Lecture Notes for Lecture 8 of CS 5600 (Computer Systems) for the Fall 2019 session at the Northeastern University Silicon Valley Campus.

Processes and Process Control

Philip Gust,
Clinical Instructor
Khoury College of Computer and Information Science

Lecture 7 Review

- In lecture 7, we continued to explore memory management by looking in more detail at how heap storage is managed and look at how the original C dynamic memory allocator was designed.
- As we saw, there are several issues that memory allocators must address. How they address these issues depends on assumptions about allocation patterns and memory usage.
- We also considered ways of optimizing how memory is managed using several advanced techniques for representing the free list and recording additional information.

Processes and Process Control

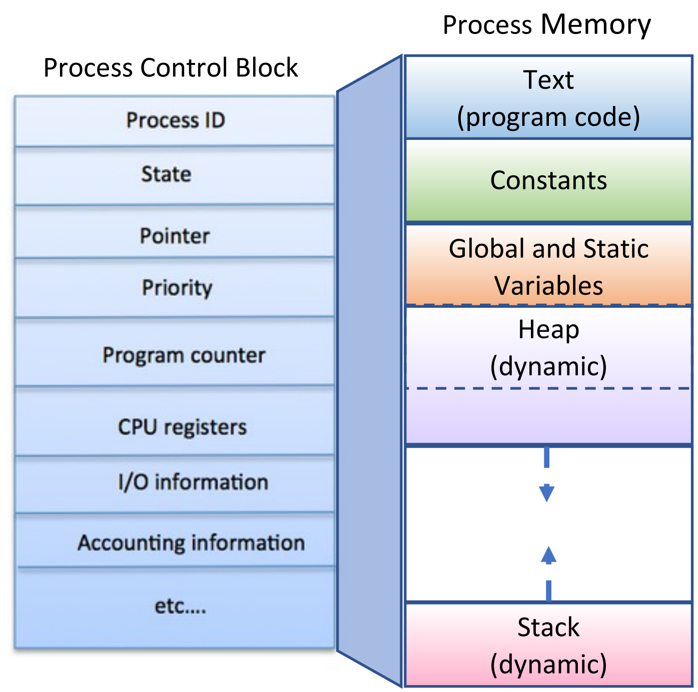
Overview

- In this lecture, we will study one of the primary resources managed by an operating system. A *process* is an instance of a program that has been loaded into memory and is being executed by a computer.
- The process contains the code and data that are defined by the program, as well as the running state of the program, and information about other resources being used by the program.
- The role of a process is to provide a virtual computer that enables a program to operate as though it has the sole use of the computer and its resources.
- In reality, the operating system and its kernel carefully manage many processes and mediate access to the resources shared by all of them.

Processes and Process Control

Process structure and resources

- The process structure reflects the structure of a program, plus resources used while running.
  - Memory segments including code, constants, globals, heap and stack
  - Processor state
  - Registers
  - System information about resources allocated to process
- The information is stored a kernel data structure known as a *process control block (PCB)*.
- Each process is identified by a unique *process id*.



