CS631 - Advanced Programming in the UNIX Environment
–
Process Groups, Sessions, Signals

Department of Computer Science
Stevens Institute of Technology
Jan Schaumann
jschauma@stevens.edu
http://www.cs.stevens.edu/~jschauma/631/

Login Process

Let's revisit the process relationships for a login:

kernel $\Rightarrow$ init(8) # explicit creation

Login Process

Let's revisit the process relationships for a login:

kernel	$\Rightarrow$	init(8)	# explicit creation
init(8)	$\Rightarrow$	getty(8)	# fork(2)

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

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

Process Groups

```
#include <unistd.h>

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

Returns: process group ID if OK, -1 otherwise

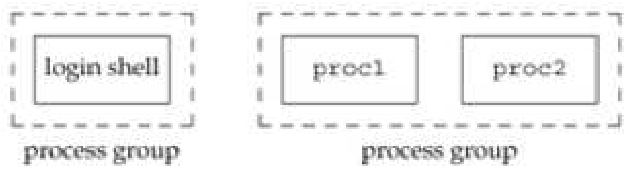
- in addition to having a PID, each process also belongs to a process group (collection of processes assocaited with the same job / terminal)
- each process group has a unique process group ID
- process group IDs (like PIDs) are positive integers and can be stored in a `pid_t` data type
- each process 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)`

Process Groups

init ⇒ *login shell*

$

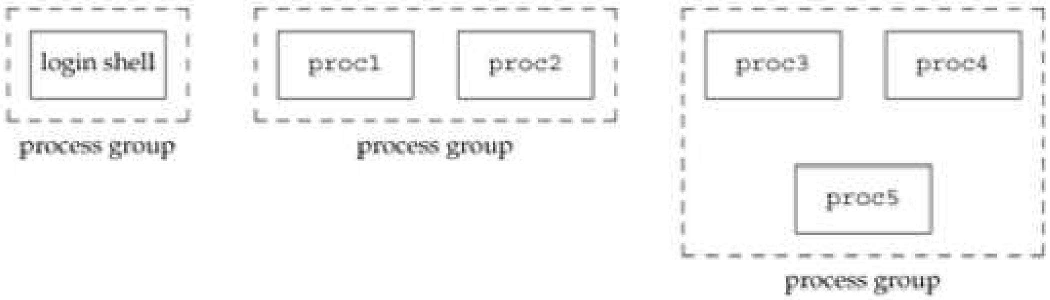
Process Groups



init ⇒ *login shell*

```
$ proc1 | proc2 &
[1] 10306
$
```

Process Groups



init $\Rightarrow$ *login shell*

```
$ proc1 | proc2 &
[1] 10306
$ proc3 | proc4 | proc5
```

Process Groups and Sessions

