Process Control

Signals

Instructor: Alan Beadle

Department of Computer Science
University of Rochester
Announcements

You can take your exam back once you have checked for grading errors

For integrity reasons, please bring up possible grading mistakes before you leave with your exam; Not after.

A4 is due in 11 days, and you should have started by now so that you have time to get help
Context Switching

- Processes are managed by a shared chunk of memory-resident OS code called the *kernel*
  - Important: the kernel is not a separate process, but rather runs as part of some existing process.
- Control flow passes from one process to another via a *context switch*
```c
int main()
{
    pid_t pid;
    int x = 1;

    pid = Fork();
    if (pid == 0) { /* Child */
        printf("child: x=%d\n", ++x);
        exit(0);
    }

    /* Parent */
    printf("parent: x=%d\n", --x);
    exit(0);
}
```

The diagram illustrates the process graph with the following steps:
- **Parent**: Starts with `main` and forks `fork`.
- **Child**: Begins with the fork and increments `x` before exiting.
- **Parent**: Decrements `x` before exiting.

`fork.c`
Interpreting Process Graphs

- Original graph:

  ![Original Graph Diagram]

- Abstracted graph:

  ![Abstracted Graph Diagram]
Interpreting Process Graphs

• Original graph:

```
child: x=2
  printf
  exit

x==1
parent: x=0
  main
  fork
  printf
  exit
```

• Abstracted graph:

```
  a
  b
  c
  d
  e
  f
```

Feasible execution ordering:

```
a  b  e  c  f  d
```
Interpreting Process Graphs

• Original graph:

• Abstracted graph:

Feasible execution ordering:

Infeasible execution ordering:
fork Example: Two consecutive forks

```c
void fork2()
{
    printf("L0\n");
    fork();
    printf("L1\n");
    fork();
    printf("Bye\n");
}
```

forks.c
Example: Two consecutive forks

```c
void fork2()
{
    printf("L0\n");
    fork();
    printf("L1\n");
    fork();
    printf("Bye\n");
}
```

forks.c
Example: Two consecutive `forks`

```c
void fork2()
{
    printf("L0\n");
    fork();
    printf("L1\n");
    fork();
    printf("Bye\n");
}  
```

Feasible output:
L0
L1
Bye
Bye
Bye
Bye
fork  Example: Two consecutive forks

```c
void fork2()
{
    printf("L0\n");
    fork();
    printf("L1\n");
    fork();
    printf("Bye\n");
}
```

Feasible output:
L0
L1
Bye
Bye
L1
Bye
Infeasible output:
L0
Bye
L1
Bye
L1
Bye
Bye
fork Example: Nested forks in parent

```c
void fork4()
{
    printf("L0\n");
    if (fork() != 0) {
        printf("L1\n");
        if (fork() != 0) {
            printf("L2\n");
        }
    }
    printf("Bye\n");
}
forks.c
```
fork Example: Nested forks in parent

```c
void fork4()
{
    printf("L0\n");
    if (fork() != 0) {
        printf("L1\n");
        if (fork() != 0) {
            printf("L2\n");
        }
    }
    printf("Bye\n");
}
```

forks.c
fork Example: Nested forks in parent

```c
void fork4()
{
    printf("L0\n");
    if (fork() != 0) {
        printf("L1\n");
        if (fork() != 0) {
            printf("L2\n");
        }
    }
    printf("Bye\n");
}
```

Feasible output:
L0
L1
Bye
Bye
L2
Bye

forks.c
Example: Nested `forks` in parent

```c
void fork4()
{
    printf("L0\n");
    if (fork() != 0) {
        printf("L1\n");
        if (fork() != 0) {
            printf("L2\n");
        }
    }
    printf("Bye\n");
}
```

Feasible output:
- `L0`
- `L1`
- `L2`
- `Bye`

Infeasible output:
- `L0`
- `Bye`
- `L1`
- `Bye`
- `L2`
- `Bye`
fork Example: Nested forks in children

```c
void fork5()
{
    printf("L0\n");
    if (fork() == 0) {
        printf("L1\n");
        if (fork() == 0) {
            printf("L2\n");
        }
    }
    printf("Bye\n");
}
forks.c
```
fork Example: Nested forks in children

```c
void fork5()
{
    printf("L0\n");
    if (fork() == 0) {
        printf("L1\n");
        if (fork() == 0) {
            printf("L2\n");
        }
    }
    printf("Bye\n");
}
```

forks.c
fork Example: Nested forks in children

```c
void fork5()
{
    printf("L0\n");
    if (fork() == 0) {
        printf("L1\n");
        if (fork() == 0) {
            printf("L2\n");
        }
    }
    printf("Bye\n");
}
```

