Signals and Jumps

CSAPP2e, Chapter 8

Recall: Running a New Program

```
int execl(char *path,
          char *arg0, …, char *argn,
          char *null)
```

- Loads & runs executable:
  - `path` is the complete path of an executable
  - `arg0` becomes the name of the process
  - `arg0`, …, `argn` → `argv[0]`, …, `argv[n]`
  - Argument list terminated by a `NULL` argument
- Returns -1 if error, otherwise doesn't return!

```
if (fork() == 0)
   execl("/usr/bin/cp", "cp", "foo", "bar", NULL);
else
   printf("hello from parent\n");
```


Interprocess Communication

- Synchronization allows very limited communication
- Pipes:
  - One-way communication stream that mimics a file in each process: one output, one input
  - See `man 7 pipe`
- Sockets:
  - A pair of communication streams that processes connect to
  - See `man 7 socket`

The World of Multitasking

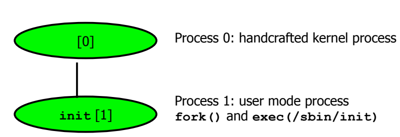
- System Runs Many Processes Concurrently
  - Process: executing program
    - State consists of memory image + register values + program counter
  - Continually switches from one process to another
    - Suspend process when it needs I/O resource or timer event occurs
    - Resume process when I/O available or given scheduling priority
  - Appears to user(s) as if all processes executing simultaneously
    - Even though most systems can only execute one process at a time
    - Except possibly with lower performance than if running alone

Programmer's Model of Multitasking

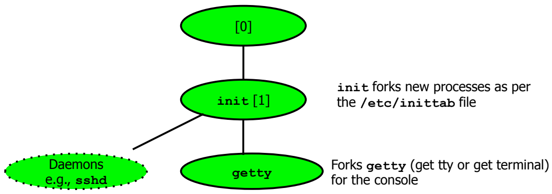
- Basic Functions
  - `fork()` spawns new process
    - Called once, returns twice
  - `exit()` terminates own process
    - Called once, never returns
    - Puts process into "zombie" status
  - `wait()` and `waitpid()` wait for and reap terminated children
  - `execl()` and `execve()` run a new program in an existing process
    - Called once, (normally) never returns
- Programming Challenge
  - Understanding the nonstandard semantics of the functions
  - Avoiding improper use of system resources
    - E.g., "Fork bombs" can disable a system

UNIX Startup: 1

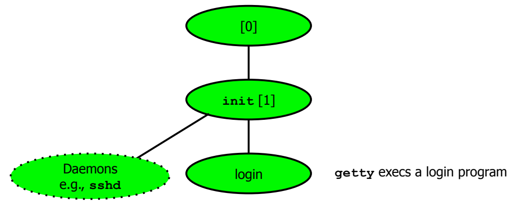
- Pushing reset button loads the PC with the address of a small bootstrap program
- Bootstrap program loads the boot block (disk block 0)
- Boot block program loads kernel from disk
- Boot block program passes control to kernel
- Kernel handcrafts the data structures for process 0



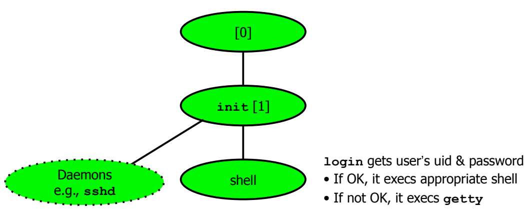
UNIX Startup: 2



UNIX Startup: 3



UNIX Startup: 4



Shell Programs

- A shell is an application program that runs programs on behalf of user
  - sh – Original Unix Bourne Shell
  - csh – BSD Unix C Shell, tcsh – Enhanced C Shell
  - bash – Bourne-Again Shell
  - ksh – Korn Shell

