Unix Process Hierarchy

```
(0)

shell (1)

Daemon, e.g. topaz (2)

Child

Grandchild
```

The World of Multitasking

System Runs Many Processes Concurrently
- Process: executing program
- Continuously 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 execute only 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, never returns
- `wait()`: terminates own process
  - Called once, never returns
- `waitpid()`: wait for and reap terminated children
- `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.

The `ps` command

```
ps aux --forest (output edited to fit slide)
```
### Unix Startup: Step 1

1. Pushing reset button loads the inc with the address of a small bootstrap program.
2. Bootstrap program loads the boot block (disk block 0).
3. Boot block program loads kernel binary (e.g., `boot/vmlinux`).
4. Boot block program passes control to kernel.
5. Kernel handcrafts the data structures for process 0.

- Process 0: handcrafted kernel process
- Child process 1 execs `/sbin/init`

### Some PC Start-up Details

- **Boot Disk / CD / Floppy**
- **CPU**
- **BIOS ROM**

- **Power on/Reset**

- **Copy Master Boot Record (MBR) into memory**
Some PC Start-up Details

CPU

Boot Disk / CD / Floppy

BIOS verifies MBR and jumps to 0x00007c00

Some PC Start-up Details

CPU

Boot Disk / CD / Floppy

BIOS ROM

LILLO (or GRUB) is loaded from first sector of active partition

Some PC Start-up Details

CPU

Boot Disk / CD / Floppy

BIOS ROM

CPU executes LILO

Linux kernel is loaded and begins initialization

Unix Startup: Step 2

init [1] forks and execs daemons per /etc/inittab, and forks and execs a getty program for the console

getty

Unix Startup: Step 3

init [1] forks and execs the login program
Unix Startup: Step 4

init

login

csh

login reads login-ID and passwd. If OK, it execs a shell. If not OK, it execs another getty.

Shell Programs

A shell is an application program that runs programs on behalf of the user.

- sh - Ancient Unix shell (Stephen Bourne, AT&T Labs, 1977)
- csh - BSD Unix "C shell"
- tcsh - csh enhanced at CMU and elsewhere
- bash - "Bourne-Again" Shell

Main

Shell Correctly Waits for and Reaps Foreground Jobs.

But what about background jobs?

- Will become zombies when they terminate.
- Will never be reaped because shell (typically) will not terminate.
- Will create a memory leak that could theoretically run the kernel out of memory.
- Modern Unix: once you exceed your process quota, your shell can't run any new commands for you: exit() returns -1

Problem with Simple Shell Example

Shell correctly waits for and reaps foreground jobs.

Shell to the Rescue!

Problem

- The shell doesn't know when a background job will finish.
- By nature, it could happen at any time.
- The shell's regular control flow can't reap exited background processes in a timely fashion.
- Regular control flow is "wait until running job completes, then reap it.

Solution: Exceptional control flow

- The kernel will interrupt regular processing to alert us when a background process completes.
- In Unix the alert mechanism is 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 (1-30)

The only information in a signal is its ID and the fact that it arrived.

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Default Action</th>
<th>Corresponding Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>SIGINT</td>
<td>Ignore</td>
<td>SIGTERM</td>
</tr>
<tr>
<td>9</td>
<td>SIGKILL</td>
<td>Ignore</td>
<td>SIGKILL</td>
</tr>
<tr>
<td>11</td>
<td>SIGSEGV</td>
<td>Ignore</td>
<td>SIGSEGV</td>
</tr>
<tr>
<td>14</td>
<td>SIGSTOP</td>
<td>Ignore</td>
<td>SIGSTOP</td>
</tr>
<tr>
<td>17</td>
<td>SIGCHLD</td>
<td>Ignore</td>
<td>SIGCHLD</td>
</tr>
</tbody>
</table>

Timer signal

- SIGALRM
- SIGRTMIN

Segmentation violation

- SIGSEGV
- SIGRTMIN

Child stopped or terminated

- SIGCHLD
- SIGRTMIN

Kill program (cannot override or ignore)

- SIGKILL
- SIGRTMIN

Interrupt from keyboard (ctl-c)

- SIGINT
- SIGRTMIN

Signal Concepts

Sending a signal

- Kernel sends (delivers) a signal to a destination process by updating some state in the context of the destination process.
- Kernel sends a signal for one of the following reasons:
  - Kernel has detected a system event such as divide-by-zero (SIGFPE) or the termination of a child process (SIGCHLD)
  - Another process has invoked the kill system call to explicitly request the kernel to send a signal to the destination process.

Signal Concepts (continued)

Receiving a signal

A destination process receives a signal when it is forced by the kernel to react in some way to the delivery of the signal.

Three possible ways to react:

- Ignore the signal (do nothing)
- Terminate the process (with optional core dump)
- Catch the signal by executing a user-level function called a signal handler.

Akin to a hardware exception handler being called in response to an asynchronous interrupt.

Signal Concepts (continued)

A signal is pending if it has been sent but not yet received.

- There can be at most one pending signal of any particular type.
- Important: Signals are not queued.
  - If a process has a pending signal of type k, then subsequent signals of type k that are sent to that process are discarded.

A process can block the receipt of certain signals.

Blocked signals can be delivered, but will not be received until the signal is unblocked.

A pending signal is received at most once.

Signal Concepts

Kernel maintains pending and blocked bit vectors in the context of each process.