Feasible output:
L0
Bye
L1
L2
Bye
Bye
forks.c
fork Example: Nested forks in children

```c
void fork5()
{
    printf("L0\n");
    if (fork() == 0) {
        printf("L1\n");
        if (fork() == 0) {
            printf("L2\n");
        }
    }
    printf("Bye\n");
}
```

Feasible output:

```
L0
Bye
L1
L2
Bye
Bye
```

Infeasible output:

```
L0
Bye
L1
Bye
Bye
L2
```
Reaping Child Processes

• When process terminates, it still consumes system resources
  • Examples: Exit status, various OS tables
  • Called a “zombie”: Living corpse, half alive and half dead

• Reaping
  • Performed by parent on terminated child (using `wait` or `waitpid`)**
Reaping Child Processes

- When process terminates, it still consumes system resources
  - Examples: Exit status, various OS tables
  - Called a “zombie”: Living corpse, half alive and half dead

- Reaping
  - Performed by parent on terminated child (using `wait` or `waitpid`)
  - Parent is given exit status information
  - Kernel then deletes zombie child process

- What if parent doesn’t reap?
  - If any parent terminates without reaping a child, then the orphaned child will be reaped by `init` process (pid == 1)
Reaping Child Processes

• When process terminates, it still consumes system resources
  • Examples: Exit status, various OS tables
  • Called a “zombie”: Living corpse, half alive and half dead

• Reaping
  • Performed by parent on terminated child (using \texttt{wait} or \texttt{waitpid})
  • Parent is given exit status information
  • Kernel then deletes zombie child process

• What if parent doesn’t reap?
  • If any parent terminates without reaping a child, then the orphaned child will be reaped by \texttt{init} process (pid == 1)
  • So, only need explicit reaping in long-running processes
    • e.g., shells and servers
void fork7() {
    if (fork() == 0) {
        /* Child */
        printf("Terminating Child, PID = %d\n", getpid());
        exit(0);
    } else {
        printf("Running Parent, PID = %d\n", getpid());
        while (1)
            ; /* Infinite loop */
    }
}
```c
void fork7() {
    if (fork() == 0) {
        /* Child */
        printf("Terminating Child, PID = %d\n", getpid());
        exit(0);
    } else {
        printf("Running Parent, PID = %d\n", getpid());
        while (1)
            ; /* Infinite loop */
    }
}
```

```bash
linux> ./forks 7 &
[1] 6639
Running Parent, PID = 6639
Terminating Child, PID = 6640
linux> ps
    PID TTY          TIME CMD
 6585 ttyp9    00:00:00 tcsh
 6639 ttyp9    00:00:03 forks
 6640 ttyp9    00:00:00 forks <defunct>
 6641 ttyp9    00:00:00 ps
linux> kill 6639
[1] Terminated
linux> ps
    PID TTY          TIME CMD
 6585 ttyp9    00:00:00 tcsh
 6642 ttyp9    00:00:00 ps
```
A program called `forks.c` demonstrates the concept of a zombie process in Unix. The program creates a child process, and then waits for the child to terminate. When the child process is killed before it terminates, the parent process remains in a zombie state.

The program outputs:

```c
void fork7() {
    if (fork() == 0) {
        /* Child */
        printf("Terminating Child, PID = \%d\n", getpid());
        exit(0);
    } else {
        printf("Running Parent, PID = \%d\n", getpid());
        while (1)
            ; /* Infinite loop */
    }
}
```

The `ps` command shows the child process as "defunct" (i.e., a zombie). For example:

```
linux> ./forks 7 &
[1] 6639
Running Parent, PID = 6639
Terminating Child, PID = 6640
linux> ps
    PID TTY          TIME CMD
6585 ttyp9    00:00:00 tcsh
6639 ttyp9    00:00:03 forks
6640 ttyp9    00:00:00 forks <defunct>
6641 ttyp9    00:00:00 ps
```