Read-evaluate loop: an interpreter!

```
int main(void)
{
    char cmdline[MAXLINE];
    while (true) {
    /* read */
    printf("> ");
    Fgets(cmdline, MAXLINE, stdin);
    if (feof(stdin))
        exit(0);

    /* evaluate */
    eval(cmdline);
    }
}
```


Simple Shell `eval` Function

```
void eval(char *cmdline)
{
    char *argv[MAXARGS]; /* argv for execve() */
    bool  bg;            /* should the job run in bg or fg? */
    pid_t pid;           /* process id */
    int   status;        /* child status */

    bg = parseline(cmdline, argv);
    if (!builtin_command(argv)) {
    if ((pid = Fork()) == 0) {   /* child runs user job */
        if (execve(argv[0], argv, environ) < 0) {
        printf("%s: Command not found.\n", argv[0]);
        exit(0);
        }
    }
    if (!bg) {   /* parent waits for fg job to terminate */
        if (waitpid(pid, &status, 0) < 0)
        unix_error("waitfg: waitpid error");
    }
    else         /* otherwise, don't wait for bg job */
        printf("%d %s", pid, cmdline);
    }
}
```


Problem with Simple Shell Example

- Correctly waits for & reaps foreground jobs
- But what about background jobs?
  - Will become zombies when they terminate
  - Will never be reaped because shell (typically) will not terminate
  - Creates a process leak that will eventually prevent the forking of new processes
- Solution: Reaping background jobs requires a mechanism called a *signal*

Signals

- A *signal* is a small message that notifies a process that an event of some type has occurred in the system
  - Kernel abstraction for exceptions and interrupts
  - Sent from the kernel (sometimes at the request of another process) to a process
  - Different signals are identified by small integer ID's
  - Typically, the only information in a signal is its ID and the fact that it arrived

ID	Name	Default Action	Corresponding Event
2	`SIGINT`	Terminate	Keyboard interrupt (`ctrl-c`)
9	`SIGKILL`	Terminate	Kill program
11	`SIGSEGV`	Terminate & Dump	Segmentation violation
14	`SIGALRM`	Terminate	Timer signal
18	`SIGCHLD`	Ignore	Child stopped or terminated

Signals: Sending

- OS kernel sends a signal to a destination process by updating some state in the context of the destination process
- Reasons:
  - OS detected an event
  - Another process used the kill system call to explicitly request the kernel to send a signal to the destination process

Signals: Receiving

- Destination process receives a signal when it is forced by the kernel to react in some way to the delivery of the signal
- Three ways to react:
  - Ignore the signal
  - Terminate the process (& optionally dump core)
  - Catch the signal with a user-level signal handler

Signals: Pending & Blocking

- Signal is pending if sent, but not yet received
  - At most one pending signal of any particular type
  - Important: Signals are not queued
    - If process has pending signal of type k, then process discards subsequent signals of type k
  - A pending signal is received at most once
- Process can block the receipt of certain signals
  - Blocked signals can be delivered, but will not be received until the signal is unblocked

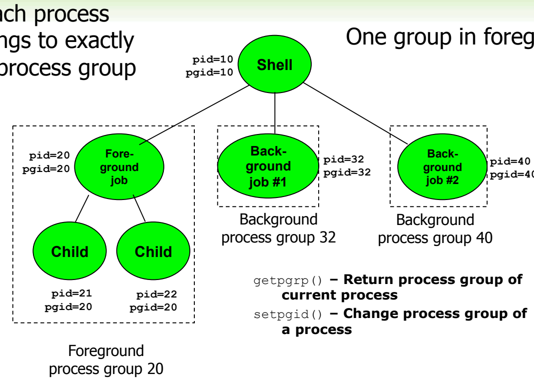
Signals: Pending & Blocking

- Kernel maintains `pending` & `blocked` bit vectors in each process context
- `pending` – represents the set of pending signals
  - Signal type k delivered → kernel sets kth bit
  - Signal type k received → kernel clears kth bit
- `blocked` – represents the set of blocked signals
  - Application sets & clears bits via `sigprocmask()`