Processes and Process Control

Process structure and resources

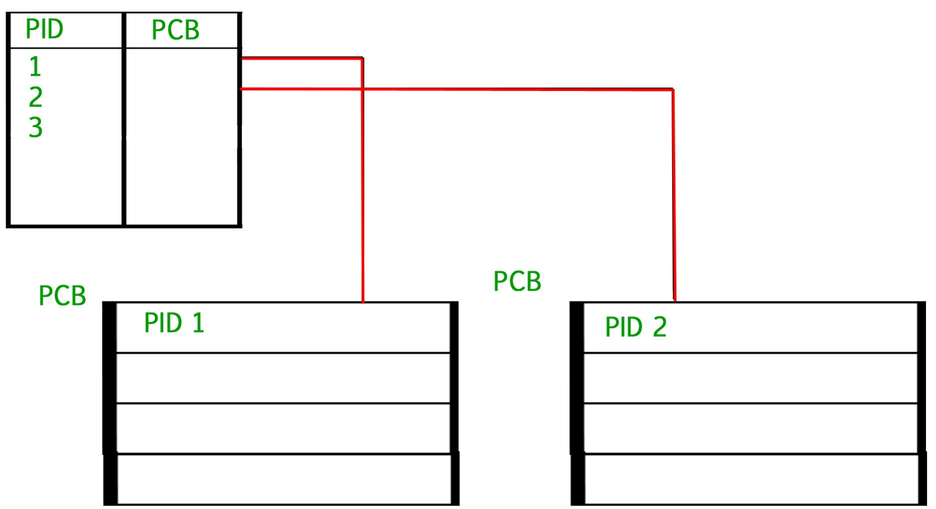
- The information in the PCB includes

Information	Description
Process ID	ID of the process
Process State	Current state, i.e. ready, running, etc.
Pointer	Pointer to parent process
Priority	Process priority and other info.
Program Counter	Program counter
CPU Registers	Saved CPU registers
I/O Information	List of I/O devices, state, open file lists
Accounting Information	Includes CPU for process execution, time limits, etc.

Processes and Process Control

Process structure and resources

- The kernel maintains pointers to each process's PCB in a process table so that it can access the PCB quickly.



Process table and process control block

Processes and Process Control

Creating a processes

- When an operating system is booted, typically several processes are created.
- Some of these are foreground processes, that interacts with a (human) user and perform work for them.
- Other are background processes, which are not associated with particular users, but instead have some specific function.
- For example, one background process may be designed to accept incoming e-mails, sleeping most of the day but suddenly springing to life when an incoming e-mail arrives.

Processes and Process Control

Creating a processes

- Users need a way to create a process from a command shell.
- The shell is given the name of a program to run and optional arguments, the shell creates a new process, and connects it with the keyboard and mouse.
- The shell then runs the process and waits for the process to complete. The new process is a child process of the shell, and the shell waits for the process to terminate.

Processes and Process Control

Creating a processes

- It is also possible to start a process that is not a child process of the shell. Instead, it is a background process.
- To start a program in the background, place an ampersand (&) character after the command.
- For example, here is how to write a command to find all text files in the current directory hierarchy, and send the output to a file as a background task:

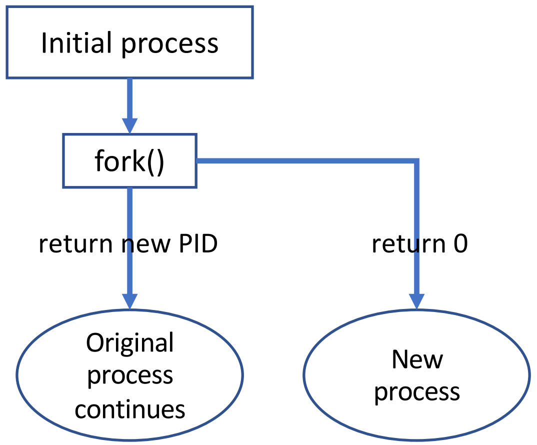
```
ls *.txt  > /tmp/textfiles.txt &
```

- A set of *job control* commands including *fg*, *bg*, and *jobs*, are available for managing background processes and even moving them to the foreground. See the builtin bash man page for details.

Processes and Process Control

Creating a processes

- Developers can create a process by calling the *fork()* system call from a C or C++ program.
- The *fork()* function creates a second process that is a duplicate of the current process, running the same program.
- The *fork()* function returns the new process ID to the original process, and 0 to the new process.



Processes and Process Control

Creating a processes

- Here is an example of how fork() and wait() work together.

