHELL # PID $N$, PPID 1, EUID $U$

ls(1) # PID $M$, PPID $N$, EUID $U$

pstree -hapun | more
Process Groups

\[ \text{init} \Rightarrow \text{login shell} \]

\$
Process Groups

$ proc1 | proc2 &
[1] 10306
$

init ⇒ login shell
Process Groups

\[
\text{init } \Rightarrow \text{ login shell}
\]

$ \text{proc1 | proc2 } \&$
$[1] \ 10306$

$ \text{proc3 | proc4 | proc5}$
Process Groups

```c
#include <unistd.h>

pid_t getpgrp(void);
pid_t getpgid(pid_t pid);
```

Returns: process group ID if OK, -1 otherwise

- In addition to having a PID, each process also belongs to a process group (collection of processes associated with the same job/terminal)
- Each process group has a unique process group ID
- Process group IDs (like PIDs) are positive integers and can be stored in a `pid_t` data type
- Each process group can have a process group leader
  - Leader identified by its process group ID == PID
  - Leader can create a new process group, create processes in the group
- A process can set its (or its children’s) process group using `setpgid(2)`
A session is a collection of one or more process groups.

If the calling process is not a process group leader, this function creates a new session. Three things happen:

- the process becomes the session leader of this new session
- the process becomes the process group leader of a new process group
- the process has no controlling terminal

```
#include <unistd.h>

pid_t setsid(void);
    Returns: process group ID if OK, -1 otherwise
```
Process Groups

$ proc1 | proc2 &
[1] 10306
$ proc3 | proc4 | proc5
Process Groups and Sessions

\[ \text{init} \rightarrow \text{login shell} \]

\[
\begin{align*}
&\$ \ proc1 \ | \ proc2 \ & \\
&\ \ [1] \ 10306 \\
&\$ \ proc3 \ | \ proc4 \ | \ proc5
\end{align*}
\]
Process Groups and Sessions

```bash
$ ps -o pid,ppid,pgid,sess,comm | ./cat1 | ./cat2

PID  PPID  PGRP  SESS  COMMAND
1989  949  7736  949  ps
1990  949  7736  949  cat1
1988  949  7736  949  cat2
  949  21401  949  949  ksh
```
Job Control

$ ps -o pid,ppid,pgid,sess,comm
   PID   PPID   PGRP   SESS  COMMAND
  24251  24250  24251  24251  ksh
  24620  24251  24620  24251  ps
$

Lecture 06: Process Groups, Sessions, Signals

October 11, 2016
Job Control

$ ps -o pid,ppid,pgid,sess,comm
  PID  PPID  PGRP  SESS  COMMAND
24251 24250 24251 24251 ksh
24620 24251 24620 24251 ps
$ echo $?  
0
$
Job Control

```
$ /bin/sleep 30 &
[1] 24748
$ ps -o pid,ppid,pgid,sess,comm
   PID   PPID   PGRP   SESS COMMAND
24251  24250  24251  24251  ksh
24748  24251  24748  24251  sleep
24750  24251  24750  24251  ps
$
[1] +  Done  /bin/sleep 30 &
$  
```
Job Control
Job Control

$ cat >file
Input from terminal,
Output to terminal.
^D
$ cat file
Input from terminal,
Output to terminal.
$ cat >/dev/null
Input from terminal,
Output to /dev/null.
Waiting forever...
Or until we send an interrupt signal.
^C
$
Job Control

$ cat file &
[1] 2056
$ Input from terminal,
Output to terminal.

[1] + Done cat file &
$ stty tostop
$ cat file &
[1] 4655
$
[1] + Stopped(SIGTTOU) cat file &
$ fg
cat file
Input from terminal,
Output to terminal.
$
Signals

And most of the time it sucks
Signal Concepts

Signals are a way for a process to be notified of asynchronous events. Some examples:

- a timer you set has gone off (SIGALRM)
- some I/O you requested has occurred (SIGIO)
- a user resized the terminal "window" (SIGWINCH)
- a user disconnected from the system (SIGHUP)
- ...