Additionally, the parent process can be killed to force the child process to become a zombie:

```
linux> kill 6639
[1] Terminated
```

After the parent is killed, the zombie child process is still visible in the `ps` output:

```
linux> ps
    PID TTY          TIME CMD
6585 ttyp9    00:00:00 tcsh
6642 ttyp9    00:00:00 ps
```
void fork7() {
    if (fork() == 0) {
        /* Child */
        printf("Terminating Child, PID = %d\n", getpid());
        exit(0);
    } else {
        printf("Running Parent, PID = %d\n", getpid());
        while (1)
            ; /* Infinite loop */
    }
}

forks.c

void fork7()
{
    if (fork() == 0) {
        /* Child */
        printf("Terminating Child, PID = %d\n", getpid());
        exit(0);
    } else {
        printf("Running Parent, PID = %d\n", getpid());
        while (1)
            ; /* Infinite loop */
    }
}

forks.c

linux> ./forks 7 &
[1] 6639
Running Parent, PID = 6639
Terminating Child, PID = 6640
linux> ps
    PID TTY          TIME CMD
 6585 ttyp9    00:00:00 tcsh
 6639 ttyp9    00:00:03 forks
 6640 ttyp9    00:00:00 forks <defunct>
 6641 ttyp9    00:00:00 ps
linux> kill 6639
[1] Terminated
linux> ps
    PID TTY          TIME CMD
 6585 ttyp9    00:00:00 tcsh
 6642 ttyp9    00:00:00 ps

- `ps` shows child process as “defunct” (i.e., a zombie)
- Killing parent allows child to be reaped by `init`
void fork8()
{
    if (fork() == 0) {
        /* Child */
        printf("Running Child, PID = %d\n", getpid());
        while (1)
            ; /* Infinite loop */
    } else {
        printf("Terminating Parent, PID = %d\n", getpid());
        exit(0);
    }
}
void fork8()
{
    if (fork() == 0) {
        /* Child */
        printf("Running Child, PID = %d\n", getpid);
        while (1)
            ; /* Infinite loop */
    } else {
        printf("Terminating Parent, PID = %d\n", getpid);
        exit(0);
    }
}

void fork8()
{
    if (fork() == 0) {
        /* Child */
        printf("Running Child, PID = %d\n", getpid);
        while (1)
            ; /* Infinite loop */
    } else {
        printf("Terminating Parent, PID = %d\n", getpid);
        exit(0);
    }
}

forks.c

linux> ./forks 8
Terminating Parent, PID = 6675
Running Child, PID = 6676

linux> ps
    PID  TTY          TIME  CMD
 6585  ttyp9    00:00:00  tcsh
 6676  ttyp9    00:00:06  forks
 6677  ttyp9    00:00:00  ps

linux> kill 6676

linux> ps
    PID  TTY          TIME  CMD
 6585  ttyp9    00:00:00  tcsh
 6678  ttyp9    00:00:00  ps
void fork8()
{
  if (fork() == 0) {
    /* Child */
    printf("Running Child, PID = %d\n", getpid());
    while (1)
      ; /* Infinite loop */
  } else {
    printf("Terminating Parent, PID = %d\n", getpid());
    exit(0);
  }
}

void forks.c

linux> ./forks 8
Terminating Parent, PID = 6675
Running Child, PID = 6676
linux> ps
   PID  TTY       TIME   CMD
 6585 tttyp9 00:00:00  tcsh
 6676 tttyp9 00:00:06  forks
 6677 tttyp9 00:00:00  ps
linux> kill 6676
linux> ps
   PID  TTY       TIME   CMD
 6585 tttyp9 00:00:00  tcsh
 6678 tttyp9 00:00:00  ps

- Child process still active even though parent has terminated. Can’t be reaped since it’s still running!
- Must kill child explicitly, or else will keep running indefinitely
wait: Synchronizing with Children

```c
void fork9() {
    int child_status;

    if (fork() == 0) {
        printf("HC: hello from child\n");
        exit(0);
    } else {
        printf("HP: hello from parent\n");
        wait(&child_status);
        printf("CT: child has terminated\n");
    }
    printf("Bye\n");
}
forks.c
```
**wait: Synchronizing with Children**

```c
void fork9() {
    int child_status;

    if (fork() == 0) {
        printf("HC: hello from child\n");
        exit(0);
    } else {
        printf("HP: hello from parent\n");
        wait(&child_status);
        printf("CT: child has terminated\n");
    }
    printf("Bye\n");
}
```