```
#include <unistd.h>

pid_t setsid(void);
```

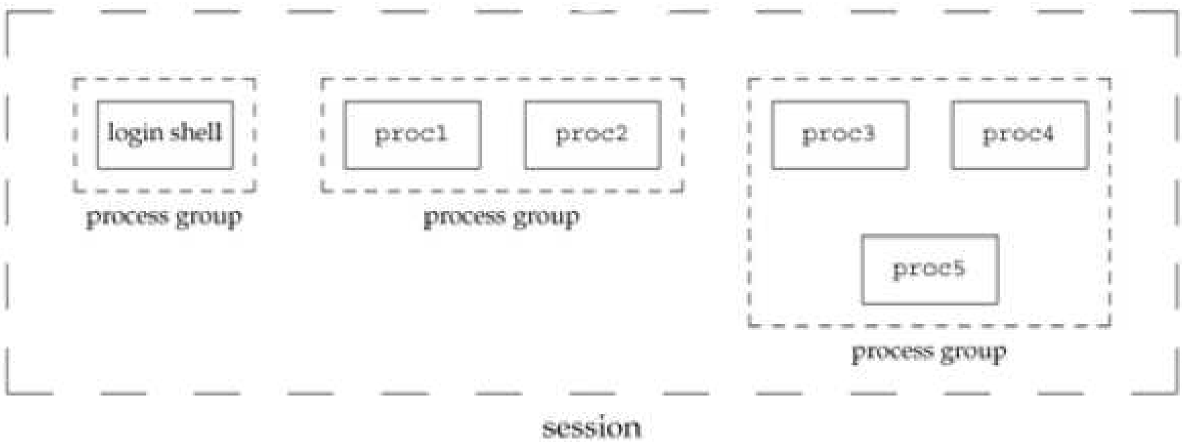
Returns: process group ID if OK, -1 otherwise

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

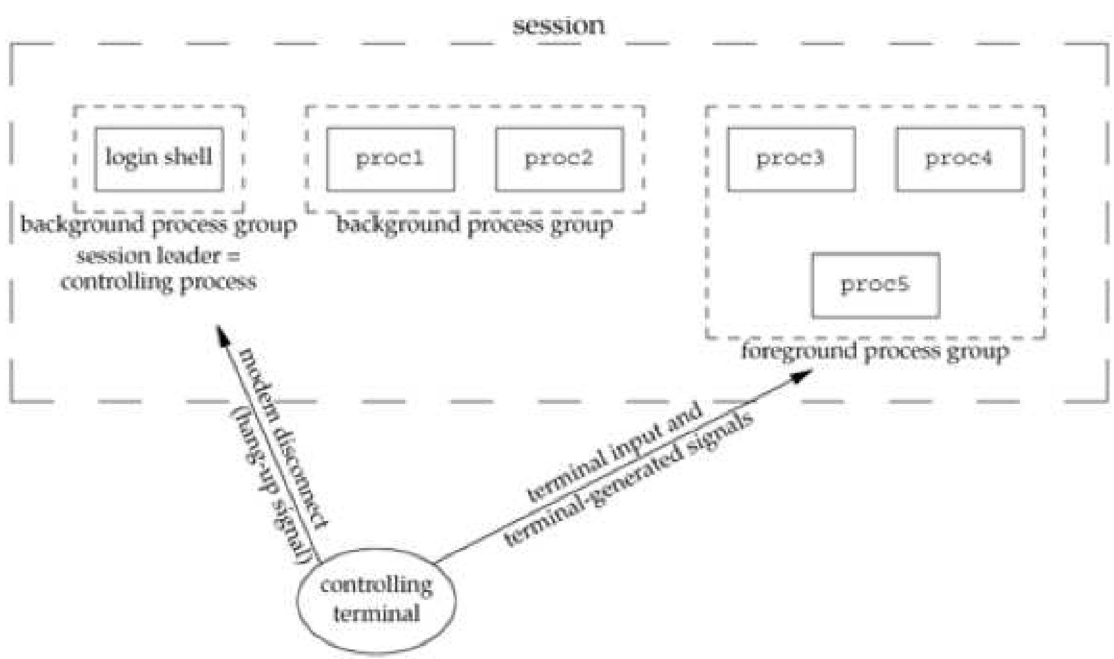
Process Groups



init ⇒ *login shell*