See also: signal(2)/signal(3)/signal(7) (note: these man pages vary significantly across platforms!)
Signal Concepts

Besides the asynchronous events listed previously, there are many ways to generate a signal:

- terminal generated signals (user presses a key combination which causes the terminal driver to generate a signal)
Signal Concepts

Besides the asynchronous events listed previously, there are many ways to generate a signal:

- terminal generated signals (user presses a key combination which causes the terminal driver to generate a signal)
- hardware exceptions (divide by 0, invalid memory references, etc)
Signal Concepts

Besides the asynchronous events listed previously, there are many ways to generate a signal:

- terminal generated signals (user presses a key combination which causes the terminal driver to generate a signal)
- hardware exceptions (divide by 0, invalid memory references, etc)
- `kill(1)` allows a user to send any signal to any process (if the user is the owner or superuser)
Signal Concepts

Besides the asynchronous events listed previously, there are many ways to generate a signal:

- terminal generated signals (user presses a key combination which causes the terminal driver to generate a signal)
- hardware exceptions (divide by 0, invalid memory references, etc)
- `kill(1)` allows a user to send any signal to any process (if the user is the owner or superuser)
- `kill(2)` (a system call, not the unix command) performs the same task
Signal Concepts

Besides the asynchronous events listed previously, there are many ways to generate a signal:

- terminal generated signals (user presses a key combination which causes the terminal driver to generate a signal)
- hardware exceptions (divide by 0, invalid memory references, etc)
- `kill(1)` allows a user to send any signal to any process (if the user is the owner or superuser)
- `kill(2)` (a system call, not the unix command) performs the same task
- software conditions (other side of a pipe no longer exists, urgent data has arrived on a network file descriptor, etc.)
kill(2) and raise(3)

```
#include <sys/types.h>
#include <signal.h>
int kill(pid_t pid, int signo);
int raise(int signo);
```

- **pid > 0** – signal is sent to the process whose PID is pid
- **pid == 0** – signal is sent to all processes whose process group ID equals the process group ID of the sender
- **pid == -1** – POSIX.1 leaves this undefined, BSD defines it (see kill(2))
Signal Concepts

Once we get a signal, we can do one of several things:

- Ignore it. (note: there are some signals which we CANNOT or SHOULD NOT ignore)
Signal Concepts

Once we get a signal, we can do one of several things:

- Ignore it. (note: there are some signals which we CANNOT or SHOULD NOT ignore)

- Catch it. That is, have the kernel call a function which we define whenever the signal occurs.
Signal Concepts

Once we get a signal, we can do one of several things:

- Ignore it. (note: there are some signals which we CANNOT or SHOULD NOT ignore)
- Catch it. That is, have the kernel call a function which we define whenever the signal occurs.
- Accept the default. Have the kernel do whatever is defined as the default action for this signal
Signal Concepts

```
$ cc -Wall ../01-intro/simple-shell.c
$ ./a.out
$$ ^C
$ echo $?  
130
$ cc -Wall ../01-intro/simple-shell2.c
$ ./a.out
$$ ^C
Caught SIGINT!

$$
```
#include <signal.h>

#include <signal.h>

void (*signal(int signo, void (*func)(int)))(int);  

Returns: previous disposition of signal if OK, SIG_ERR otherwise
signal(3)

```c
#include <signal.h>
void (*signal(int signo, void (*func)(int)))(int);
Returns: previous disposition of signal if OK, SIG_ERR otherwise
```

**func** can be:

- **SIG_IGN** which requests that we ignore the signal `signo`
- **SIG_DFL** which requests that we accept the default action for signal `signo`
- the address of a function which should catch or handle a signal
Signal Examples

$ cc -Wall siguser.c
$ ./a.out

^Z
$ bg
$ ps | grep a.out
11106 ttys002 0:00.00 ./a.out
$ kill -USR1 11106
received SIGUSR1
$ kill -USR2 11106
received SIGUSR2
$ kill -INT 11106
$

[2]- Interrupt ./a.out
$

Lecture 06: Process Groups, Sessions, Signals

October 11, 2016
Program Startup

When a program is *exec*ed, the status of all signals is either *default* or *ignore*. 
Program Startup

When a program is `exec`ed, the status of all signals is either `default` or `ignore`.