Process Groups

Each process belongs to exactly one process group

One group in foreground



`getpgrp()` – **Return process group of current process**

`setpgid()` – **Change process group of a process**

Sending Signals with `/bin/kill`

Sends arbitrary signal to a process or process group

`kill –9 11662`
Send SIGKILL to process 11662

`kill –9 –11661`
Send SIGKILL to every process in process group 11661

```
UNIX% fork2anddie
Child1: pid=11662 pgrp=11661
Child2: pid=11663 pgrp=11661

UNIX% ps x
   PID TTY     STAT TIME COMMAND
 11263 pts/7   Ss   0:00 -tcsh
 11662 pts/7   R    0:18 ./fork2anddie
 11663 pts/7   R    0:16 ./fork2anddie
 11664 pts/7   R+   0:00 ps x
UNIX% kill -9 -11661
UNIX% ps x
   PID TTY     STAT TIME COMMAND
 11263 pts/7   Ss   0:00 -tcsh
 11665 pts/7   R+   0:00 ps x
UNIX%
```


Sending Signals from the Keyboard

- Typing ctrl-c (ctrl-z) sends SIGINT (SIGTSTP) to every job in the foreground process group
  - SIGINT – default action is to terminate each process
  - SIGTSTP – default action is to stop (suspend) each process

Example of `ctrl-c` and `ctrl-z`

```
UNIX% ./fork1
Child: pid=24868 pgrp=24867
Parent: pid=24867 pgrp=24867
<typed ctrl-z>
Suspended
UNIX% ps x
  PID TTY      STAT   TIME COMMAND
24788 pts/2    Ss     0:00 -tcsh
24867 pts/2    T      0:01 fork1
24868 pts/2    T      0:01 fork1
24869 pts/2    R+     0:00 ps x
UNIX% fg
fork1
<typed ctrl-c>
UNIX% ps x
  PID TTY      STAT   TIME COMMAND
24788 pts/2    Ss     0:00 -tcsh
24870 pts/2    R+     0:00 ps x
```

```
S=Sleeping
R=Running or Runnable
T=Stopped
Z=Zombie
```


`kill()`

```
void kill_example(void)
{
    pid_t pid[N], wpid;
    int   child_status, i;
    for (i = 0; i < N; i++)
    if ((pid[i] = fork()) == 0)
        while (1); /* Child infinite loop */

    /* Parent terminates the child processes */
    for (i = 0; i < N; i++) {
    printf("Killing process %d\n", pid[i]);
    kill(pid[i], SIGINT);
    }

    /* Parent reaps terminated children */
    for (i = 0; i < N; i++) {
    wpid = wait(&child_status);
    if (WIFEXITED(child_status))
        printf("Child %d terminated with exit status %d\n",
            wpid, WEXITSTATUS(child_status));
    else
        printf("Child %d terminated abnormally\n", wpid);
    }
}
```


Receiving Signals: How It Happens

- Suppose kernel is returning from an exception handler & is ready to pass control to process p
- Kernel computes `pnb = pending & ~blocked`
  - The set of pending nonblocked signals for process p
- If `pnb == 0`
  - Pass control to next instruction in the logical control flow for p
- Else
  - Choose least nonzero bit k in `pnb` and force process p to receive signal k
  - The receipt of the signal triggers some action by p
  - Repeat for all nonzero k in `pnb`
  - Pass control to next instruction in the logical control flow for p

Signals: Default Actions

- Each signal type has predefined *default action*
- One of:
  - Process terminates
  - Process terminates & dumps core
  - Process stops until restarted by a SIGCONT signal
  - Process ignores the signal

Signal Handlers

- `#include <signal.h>`
- `typedef void (*sighandler_t)(int);`
- `sighandler_t signal(int signum, sighandler_t handler);`
- Two args:
  - `signum` – Indicates which signal, e.g.,
    - SIGSEGV, SIGINT, ...
  - `handler` – Signal "disposition", one of
    - Pointer to a handler routine, whose int argument is the kind of signal raised
    - SIG_IGN – ignore the signal
    - SIG_DFL – use default handler
- Returns previous disposition for this signal
  - Details: `man signal` and `man 7 signal`