```
$ proc1 | proc2 &
[1] 10306
$ proc3 | proc4 | proc5
```

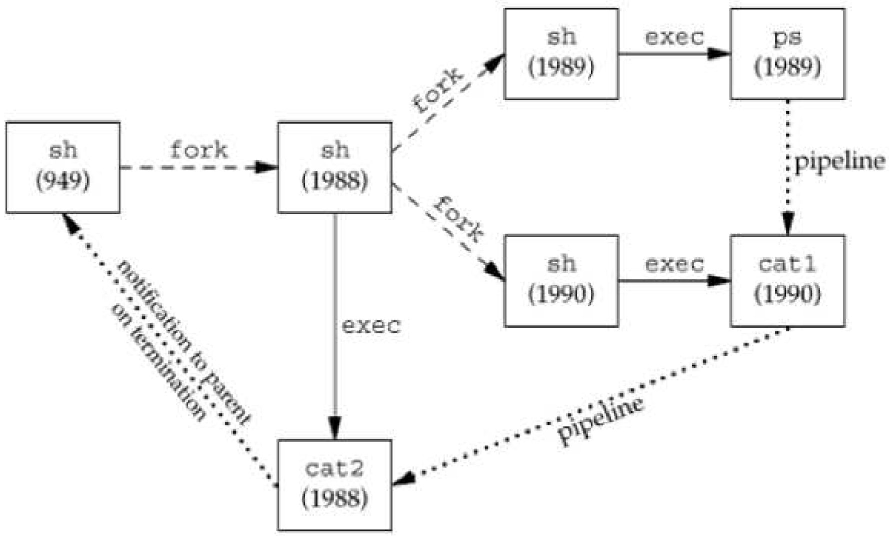
Process Groups and Sessions



init ⇒ *login shell*

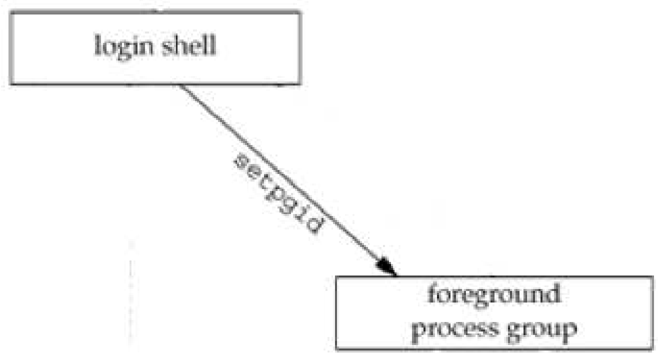
```
$ proc1 | proc2 &
[1] 10306
$ proc3 | proc4 | proc5
```

Process Groups and Sessions



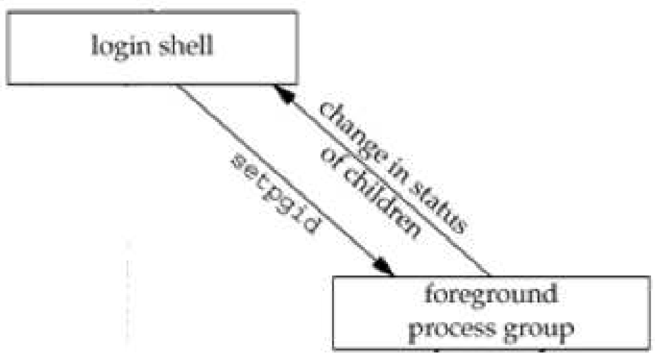

```
$ ps -o pid,ppid,pgid,sess,comm | ./cat1 | ./cat2
  PID  PPID  PGRP  SESS COMMAND
 1989   949  7736   949 ps
 1990   949  7736   949 cat1
 1988   949  7736   949 cat2
  949 21401   949   949 ksh
```

Job Control



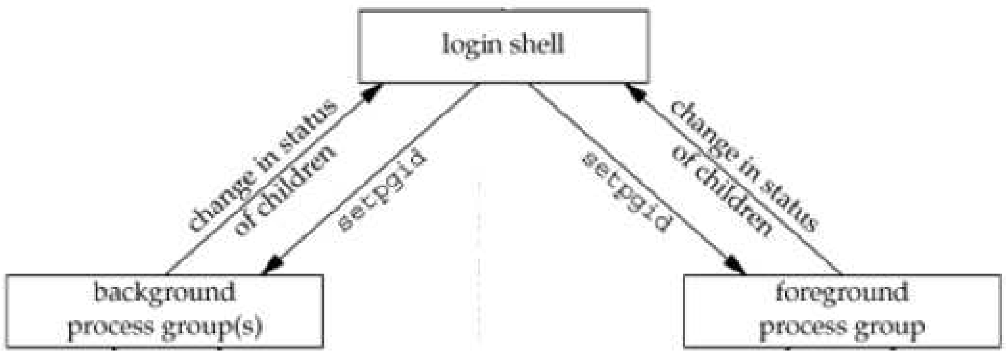