*forks.c*
wait: Synchronizing with Children

```c
void fork9() {
    int child_status;

    if (fork() == 0) {
        printf("HC: hello from child\n");
        exit(0);
    } else {
        printf("HP: hello from parent\n");
        wait(&child_status);
        printf("CT: child has terminated\n");
    }
    printf("Bye\n");
}
```

Feasible output:
HC
HP
CT
Bye
wait: Synchronizing with Children

```c
void fork9() {
    int child_status;

    if (fork() == 0) {
        printf("HC: hello from child\n");
        exit(0);
    } else {
        printf("HP: hello from parent\n");
        wait(&child_status);
        printf("CT: child has terminated\n");
    }
    printf("Bye\n");
}
```

Feasible output:  
HC  
HP  
CT  
Bye

Infeasible output:  
HP  
CT  
Bye  
HC
wait: Synchronizing with Children

- Parent reaps a child by calling the `wait` function

```c
int wait(int *child_status)
```

-Suspends current process until one of its children terminates
-Return value is the `pid` of the child process that terminated

-If `child_status` = `NULL`, then the integer it points to will be set to a value that indicates reason the child terminated and the exit status:
  - Checked using macros defined in `wait.h`
    - `WIFEXITED`, `WEXITSTATUS`, `WIFSIGNALED`, `WTERMSIG`, `WIFSTOPPED`, `WSTOPSIG`, `WIFCONTINUED`
  - See textbook for details
Another `wait` Example

- If multiple children completed, will take in arbitrary order
- Can use macros WIFEXITED and WEXITSTATUS to get information about exit status

```c
callback void fork10() {
    int i, child_status;

    for (i = 0; i < N; i++)
        if (fork() == 0) {
            exit(100+i); /* Child */
        }

    for (i = 0; i < N; i++) {
        pid_t 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 terminate abnormally\n", wpid);
    }
}
forks.c
```
**waitpid: Waiting for a Specific Process**

- `pid_t waitpid(pid_t pid, int &status, int options)`
  - Suspends current process until specific process terminates
  - Various options (see textbook)

```c
void fork11() {
    pid_t pid[N];
    int i;
    int child_status;

    for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == 0)
            exit(100+i); /* Child */
    for (i = N-1; i >= 0; i--)
        {
            pid_t wpid = waitpid(pid[i], &child_status, 0);
            if (WIFEXITED(child_status))
                printf("Child %d terminated with exit status %d\n", wpid, WEXITSTATUS(child_status));
            else
                printf("Child %d terminate abnormally\n", wpid);
        }
}
```
execute: Loading and Running Programs

Executes "/bin/ls -lt /usr/include" in child process using current environment:

```c
char *myargv[] = {"/bin/ls", "-lt", "/usr/include"};
char *environ[] = {"USER=droh", "PWD="/usr/droh"};

if ((pid = Fork()) == 0) {
    /* Child runs program */
    if (execve(myargv[0], myargv, environ) < 0) {
        printf("%s: Command not found.\n", myargv[0]);
        exit(1);
    }
}
```
execve: Loading and Running Programs

- `int execve(char *filename, char *argv[], char *envp[])`
execve: Loading and Running Programs

- `int execve(char *filename, char *argv[], char *envp[])`

- Loads and runs in the current process:
  - Executable file `filename`
  - Argument list `argv`
    - By convention `argv[0]==filename`
  - Environment variable list `envp`
    - “name=value” strings (e.g., `USER=droh`)
execve: Loading and Running Programs

- `int execve(char *filename, char *argv[], char *envp[])`
- Loads and runs in the current process:
  - Executable file `filename`
  - Argument list `argv`
    - By convention `argv[0]==filename`
  - Environment variable list `envp`
    - “name=value” strings (e.g., `USER=droh`)
- Overwrites code, data, and stack
  - Retains PID, open files and signal context
execve: Loading and Running Programs