When a process `fork(2)`s, the child inherits the parent's signal dispositions.
Program Startup

When a program is `exec`ed, the status of all signals is either `default` or `ignore`.

When a process `fork(2)`s, the child inherits the parent's signal dispositions.

A limitation of the `signal(3)` function is that we can only determine the current disposition of a signal by *changing* the disposition.
About that `timeout(1)` then...

$ timeout 60 /bin/sh -c "ls | more"

vs

$ /bin/sh -c "timeout 60 /bin/sh -c "ls | more""
About that `timeout(1) then...`

```
$ timeout 60 /bin/sh -c "ls | more; sleep 60"

$ pstree -hapun
[...]
  `-ksh,10981
  `-sh,12044 -c timeout 60 /bin/sh -c "ls | more; sleep 30"
  `-timeout,12045 60 /bin/sh -c ls | more; sleep 30
  `-sh,12046 -c ls | more; sleep 30
      `-sleep,12049 30
[...]
```

```
$ ps x -o pid,ppid,pgid,sesst,tpgid,stat,comm | egrep -v "(ssh|ps|egrep)"

    PID  PPID  PGID  SESS  TPGID  STAT   COMMAND
    7676  7675  7676  7676  7676  Ss+    ksh
   10981 10980 10981 10981 12044  Ss     ksh
   12044 10981 12044 10981 12044  S+     sh
   12045 12044 12045 10981 12044  S     timeout
   12046 12045 12045 10981 12044  S     sh
   12049 12046 12045 10981 12044  S     sleep
```
About that \texttt{timeout(1)} then...

\begin{verbatim}
$ /bin/sh -c timeout 60 "/bin/sh -c \"ls | more\""

$ pstree -hapun
[...]
  |     | ‘-ksh,10981
  |     | ‘-sh,12434 -c timeout 60 /bin/sh -c "ls | more"
  |     | ‘-timeout,12435 60 /bin/sh -c ls | more
  |     | ‘-sh,12436 -c ls | more
  |     | |-ls,12437
  |     | ‘-more,12438
[...]

$ ps x -o pid,ppid,pgid, sess, tpgid, stat, comm | egrep -v "(ssh|ps|egrep)"

<table>
<thead>
<tr>
<th>PID</th>
<th>PPID</th>
<th>PGID</th>
<th>SESS</th>
<th>TPGID</th>
<th>STAT</th>
<th>COMMAND</th>
</tr>
</thead>
<tbody>
<tr>
<td>7676</td>
<td>7675</td>
<td>7676</td>
<td>7676</td>
<td>7676</td>
<td>Ss+</td>
<td>ksh</td>
</tr>
<tr>
<td>10981</td>
<td>10980</td>
<td>10981</td>
<td>10981</td>
<td>12434</td>
<td>Ss</td>
<td>ksh</td>
</tr>
<tr>
<td>12434</td>
<td>10981</td>
<td>12434</td>
<td>10981</td>
<td>12434</td>
<td>S+</td>
<td>sh</td>
</tr>
<tr>
<td>12435</td>
<td>12434</td>
<td>12435</td>
<td>10981</td>
<td>12434</td>
<td>S</td>
<td>timeout</td>
</tr>
<tr>
<td>12436</td>
<td>12435</td>
<td>12435</td>
<td>10981</td>
<td>12434</td>
<td>T</td>
<td>sh</td>
</tr>
<tr>
<td>12437</td>
<td>12436</td>
<td>12435</td>
<td>10981</td>
<td>12434</td>
<td>T</td>
<td>ls</td>
</tr>
<tr>
<td>12438</td>
<td>12436</td>
<td>12435</td>
<td>10981</td>
<td>12434</td>
<td>T</td>
<td>more</td>
</tr>
</tbody>
</table>
\end{verbatim}
About that `timeout(1)` then...