```
$ ps -o pid,ppid,pgid,sess,comm
  PID  PPID  PGRP  SESS COMMAND
24251 24250 24251 24251 ksh
24620 24251 24620 24251 ps
$
```

Job Control



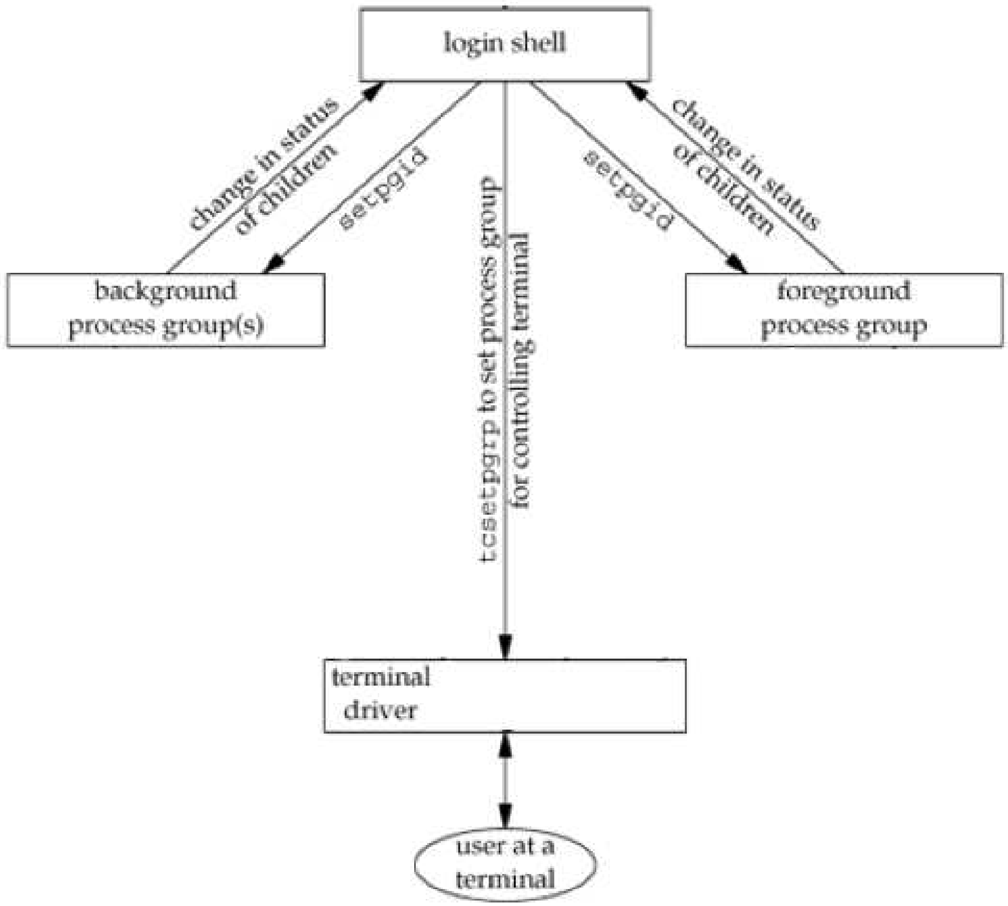

```
$ ps -o pid,ppid,pgid,sess,comm
  PID  PPID  PGRP  SESS COMMAND
24251 24250 24251 24251 ksh
24620 24251 24620 24251 ps
$ echo $?
0
$
```

Job Control



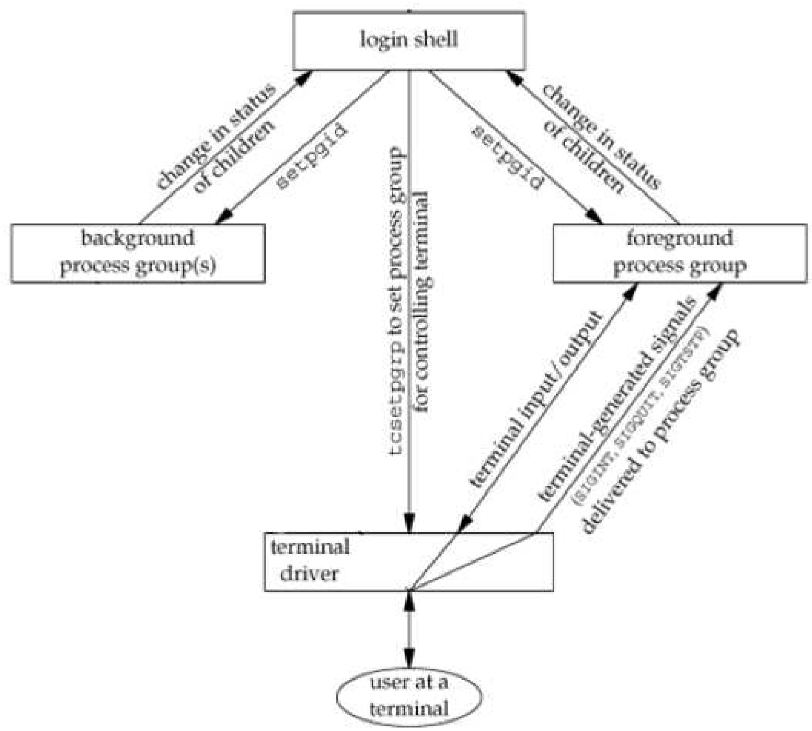