Signal Handlers: Example 1

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

void sigint_handler(int sig) {
  printf("Control-C caught.\n");
  exit(0);
}

int main(void) {
  signal(SIGINT, sigint_handler);
  while (true) {
  }
}
```


Signal Handlers: Example 2

```
#include <stdio.h>
#include <signal.h>
#include <stdbool.h>

int ticks = 5;

void sigalrm_handler(int sig) {
  printf("tick\n");

  ticks -= 1;
  if (ticks > 0) {
    signal(SIGALRM,
           sigalrm_handler);
    alarm(1);
  } else {
    printf("*BOOM!*\n");
    exit(0);
  }
}
```

```
int main(void) {
  signal(SIGALRM,
         sigalrm_handler);
  alarm(1); /* send SIGALRM in
               1 second */

  while (true) {
    /* handler returns here */
  }
}
```

`signal` resets handler to default action each time handler runs, `sigset`, `sigaction` do not

```
UNIX% ./alrm
tick
tick
tick
tick
tick
*BOOM!*
UNIX%
```


Signal Handlers (POSIX)

- OS may allow more detailed control:
- `int sigaction(int sig,`
- `               const struct sigaction *act,`
- `               struct sigaction *oact);`
- `struct sigaction` includes a handler:
- `void sa_handler(int sig);`
- `Signal` from `csapp.c` is a clean wrapper around `sigaction`

Pending Signals Not Queued

```
int ccount = 0;

void child_handler(int sig)
{
    int   child_status;
    pid_t pid = wait(&child_status);
    ccount -= 1;
    printf("Received signal %d from process %d\n", sig, pid);
}

void example(void)
{
    pid_t pid[N];
    int   child_status, i;
    ccount = N;
    Signal(SIGCHLD, child_handler);
    for (i = 0; i < N; i+=1)
     if ((pid[i] = fork()) == 0) {
         /* Child: Exit */
         exit(0);
     }
    while (ccount > 0)
     pause();/* Suspend until signal occurs */
}
```

For each signal type, single bit indicates whether a signal is pending

Will probably lose some signals: `ccount` never reaches 0

Living With Non-Queuing Signals

Must check for all terminated jobs: typically loop with `wait`

```
void child_handler2(int sig)
{
    int   child_status;
    pid_t pid;
    while ((pid = waitpid(-1, &child_status, WNOHANG)) > 0) {
    ccount -= 1;
    printf("Received signal %d from process %d\n", sig, pid);
     }
}

void example(void)
{
    . . .
    Signal(SIGCHLD, child_handler2);
    . . .
}
```

More Signal Handler Funkiness

- Consider signal arrival during long system calls, e.g., `read`
- Signal handler interrupts `read()` call
  - Some flavors of Unix (e.g., Solaris):
    - `read()` fails with `errno==EINTER`
    - Application program may restart the slow system call
  - Some flavors of Unix (e.g., Linux):
    - Upon return from signal handler, `read()` restarted automatically
- Subtle differences like these complicate writing portable code with signals
  - `Signal` wrapper in `csapp.c` helps, uses `sigaction` to restart system calls automatically

Signal Handlers (POSIX)

- Handler can get extra information in `siginfo_t` when using `sigaction` to set handlers

  E.g., for SIGSEGV:
  - Whether virtual address didn't map to any physical address, or whether the address was being accessed in a way not permitted (e.g., writing to read-only space)
  - Address of faulty reference

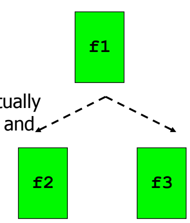
  Details: `man siginfo`