- int execve(char *filename, char *argv[], char *envp[])
- Loads and runs in the current process:
  - Executable file filename
  - Argument list argv
    - By convention argv[0] == filename
  - Environment variable list envp
    - "name=value" strings (e.g., USER=droh)
- Overwrites code, data, and stack
  - Retains PID, open files and signal context
- Called once and never returns
  - ...except if there is an error
execve Example

Executes "/bin/ls -lt /usr/include" in child process using current environment:

```c
if ((pid = Fork()) == 0) { /* Child runs program */
    if (execve(myargv[0], myargv, environ) < 0) {
        printf("%s: Command not found.\n", myargv[0]);
        exit(1);
    }
}
```
Summary

• Processes
  • At any given time, system has multiple active processes
  • Only one can execute at a time on a single core, though
  • Each process appears to have total control of processor + private memory space

• Spawning processes
  • Call fork
  • One call, two returns

• Process completion
  • Call exit
  • One call, no return

• Reaping and waiting for processes
  • Call wait or waitpid

• Loading and running programs
  • Call execve (or variant)
  • One call, (normally) no return
Today

• Process Control

• Signals: The Way to Communicate with Processes
Signals

• A **signal** is a small message that notifies a process that an event of some type has occurred in the system
  - Sent from the **OS kernel**
  - Could be requested by another process, by user, or automatically by the kernel
  - Signal type is identified by small integer ID’s (1-30)
Signals

- A **signal** is a small message that notifies a process that an event of some type has occurred in the system
  - Sent from the **OS kernel**
  - Could be requested by another process, by user, or automatically by the kernel
  - Signal type is identified by small integer ID’s (1-30)

<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>Terminate</td>
<td>User typed ctrl-c</td>
</tr>
<tr>
<td>9</td>
<td>SIGKILL</td>
<td>Terminate</td>
<td>Kill program (cannot override or ignore)</td>
</tr>
<tr>
<td>11</td>
<td>SIGSEGV</td>
<td>Terminate</td>
<td>Segmentation violation</td>
</tr>
<tr>
<td>14</td>
<td>SIGALRM</td>
<td>Terminate</td>
<td>Timer signal</td>
</tr>
<tr>
<td>17</td>
<td>SIGCHLD</td>
<td>Ignore</td>
<td>Child stopped or terminated</td>
</tr>
</tbody>
</table>
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:
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:
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:
    • Exception: divide-by-zero (SIGFPE)
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:
    - Exception: divide-by-zero (SIGFPE)
    - Interrupt: user pressing Ctrl + C (SIGINT)
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:
    • Exception: divide-by-zero (SIGFPE)
    • Interrupt: user pressing Ctrl + C (SIGINT)
    • The termination of a child process (SIGCHLD)
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:
    • Exception: divide-by-zero (SIGFPE)
    • Interrupt: user pressing Ctrl + C (SIGINT)
    • 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: 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:
    - Exception: divide-by-zero (SIGFPE)
    - Interrupt: user pressing Ctrl + C (SIGINT)
    - 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.
    - Note: `kill` doesn’t mean you are going to kill the target process. It is just a system call that allows you to send signals. Of course the signal you send could be SIGKILL.
Signal Concepts: 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

• Some possible ways to react:
Signal Concepts: 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

• Some possible ways to react:
  • Ignore the signal (do nothing)
Signal Concepts: 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

• Some possible ways to react:
  • Ignore the signal (do nothing)
  • Terminate the process
Signal Concepts: 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

• Some possible ways to react:
  • Ignore the signal (do nothing)
  • Terminate the process
  • Catch the signal by executing a user-level function called signal handler
Signal Concepts: 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.

• Some possible ways to react:
  • Ignore the signal (do nothing)
  • Terminate the process
  • Catch the signal by executing a user-level function called signal handler
    • Similar to a hardware exception handler being called in response to an asynchronous interrupt:
Signal Concepts: 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.

