§ The Unix Family Tree
BIOS

bootloader

kernel

“handcrafted” process
/etc/inittab \rightarrow init \rightarrow kernel

fork & exec
“Daemons”
e.g., sshd, httpd

fork & exec

kernel

init

fork & exec

getty
The diagram illustrates the relationship between the kernel, init, shell, and exec stages in a computer system. The kernel initiates the process, followed by init which then activates the shell (e.g., sh) to execute the commands.
kernel

init

shell (e.g. sh)

user process

user process

user process

user process

(a **fork-ing** party!)
kernel

init

display manager (e.g., xdm)

X Server (e.g., XFree86)

window manager (e.g., twm)

(or, for the GUI-inclined)
window manager (e.g. twm) ➔

terminal emulator (e.g. xterm) ➔

shell (e.g. sh) ➔

user process

user process

user process

user process
§ The Shell (*aka* the CLI)
the original operating system user interface
essential function: let the user issue requests to the operating system

e.g., fork/exec a program,

manage processes (list/stop/term),

browse/manipulate the file system
(a read-eval-print-loop REPL for the OS)
pid_t pid;
char buf[80], *argv[10];

while (1) {
    /* print prompt */
    printf("$ ");

    /* read command and build argv */
    fgets(buf, 80, stdin);
    for (i=0, argv[0] = strtok(buf, " 
");
        argv[i];
        argv[++i] = strtok(NULL, " 
");
    
    /* fork and run command in child */
    if ((pid = fork()) == 0)
        if (execvp(argv[0], argv) < 0) {
            printf("Command not found\n");
            exit(0);
        }

    /* wait for completion in parent */
    waitpid(pid, NULL, 0);
}
Demo:

examples/processes/simple_shell1.c
... but we are far from done :-}


all shells provide *task management* features
i.e., to run, track and manage *multiple*
processes at a time
distinguish between *foreground* (fg) and *background* (bg) processes

- fg process “blocks” additional commands from being run

- can have multiple bg processes at once
some shell conventions:

- start bg process: `prog_name` &
- `fg/bg`: move a process into fg/bg
Demo:

/bin/zsh
fgets(buf, 80, stdin);

/* check if bg job requested */
if (buf[strlen(buf)-2] == '&') {
    bg = 1;
    buf[strlen(buf)-2] = 0;
} else
    bg = 0;

for (i=0, argv[0] = strtok(buf, " 
");
    argv[i];
    argv[++i] = strtok(NULL, " 
");

/* fork and run command in child */
if ((pid = fork()) == 0)
    if (execvp(argv[0], argv) < 0) {
        printf("Command not found\n");
        exit(0);
    }

/* wait for completion only if bg */
if (!bg) {
    waitpid(pid, NULL, 0);
}
Demo:

examples/processes/simple_shell2.c
background zombies!!!
/* background zombie reaping? */

if (!bg) {
    /* wait for fg job completion */
    waitpid(pid, NULL, 0);
}

/* ... and reap all bg zombies at once */
while (waitpid(-1, NULL, WNOHANG) > 0);
(this is a hack.)

- inefficient & ugly

- no guarantee when reaping will occur
what we really want is a way to be notified when a child turns into a zombie

… so that we can run our reaping code
“notification” → exceptional control flow
§ Signals
signals are messages delivered by the kernel to user processes

- in response to OS events (e.g., segfault)
- or at the request of other processes
how “delivered”?  
- by executing a *handler function* in the receiving process
aspects of signal processing:

1. *sending* a signal to a process
2. *registering* a handler for a given signal
3. *delivering* a signal (kernel mechanism)
4. *designing* a signal handler
1. sending a signal to a process

```c
int kill(pid_t pid, int sig);
```
<table>
<thead>
<tr>
<th>No</th>
<th>Name</th>
<th>Default Action</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>SIGHUP</td>
<td>terminate process</td>
<td>terminal line hangup</td>
</tr>
<tr>
<td>2</td>
<td>SIGINT</td>
<td>terminate process</td>
<td>interrupt program</td>
</tr>
<tr>
<td>3</td>
<td>SIGQUIT</td>
<td>create core image</td>
<td>quit program</td>
</tr>
<tr>
<td>6</td>
<td>SIGABRT</td>
<td>create core image</td>
<td>abort program (formerly SIGIOT)</td>
</tr>
<tr>
<td>9</td>
<td>SIGKILL</td>
<td>terminate process</td>
<td>kill program</td>
</tr>
<tr>
<td>10</td>
<td>SIGBUS</td>
<td>create core image</td>
<td>bus error</td>
</tr>
<tr>
<td>11</td>
<td>SIGSEGV</td>
<td>create core image</td>
<td>segmentation violation</td>
</tr>
<tr>
<td>12</td>
<td>SIGSYS</td>
<td>create core image</td>
<td>non-existent system call invoked</td>
</tr>
<tr>
<td>13</td>
<td>SIGPIPE</td>
<td>terminate process</td>
<td>write on a pipe with no reader</td>
</tr>
<tr>
<td>14</td>
<td>SIGALRM</td>
<td>terminate process</td>
<td>real-time timer expired</td>
</tr>
<tr>
<td>17</td>
<td>SIGSTOP</td>
<td>stop process</td>
<td>stop (cannot be caught or ignored)</td>
</tr>
<tr>
<td>18</td>
<td>SIGTSTP</td>
<td>stop process</td>
<td>stop signal generated from keyboard</td>
</tr>
<tr>
<td>19</td>
<td>SIGCONT</td>
<td>discard signal</td>
<td>continue after stop</td>
</tr>
<tr>
<td>20</td>
<td>SIGCHLD</td>
<td>discard signal</td>
<td>child status has changed</td>
</tr>
<tr>
<td>30</td>
<td>SIGUSR1</td>
<td>terminate process</td>
<td>User defined signal 1</td>
</tr>
<tr>
<td>31</td>
<td>SIGUSR2</td>
<td>terminate process</td>
<td>User defined signal 2</td>
</tr>
</tbody>
</table>
```c
int main () {
    int stat;
    pid_t pid;
    if ((pid = fork()) == 0)
        while(1) ;
    else {
        kill(pid, SIGINT);
        wait(&stat);
        if (WIFSIGNALED(stat))
            psignal(WTERMSIG(stat),
                    "Child term due to");
    }
}
```

Child term due to: Interrupt
sometimes it’s convenient to be able to send a signal to *multiple* processes at once
mechanism: *process groups*

- each process belongs to a *process group*, identified by group id (PGID)
  - PGIDs are positive integers, and in a separate namespace from PIDs
- processes inherit their parents’ PGIDs
/* set pid's group to given pgid */
int setpgid(pid_t pid, pid_t pgid);

- if pid=0, alter the calling process
- if pgid=0, set the process’s PGID equal to its PID
int kill(pid_t pid, int sig);

- if kill is given a negative pid, signal is sent to all processes with PGID=abs(pid)
shell
pid=10, pgid=10

fork

child process
pid=11, pgid=10
shell
pid=10, pgid=10

child process
pid=11, pgid=11

setpgid(0,0)
shell
pid=10, pgid=10

group leader
pid=11, pgid=11

fork

fork

fork

group member
pid=12, pgid=11

fork

group member
pid=13, pgid=11

fork

fork

group member
pid=14, pgid=11
kill(-11, SIGINT)
2. registering a handler for a given signal

typedef void (*sig_t) (int);
sig_t signal(int sig, sig_t func);
sig_t signal(int sig, sig_t func);

- func is typically a pointer to a signal handler function — “callback” API

- some signals cannot be caught!
  (e.g., SIGKILL)
sig_t signal(int sig, sig_t func);

- func can also take special values:
  - SIG_IGN: ignore signal
  - SIG_DFL: use default action
```c
int main () {
    signal(SIGINT, SIG_IGN);
    kill(getpid(), SIGINT);
    while(1) {
        sleep(1);
        printf("And I still live!!!\n");
    }
    return 0;
}
```

And I still live!!!
And I still live!!!
^CAnd I still live!!!
And I still live!!!
^C^CAnd I still live!!!
^C^C^CAnd I still live!!!
Q: how does \(^C \rightarrow\) SIGINT?

A: the terminal emulator (tty device) maps keystrokes to signals, which are sent to the session leader’s process group (typically, login shell)
$ stty -a
speed 9600 baud; 50 rows; 110 columns;
...
cchars: discard = ^O; dsusp = ^Y; eof = ^D; intr = ^C;
  lnext = ^V; quit = ^\; reprint = ^R; start = ^Q;
  status = ^T; stop = ^S; susp = ^Z; werase = ^W;
controlling tty

SIGINT

shell
pid=10, pgid=10

must forward signal to FG group

user process
pid=11, pgid=11

user process
pid=12, pgid=12

user process
pid=13, pgid=12
† child processes inherit their parent’s signal handlers!
‡ but lose them when exec-ing a program
void sigint_handler (int sig) {
    printf("Signal %d received\n", sig);
    sleep(1);
}

int main () {
    signal(SIGINT, sigint_handler);
    while (1) {
        pause(); /* pauses until signal */
        printf("Back in main\n");
    }
}
Demo:

examples/processes/sighandler1.c
3. *delivering* a signal (kernel mechanism)
per-process kernel structures: 2 bit vectors
- “pending” — 1 bit per pending signal
- “blocked” — 1 bit per blocked signal
adjusting blocked signals (*signal mask*):

```
int sigprocmask(int how, /* SIG_BLOCK, SIG_UNBLOCK, or SIG_SETMASK */
    const sigset_t *set, /* specified signals */
    sigset_t *oset);    /* gets previous mask */
```