Use the source, Luke!

coreutils/src/timeout.c

```c
/* Ensure we’re in our own group so all subprocesses can be killed. Note we don’t just put the child in a separate group as then we would need to worry about foreground and background groups and propagating signals between them. */
if (!foreground)
    setpgid (0, 0);

[...]

    signal (SIGTTIN, SIG_DFL);
    signal (SIGTTOU, SIG_DFL);

    execvp (argv[0], argv);
```
About that `timeout(1)` then...

Use the source, Luke!

util-linux/text-utils/more.c

```c
#define stty(fd, argp) tcsetattr(fd, TCSANOW, argp)

if (!no_tty) {
    signal(SIGQUIT, onquit);
    signal(SIGINT, end_it);
#endif /* SIGWINCH */
    signal(SIGWINCH, chgwinsz);
#endif

if (signal(SIGTSTP, SIG_IGN) == SIG_DFL) {
    signal(SIGTSTP, onsusp);
    catch_susp++;
}

stty (fileno(stderr), &otty);
```
sigaction(2)

```
#include <signal.h>

int sigaction(int signo, const struct sigaction *act, struct sigaction *oact);
```

This function allows us to examine or modify the action associated with a particular signal.

```
struct sigaction {
    void (*sa_handler)(); /* addr of signal handler, or
    SIG_IGN or SIG_DFL */
    sigset_t sa_mask;    /* additional signals to block */
    int sa_flags;        /* signal options */
};
```

signal(3) is (nowadays) commonly implemented via sigaction(2).
More advanced signal handling via signal sets

- `int sigemptyset(sigset_t *set)` — initialize a signal set to be empty
- `int sigfillset(sigset_t *set)` — initialize a signal set to contain all signals
- `int sigaddset(sigset_t *set, int signo)`
- `int sigdelset(sigset_t *set, int signo)`
- `int sigismember(sigset_t *set, int signo)`
Resetting Signal Handlers