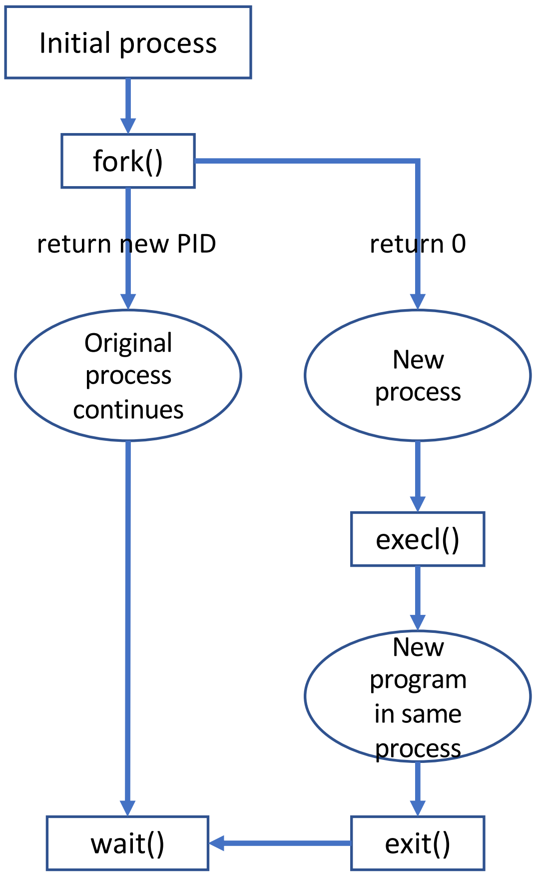
```
// C program to demonstrate working of wait()
#include <stdlib.h>
#include <stdio.h>
#include <sys/wait.h>
#include <unistd.h>

int main() {
   if (fork()== 0) {
      printf("HC: hello from child\n");
   } else {
      printf("HP: hello from parent\n");
      wait(NULL);   // for child
      printf("CT: child has terminated\n");
   }

   printf("Bye\n");
   return 0;
}
```

```
HC: hello from child
Bye
HP: hello from parent
CT: child has terminated
Bye
                 (or)
HP: hello from parent
HC: hello from child
Bye
CT: child has terminated        // this sentence does
Bye                             // not print before HC
                                // because of wait.
```

Processes and Process Control

Creating a processes

- The fork() function can be combined with two other system calls to launch a program in the new process and have the original process wait for it to complete.
- The execl() function replaces the currently running program in its process, and the wait() function waits for child process to exit before continuing.



Processes and Process Control

Creating a processes

- Here is an example of how fork() and exec() work together.

```
// C program to demonstrate working of exec()
#include <stdlib.h>
#include <stdio.h>
#include <sys/wait.h>
#include <unistd.h>

int main(void) {
    pid_t child_pid = fork();
    if (child_pid == 0) {
        printf("HC: hello from child\n");

        // exec 'ls' from PATH
        execlp("ls", "ls", "/usr/bin", (char*)0);

        // exec must have failed
        printf("Unknown command\n");
        exit(0);
    } else {
        printf("HP: hello from parent\n");
        pid_t t_pid = wait(NULL);   // for child
        printf("CT: child has terminated\n");
    }

    printf("Bye\n");
    return 0;
}
```