(SIGKILL & SIGTSTP can’t be blocked!)
note: a newly forked child will inherit its parent’s blocked vector, but its pending vector will start out empty!
<table>
<thead>
<tr>
<th>pending</th>
<th>blocked</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>31</td>
</tr>
<tr>
<td>0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0</td>
<td>0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0</td>
<td>0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0</td>
</tr>
</tbody>
</table>
pending

| 31 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

blocked

| 31 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

```c
sigset_t mask;
sigemptyset(&mask);
sigaddset(&mask, SIGINT); /* SIGINT = 2 */
sigaddset(&mask, SIGALRM); /* SIGALRM = 14 */
sigprocmask(SIG_BLOCK, &mask, NULL);
```
sigset_t mask;
sigemptyset(&mask);
sigaddset(&mask, SIGINT); /* SIGINT = 2 */
sigaddset(&mask, SIGALRM); /* SIGALRM = 14 */
sigprocmask(SIG_BLOCK, &mask, NULL);
kill(the_pid, SIGINT);

```c
sigset_t mask;
sigemptyset(&mask);
sigaddset(&mask, SIGINT); /* SIGINT = 2 */
sigaddset(&mask, SIGALRM); /* SIGALRM = 14 */
sigprocmask(SIG_BLOCK, &mask, NULL);
```
```c
sigset_t mask;
sigemptyset(&mask);
sigaddset(&mask, SIGINT); /* SIGINT = 2 */
sigaddset(&mask, SIGALRM); /* SIGALRM = 14 */
sigprocmask(SIG_BLOCK, &mask, NULL);

kill(the_pid, SIGINT);
```

```
<table>
<thead>
<tr>
<th>Pending</th>
<th>Blocked</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0</td>
<td></td>
</tr>
<tr>
<td>0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0</td>
<td></td>
</tr>
</tbody>
</table>
```
pending | 31
---|---
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0

blocked | 31
---|---
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0

before resuming this process, kernel computes pending & ~blocked

pending | 31
---|---
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0

& ~blocked | 31
---|---
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 1

31 | 0
---|---
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
(pending & ~blocked) ⇒ 0

i.e., no signals to deliver — resume regular control flow
```c
kill(the_pid, SIGTERM);
kil(the_pid, SIGUSR1);
```
<table>
<thead>
<tr>
<th>pending</th>
<th>&amp; ~blocked</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0</td>
<td>1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 1 1 1 1 1 1 1 1 1 0 1 1</td>
</tr>
<tr>
<td>0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0</td>
<td>0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0</td>
</tr>
</tbody>
</table>
deliver signals in order
(i.e., ignore, terminate, or run handler)
/* (user space code) */
void handler(int sig) {
    ...
}

010000000000000000000001000000000000000000000000000000
/*(user space code)*/

void handler(int sig) {
   ...
}

mark signal as “delivered”
(and block this signal until the handler returns)
Q: what happens if a signal is received as its handler is running?
/* (user space code) */

void handler(int sig) {
    ...
    ...
    ...
    ...
}

A: mark it as pending, but don’t run the handler again! (signal currently blocked)
Q: what happens if a signal is sent many times before its handler is run?
Q: what can we do?
A: nothing. (we can’t queue signals!)
Q: what happens if a signal is received as a handler for a lower priority one is already running?
A: we *preempt* the lower priority handler (and resume it — if possible — later)
4. designing a signal handler
Q: what can go wrong?
```c
struct foo { int x, y, z; } f;

int main () {
    int i = 1;

    f = (struct foo){ 0, 0, 0 };

    signal(SIGALRM, tick);

    alarm(1); /* send SIGALRM in 1s */

    while(1) {
        f = (struct foo){ i, i, i };  
        i = (i + 1) % 100;
    }
}

void tick(int s) {
    printf("%d %d %d\n", f.x, f.y, f.z);
    alarm(1); /* send SIGALRM in 1s */
}
```
```c
int main () {
    int i;
    signal(SIGUSR1, handler);
    signal(SIGUSR2, handler);
    for (i=0; i<10; i++) {
        if (fork() == 0) {
            while (1) {
                kill(getppid(), SIGUSR1);
                kill(getppid(), SIGUSR2);
            }
        }
    }
    while(1) pause();
}

void handler(int s) {
    static int x = 10, y = 20;
    int tmp = x;
    x = y;
    y = tmp;
    printf("%d %d\n", x, y);
}
```
```c
int x = 10, y = 20;

int main () {
    int i;
    signal(SIGUSR1, handler1);
    signal(SIGUSR2, handler2);
    for (i=0; i<10; i++) {
        if (fork() == 0)
            while (1) {
                kill(getppid(), SIGUSR1);
                kill(getppid(), SIGUSR2);
            }
    }
    while(1) pause();
}

void handler1(int s) { swapglobs(); }

void handler2(int s) { swapglobs(); }

void swapglobs() {
    int tmp = x;
    x = y;
    y = tmp;
    printf("%d %d\n", x, y);
}
```
lesson 1: signals can be delivered at any time
- may interrupt any nonatomic operation
- problematic if using global variables!
**design goal 1**: minimize use of global variables in sighandlers