```
static void segv_handler(int sig, siginfo_t *sip, ucontext_t *uap)
{
    fprintf(stderr, "Segmentation fault caught!\n");
    fprintf(stderr, "Caused by access of invalid address %p.\n",
                    sip->si_addr);
    exit(1);
}
```


Other Types of Exceptional Control Flow

- Non-local Jumps
  - C mechanism to transfer control to any program point higher in the current stack

`f1` eventually calls `f2` and `f3`.



When can non-local jumps be used:
- **Yes: f2 to f1**
- **Yes: f3 to f1**

- **No: f1 to either f2 or f3**
- **No: f2 to f3, or vice versa**

Non-local Jumps

- `setjmp()`
  - Identify the current program point as a place to jump to
- `longjmp()`
  - Jump to a point previously identified by `setjmp()`

Non-local Jumps: setjmp()

- `int setjmp(jmp_buf env)`
  - Identifies the current program point with the name `env`
    - `jmp_buf` is a pointer to a kind of structure
    - Stores the current register context, stack pointer, and PC in `jmp_buf`
  - Returns 0

Non-local Jumps: longjmp()

- `void longjmp(jmp_buf env, int val)`
  - Causes another return from the `setjmp()` named by `env`
    - This time, `setjmp()` returns `val`
      - (Except, returns 1 if val==0)
    - Restores register context from jump buffer `env`
    - Sets function's return value register (SPARC: %o0) to `val`
    - Jumps to the old PC value stored in jump buffer `env`
  - `longjmp()` doesn't return!

Non-local Jumps

✧ From the UNIX `man` pages:

```
WARNINGS
     If longjmp() or siglongjmp() are called even though env  was
     never  primed  by a call to setjmp() or sigsetjmp(), or when
     the last  such  call  was  in  a  function  that  has  since
     returned, absolute chaos is guaranteed.
```


Non-local Jumps: Example 1

```
#include <setjmp.h>

jmp_buf  buf;

int main(void)
{
   if (setjmp(buf) == 0)
        printf("First time through.\n");
   else
        printf("Back in main() again.\n");

   f1();
}
```

```
f1()
{
    …
    f2();
    …
}

f2()
{
    …
    longjmp(buf, 1);
    …
}
```


Non-local Jumps: Example 2

```
#include <stdio.h>
#include <signal.h>
#include <setjmp.h>

sigjmp_buf buf;

void handler(int sig)
{
  siglongjmp(buf, 1);
}

int main(void)
{
  Signal(SIGINT, handler);

  if (sigsetjmp(buf, 1) == 0)
    printf("starting\n");
  else
    printf("restarting\n");
  …
```

```
  …
  while(1) {
    sleep(5);
    printf("  waiting...\n");
  }
}
```

```
> a.out
starting

waiting...          <------- Control-c

waiting...
restarting

waiting...          <------- Control-c

waiting...          <------- Control-c

waiting...
restarting
waiting...
```


Application-level Exceptions

✧ Similar to non-local jumps
- Transfer control to other program points outside current block
- More abstract – generally "safe" in some sense
- Specific to application language

Summary: Exceptions & Processes

✧ Exceptions
- Events that require nonstandard control flow
- Generated externally (interrupts) or internally (traps & faults)

✧ Processes
- At any given time, system has multiple active processes
- Only one can execute at a time, though
- Each process appears to have total control of processor & private memory space

Summary: Processes

✧ Spawning
- `fork` – one call, two returns

✧ Terminating
- `exit` – one call, no return

✧ Reaping
- `wait` or `waitpid`

✧ Replacing Program Executed
- `execl` (or variant) – one call, (normally) no return

Summary: Signals & Jumps

- Signals – process-level exception handling
  - Can generate from user programs
  - Can define effect by declaring signal handler
  - Some caveats
    - Very high overhead
      - >10,000 clock cycles
      - Only use for exceptional conditions
    - Don't have queues
      - Just one bit for each pending signal type
- Non-local jumps – exceptional control flow within process
  - Within constraints of stack discipline