*Note:* on some systems, invocation of the handler *resets* the disposition to `SIG_DFL`!

```bash
$ cc -DSLEEP=3 -Wall pending.c
$ ./a.out
=> Establishing initial signal hander via signal(3).
\sig_quit: caught SIGQUIT (1), now sleeping
\sig_quit: exiting (1)
=> Time for a second interruption.
\sig_quit: caught SIGQUIT (2), now sleeping
\sig_quit: exiting (2)
=> Establishing a resetting signal hander via signal(3).
\sig_quit_reset: caught SIGQUIT (3), sleeping and resetting.
\sig_quit_reset: restored SIGQUIT handler to default.
=> Time for a second interruption.
\Quit: 3
$```

Lecture 06: Process Groups, Sessions, Signals October 11, 2016
Signal Queuing

Signals arriving while a handler runs are queued.

$ ./a.out >/dev/null

^\sig_quit: caught SIGQUIT (1), now sleeping
^\^\^\^\^\^\sig_quit: exiting (1)
\sig_quit: caught SIGQUIT (2), now sleeping
\sig_quit: exiting (2)
\sig_quit: caught SIGQUIT (3), now sleeping
\sig_quit: exiting (3)
\sig_quit: caught SIGQUIT (4), now sleeping
\sig_quit: exiting (4)
[...]

(Note that "simultaneously" delivered signals may be "merged" into one.)
Signal Queuing

Signals arriving while a handler runs are queued. Unless they are blocked.

$ ./a.out
[...]
=> Establishing a resetting signal handler via signal(3).
\sig_quit_reset: caught SIGQUIT (1), sleeping and resetting.
sig_quit_reset: restored SIGQUIT handler to default.
=> Time for a second interruption.
=> Blocking delivery of SIGQUIT...
=> Now going to sleep for 3 seconds...
\n
=> Checking if any signals are pending...
=> Checking if pending signals might be SIGQUIT...
Pending SIGQUIT found.
=> Unblocking SIGQUIT...
Quit: 3
Signal Queuing

Multiple identical signals are queued, but you can receive a different signal while in a signal handler.

$ ./a.out >/dev/null
\sig_quit: caught SIGQUIT (1), now sleeping
\sig_int: caught SIGINT (2), returning immediately
sig_quit: exiting (2)
sig_quit: caught SIGQUIT (3), now sleeping
\sig.quit: exiting (3)
sig_quit: caught SIGQUIT (4), now sleeping
sig_quit: exiting (4)
[...]
Interrupted System Calls

Some system calls can block for long periods of time (or forever). These include things like:

- `read(2)`s from files that can block (pipes, networks, terminals)
Interrupted System Calls

Some system calls can block for long periods of time (or forever). These include things like:

- `read(2)`s from files that can block (pipes, networks, terminals)
- `write(2)` to the same sort of files
Interrupted System Calls

Some system calls can block for long periods of time (or forever). These include things like:

- `read(2)`s from files that can block (pipes, networks, terminals)
- `write(2)` to the same sort of files
- `open(2)` of a device that waits until a condition occurs (for example, a modem)
Interrupted System Calls

Some system calls can block for long periods of time (or forever). These include things like:

- `read(2)`s from files that can block (pipes, networks, terminals)
- `write(2)` to the same sort of files
- `open(2)` of a device that waits until a condition occurs (for example, a modem)
- `pause(3)`, which purposefully puts a process to sleep until a signal occurs
Interrupted System Calls

Some system calls can block for long periods of time (or forever). These include things like:

- `read(2)`s from files that can block (pipes, networks, terminals)
- `write(2)` to the same sort of files
- `open(2)` of a device that waits until a condition occurs (for example, a modem)
- `pause(3)`, which purposefully puts a process to sleep until a signal occurs
- certain `ioctl(3)`s
- certain IPC functions
Interrupted System Calls

Some system calls can block for long periods of time (or forever). These include things like:

- `read(2)`s from files that can block (pipes, networks, terminals)
- `write(2)` to the same sort of files
- `open(2)` of a device that waits until a condition occurs (for example, a modem)
- `pause(3)`, which purposefully puts a process to sleep until a signal occurs
- `certain ioctl(3)`s
- `certain IPC functions`

Catching a signal during execution of one of these calls traditionally led to the process being aborted with an `errno` return of `EINTR`. 
Interrupted System Calls

Previously necessary code to handle EINTR:

```c
again:
    if ((n = read(fd, buf, BUFFSIZE)) < 0) {
        if (errno == EINTR)
            goto again;  /* just an interrupted system call */
        /* handle other errors */
    }
```

Nowadays, many Unix implementations automatically restart certain system calls.
Interrupted System Calls

$ cc -Wall eintr.c
$ ./a.out
^C
read call was interrupted
||
$ ./a.out
^\a
read call was restarted
|a|
$
Reentrant functions

An example of calling nonreentrant functions from a signal handler:

```
$ cc -Wall reentrant.c; ./a.out
in signal handler
in signal handler
in signal handler
no 'root' found!
$ ./a.out
in signal handler
return value corrupted: pw_name = root
$ ./a.out
in signal handler
in signal handler
User jschauma not found!
$ ./a.out
in signal handler
in signal handler
Memory fault (core dumped)
```
Reentrant Functions

If your process is currently handling a signal, what functions are you allowed to use?

See p. 306 in Stevens for a list.
Homework

Read, try, play with and understand all examples.