```
$ dd if=/dev/zero of=/dev/null bs=512 count=2048000 >/dev/null 2>&1 &
[1] 24748
$ ps -o pid,ppid,pgid,sess,comm
  PID  PPID  PGRP  SESS COMMAND
24251 24250 24251 24251 ksh
24748 24251 24748 24251 dd
24750 24251 24750 24251 ps
$
[1] +  Done     dd if=/dev/zero of=/dev/null bs=512 count=2048000 >/dev/null 2>&1 &
$
```

Job Control


login shell
change in status
of children
setpgid
tcsetpgrp to set process group
for controlling terminal
setpgid
change in status
of children
background
process group(s)
foreground
process group
terminal
driver
user at a
terminal

Job Control

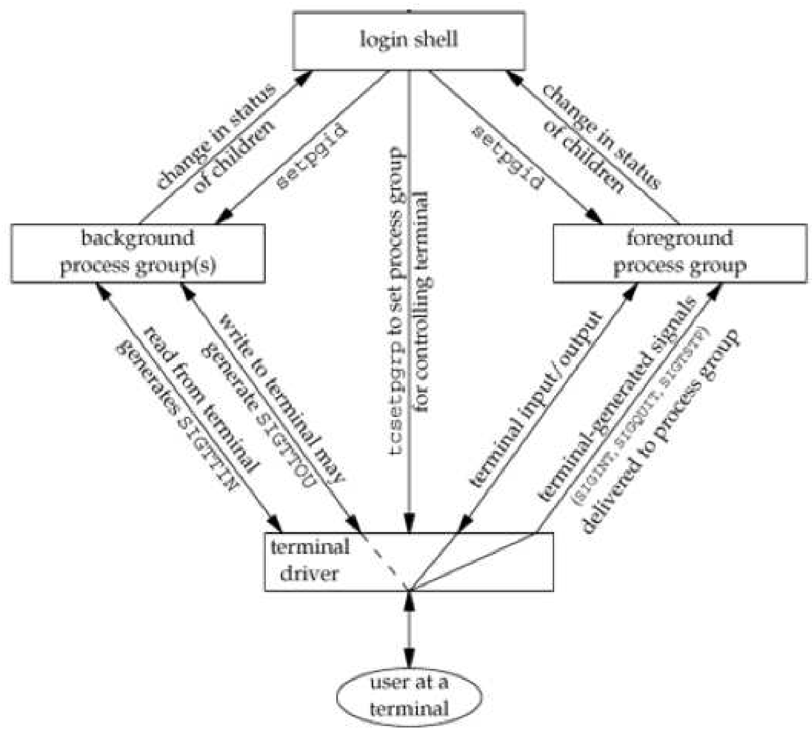
```
$ cat >file
Input from terminal,
Output to terminal.
^D
$ cat file
Input from terminal,
Output to terminal.
$ cat >/dev/null
Input from terminal,
Output to /dev/null.
Waiting forever...
Or until we send an interrupt signal.
^C
$
```



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

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!

$$
```

signal(3)

```
#include <signal.h>

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

Returns: previous disposition of signal if OK, `SIG_ERR` otherwise

signal(3)

```
#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
- or 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.ou[t]
11106 ttys002    0:00.00 ./a.out
$ kill -USR1 11106
received SIGUSR1
$ kill -USR2 11106
received SIGUSR2
$ kill -INT 11106
$
[2]-  Interrupt               ./a.out
$
```

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)` – intialize 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`!

```
$ 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
$
```

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 hander 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...
^\
=> 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
^\^\^\^\^Csig_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)
[...]
```

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.

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`:

```
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.

Work on your midterm project.