- pending—represents the set of pending signals
  - Kernel clears bit k in pending whenever a signal of type k is delivered.
- blocked—represents the set of blocked signals
  - Can be set and cleared by the application using the sigprocmask function.

Process Groups

Every process belongs to exactly one process group

getpgrp()—Return process group of current process
setpgid()—Change process group of a process
Installing Signal Handlers

The **signal** function modifies the default action associated with the receipt of signal **signal**:  
```c
void (*signal(int signum, void (*handler)(int)));
```

Different values for **handler**:
- **SIG_IGN**: Ignores signals of type **signal**.
- **SIG_DFL**: Reverts to the default action.
- **SIG_SET**: Set handler to **handler**.

## Receiving Signals

Suppose kernel is returning from an exception handler and is ready to pass control to process **p**.

### Kernel computes **pnb** = **pending** & ~**blocked**
- The set of pending nonblocked signals for process **p**

**pnb** = 0
- Pass control to next instruction in the logical 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 logical flow for **p**.

```c
int i, child_status;
pid_t pid[N];

for (i = 0; i < N; i++) {
    pid[i] = fork();
    if (pid[i] == 0) {
        exit(0);
    } else {
        printf("Child %d terminated with exit status %d\n", wpid, WEXITSTATUS(child_status));
    }
}
```

## Sending Signals with **kill** Program

- **kill** program sends arbitrary signal to a process or process group.

**Examples**
- **kill** -9 24818
- Send **SIGINT** to process 24818
- **kill** -9 -24817
- Send **SIGINT** to every process in process group 24817.

## Sending Signals with **kill** Function

```c
void sendkill():
    kill(pid[i], SIGINT);
```

## Default Actions

Each signal type has a predefined **default action**, which is one of:
- The process terminates
- The process stops until restarted by a SIGCONT signal.
- The process ignores the signal.

```
0x22
```

## Signal Handling Example

```c
void sig_handler(void sig)
    printf("Received signal %d\n", sig);
}
```

```
0x57
```
Signals Handlers as Concurrent Flows

A signal handler is a separate logical flow (thread) that runs concurrently with the main program.

- “Concurrently” in the “non-sequential” sense

```
while (1) handler()
```

Signal Handler Funkiness

Pending signals are not queued

- For each signal type, kernel has one bit indicating whether or not signal is pending
- Even if multiple processes have sent this signal

```
count = 0;
void child_handler(int sig)
{
  int child_status;
  pid_t pid = wait(&child_status);
  count--;
  printf("Received signal %d from process %d
", sig, pid);
}
```

A Program That Reacts to Externally Generated Events (ctrl-c)

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

void handler(int sig)
{
  if (sig == SIGINT) {
    printf("You think hitting ctrl-c will stop the bomb?\n");
    sleep(2);
    printf("Well...");
    fflush(stdout);
    sleep(1);
    printf("OK\n");
    exit(0);
  }
}

int main() {
  signal(SIGINT, handler); /* installs ctrl-c handler */
  while(1) {
    . . .
    signal(SIGINT, child_handler);
    . . .
  }
  return 0;
}
```
A Program That Reacts to Internally Generated Events

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

int beeps = 0;

/* SIGALRM handler */
void handler(int sig) {
    printf("BEEP\n");
    fflush(stdout);
    if (++beeps < 5)
       alarm(1);
    else {
       printf("BOOM!\n");
       exit(0);
    }
}

main() {
    signal(SIGALRM, handler);
    alarm(1); /* send SIGALRM in 1 second */

    while (1) {
        /* handler returns here */
    }
}

Nonlocal Jumps: set jmp/long jmp

Powerful (but dangerous) user-level mechanism for transferring control to an arbitrary location.

- Controlled to way to break the procedure call/return discipline
- Useful for error recovery and signal handling

```c
int setjmp(jmp_buf j)
```

- Must be called before longjmp()
- Identifies a return site for a subsequent longjmp().
- Called once, returns one or more times

**Implementation:**
- Remember where you are by storing the current register context, stack pointer, and PC value in jmp_buf.
- Return 0

```c
void longjmp(jmp_buf j, int i)
```

- Meaning:
  - return from the setjmp remembered by jump buffer `j` again...
  - ...this time returning `i` instead of 0
- Called after setjmp
- Called once, but never returns

**longjmp implementation:**
- Restore register context from jump buffer `j`
- Set `eax` (the return value) to `i`
- Jump to the location indicated by the PC stored in jump buffer `j`.

Limitations of Nonlocal Jumps

Works within stack discipline

- Can only long jump to environment of function that has been called but not yet completed
Putting It All Together: A Program That Restarts Itself When `ctrl-c`

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

sigjmp_buf buf;

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

main() {
    signal(SIGINT, handler);
    if (!sigsetjmp(buf, 1))
        printf("starting\n");
    else
        printf("restarting\n");

    while (1) {
        sleep(1);
        printf("processing...\n");
    }
}
```

bass> a.out

```
Ctrl-c
starting
processing...
processing...
restarting
processing...
processing...
restarting
processing...
Ctrl-c
```

Summary

**Signals provide 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
  - Use only for exceptional conditions
- Signals don't have queues
  - Just one bit for each pending signal type

**Nonlocal jumps provide exceptional control flow within process**
- Within constraints of stack discipline