- if needed, ideally use data that can be read/written atomically (*most* primitives)
lesson 2: a sighandler may execute in overlapping fashion (with itself)

- when used to handle multiple signals
design goal 2: prefer separate handlers for different signals

- otherwise, must design handlers to be *reentrant* — i.e., able to be called again (re-entered) when already executing
lesson 3: execution of sighandlers for separate signals may overlap

- any functions they call may have overlapping execution
design goal 3: keep sighandlers simple; minimize calls to other functions

- any functions called by sighandlers should be reentrant!
Back to background job reaping ...
int main () {
  ...
  while (1) {
    ...
    fgets(buf, 100, stdin);
    ...
    if ((pid = fork()) == 0) {
      if (execvp(argv[0], argv) < 0) {
        printf("Command not found\n");
        exit(0);
      }
    }
    
    if (!bg) {
      waitpid(pid, NULL, 0);
    }
  }
  ...
}
```c
int main () {
    ...
    signal(SIGCHLD, sigchld_handler);

    while (1) {
        ...
        if ((pid = fork()) == 0) {
            ...
        }
        if (!bg) {
            waitpid(pid, NULL, 0);
        }
    }
}

void sigchld_handler(int sig) {
    pid_t pid;
    printf("sigchld handler called\n");
    while ((pid = waitpid(-1, NULL, WNOHANG)) > 0) {
        /* Q: why a loop? */
        printf("Reaping in sigchld handler\n");
    }
}
```

pid_t fg_pid = -1;

int main () {
    ...
    signal(SIGCHLD, sigchld_handler);

    while (1) {
        ...

        ❶ if ((pid = fork()) == 0) {
            ...
        }

        if (!bg) {
            ❷ fg_pid = pid;
            while (fg_pid != -1)
            sleep(1);
        } ❸
    }

    
    void sigchld_handler(int sig) {
        pid_t pid;
        printf("sigchld handler called\n");
        while ((pid = waitpid(-1, NULL, WNOHANG)) > 0) {
            printf("Reaping in sigchld handler\n");
            if (fg_pid == pid)
            ❹ fg_pid = -1;
        }
    }

    $ sleep 1 &
    $ sigchld handler called
    $ sleep 1
    sigchld handler called
    $ sleep 1
    sigchld handler called
    Reaping in sigchld handler
    $
pid_t fg_pid = -1;

int main () {
    ...
    signal(SIGCHLD, sigchld_handler);

    while (1) {
        ...
        1 if ((pid = fork()) == 0) {
            ...
        }
        if (!bg) {
            4 fg_pid = pid;
            while (fg_pid != -1)
                sleep(1);
        }
    }
    ...
}

void sigchld_handler(int sig) {
    pid_t pid;
    printf("sigchld handler called\n");
    while ((pid = waitpid(-1, NULL, WNOHANG)) > 0) {
        printf("Reaping in sigchld handler\n");
        3 if (fg_pid == pid)
            fg_pid = -1;
    }
}
insidious *race condition* caused by *concurrency* (can’t predict when child will terminate / when signal will arrive)

need to ensure that certain sequences of events *cannot be interrupted*
direct approach: block signals
```c
int main () {
    sigset_t mask;
    sigemptyset(&mask);
    sigaddset(&mask, SIGCHLD);
    ...
    while (1) {
        ...
        sigprocmask(SIG_BLOCK, &mask, NULL);
        ❶ if ((pid = fork()) == 0) {
            ...
        }
        if (!bg) {
            ❷ fg_pid = pid;
            sigprocmask(SIG_UNBLOCK, &mask, NULL);
            while (fg_pid != -1)
                sleep(1);
        }
    }
    ...
}

void sigchld_handler(int sig) {
    ...
    while ((pid = waitpid(-1, NULL, WNOHANG)) > 0) {
        if (fg_pid == pid)
            fg_pid = -1;
    }
}
```

**ensures** ❶, ❷ cannot be interrupted by ❸

SIGCHLD is blocked!
† can also block signals when forced to call non-reentrant functions from sighandlers