• Some possible ways to react:
  • Ignore the signal (do nothing)
  • Terminate the process
  • Catch the signal by executing a user-level function called **signal handler**
    • Similar to a hardware exception handler being called in response to an asynchronous interrupt:

(1) Signal received by process

(2) Control passes to signal handler

(3) Signal handler runs

(4) Signal handler returns to next instruction
Sending Signals with `/bin/kill` Program

- `/bin/kill` program sends arbitrary signal to a process

Examples

- `/bin/kill -9 24818`
  Send SIGKILL to process 24818
- `/bin/kill` itself doesn’t kill the process. 9 is the ID for the SIGKILL signal, which terminates the process

```bash
linux> ./forks 16
Child1: pid=24818 pgrp=24817
Child2: pid=24819 pgrp=24817

linux> ps
  PID TTY          TIME CMD
24788 pts/2    00:00:00 tcsh
24818 pts/2    00:00:02 forks
24819 pts/2    00:00:02 forks
24820 pts/2    00:00:00 ps
```
Sending Signals with `/bin/kill` Program

- `/bin/kill` program sends arbitrary signal to a process

**Examples**
- `/bin/kill -9 24818`
  Send SIGKILL to process 24818
- `/bin/kill` itself doesn’t kill the process. 9 is the ID for the SIGKILL signal, which terminates the process

```
linux> ./forks 16
Child1: pid=24818 pgrp=24817
Child2: pid=24819 pgrp=24817

linux> ps
     PID  TTY   TIME CMD
24788 pts/2  00:00:00 tcsh
24818 pts/2  00:00:02 forks
24819 pts/2  00:00:02 forks
24820 pts/2  00:00:00 ps
```
Process Groups

- Every process belongs to exactly one process group
Process Groups

• Every process belongs to exactly one process group

getpgrp()
Return process group of current process

setpgid()
Change process group of a process
Sending Signals with \( /bin/kill \) Program

- \( /bin/kill \) program sends arbitrary signal to a process or process group

**Examples**
- \( /bin/kill \ -9 \ -24817 \)
  Send SIGKILL to every process in process group 24817

```bash
linux> ./forks 16
Child1: pid=24818 pgrp=24817
Child2: pid=24819 pgrp=24817

linux> ps
   PID TTY          TIME CMD
24788 pts/2    00:00:00 tcsh
24818 pts/2    00:00:02 forks
24819 pts/2    00:00:02 forks
24820 pts/2    00:00:00 ps

linux> /bin/kill -9 -24817

linux> ps
   PID TTY          TIME CMD
24788 pts/2    00:00:00 tcsh
24823 pts/2    00:00:00 ps
```

```bash
linux> 
```
Sending Signals with `/bin/kill` Program

• `/bin/kill` program sends arbitrary signal to a process or process group

• Examples
  • `/bin/kill -9 -24817`
    Send SIGKILL to every process in process group 24817
Sending Signals from the Keyboard

• Typing ctrl-c causes the kernel to send a SIGINT to every process in the foreground process group.
  • SIGINT – default action is to terminate each process

• Typing ctrl-z causes the kernel to send a SIGTSTP to every job in the foreground process group.
  • SIGTSTP – default action is to stop (suspend) each process
Example of `ctrl-c` and `ctrl-z`

```
bluefish> ./forks 17
Child: pid=28108 pgrp=28107
Parent: pid=28107 pgrp=28107

<types ctrl-z>
Suspended
```

```
bluefish> ps w

  PID   TTY      STAT   TIME   COMMAND
27699 pts/8    Ss     0:00   -tcsh
28107 pts/8    T       0:01   ./forks 17
28108 pts/8    T       0:01   ./forks 17
28109 pts/8    R+      0:00   ps w

```

```
bluefish> fg
./forks 17
<types ctrl-c>
```

```
bluefish> ps w

  PID   TTY      STAT   TIME   COMMAND
27699 pts/8    Ss     0:00   -tcsh
28110 pts/8    R+      0:00   ps w
```

STAT (process state)
Legend:

**First letter:**
S: sleeping
T: stopped
R: running

**Second letter:**
s: session leader
+: foreground proc group

See “man ps” for more details
Sending Signals with `kill` Function

```c
void fork12()
{
    pid_t pid[N];
    int i;
    int child_status;

    for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == 0) {
            /* Child: Infinite Loop */
            while(1)
        }

    for (i = 0; i < N; i++) {
        printf("Killing process %d\n", pid[i]);
        kill(pid[i], SIGINT);
    }

    for (i = 0; i < N; i++) {
        pid_t 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);
    }
}
```

`forks.c`