Processes and Process Control

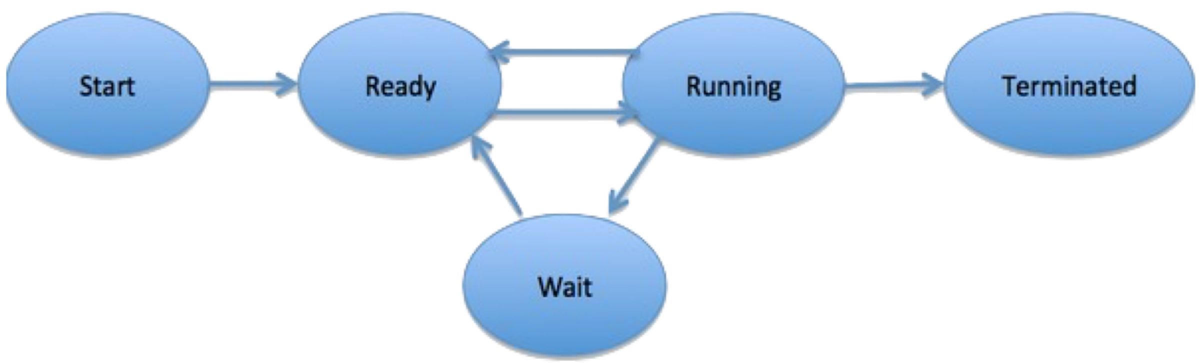
Process lifecycle

- A process has a *lifecycle* that is managed by the kernel. This is because multiple processes need to share the limited resources of the operating system.
- The kernel has a resource manager called a *scheduler* that ensures each process gets a slice of time to work, and then gives another waiting process a chance to run.
- Processes also naturally need to wait content from a user, the disk or the network. During those intervals, the scheduler can swap in another process that that is ready to run for an interval.

Processes and Process Control

Process lifecycle

- Here is a description and illustration of a typical process lifecycle.
  - **Start**: This is the initial state when a process is first started/created.
  - **Ready**: The process is waiting to be assigned to a processor.
  - **Running**: Once the process has been assigned to a processor, it has this state.
  - **Waiting**: The process moves into waiting state if waiting for a resource.
  - **Terminated**: Once the process finishes or terminates it moves to this state.



Processes and Process Control

Managing running processes

- Once a process has been created, there are several POSIX utility commands for viewing and managing processes.
- The *ps* command enables users to view processes that are running and their current status. By default, ps displays a compact listing of just the current user's processes. See the man page for options.

```
MacBook-Pro-2:phil $ ps
  PID   TTY       TIME      CMD
  431   ttys000   0:00.50   -bash
  444   ttys001   0:01.89   -bash
39682   ttys001   0:00.03   vi test.sh
52773   ttys002   0:00.36   -bash
```

- The PID is the process id, and TTY is the terminal where bash and other programs are running.

Processes and Process Control

Managing running processes

- The *top* command periodically displays a sorted list of system processes. The default sorting key is pid, but other keys can be used instead. Various output options are available.

Processes: 570 total, 5 running, 1 stuck, 564 sleeping, 2617 threads   03:10:53
Load Avg: 7.00, 6.18, 5.23 CPU usage: 37.16% user, 4.5% sys, 58.78% idle
SharedLibs: 229M resident, 50M data, 59M linkedit.
MemRegions: 489061 total, 5715M resident, 156M private, 1383M shared.
PhysMem: 16G used (2953M wired), 16M unused.
VM: 22T vsize, 1116M framework vsize, 88367387(384) swapins, 90699887(0) swapout
Networks: packets: 25143514/11G in, 12987304/7330M out.
Disks: 17476213/676G read, 4336666/507G written.

PID	COMMAND	%CPU	TIME	#TH	#WQ	#PORT	MEM	PURG	CMPRS	PGRP
52773	bash	0.0	00:00.39	1	0	19	524K	0B	464K	52773
52772	login	0.0	00:00.37	2	1	30	12K	0B	11M	52772
52714	eclipse	0.8	69:05.17	80	3	644	236M-	12K	971M+	45910
50858	com.apple.We	0.0	01:00.90	5	1	405	13M	0B	126M	50858
48274	VTDecoderXPC	0.0	00:00.23	2	1	47	28K	0B	17M	48274
48010	CloudKeychai	0.0	00:00.16	2	1	46	712K	0B	1580K	48010
47991	dmd	0.0	00:00.07	2	1	31	28K	0B	1572K	47991

Processes and Process Control

Process Tree

- Every process except process 0 is created when another process executes the fork() system call. The process that invoked fork is the parent process and the newly created process is the child process.
- Every process (except process 0) has one parent process, but can have many child processes. Process 0 is a special process that is created when the system boots.
- After forking a child process (process 1), process 0 becomes the"idle task"). Process 1, known as init, is the ancestor of every other process in the system.

Processes and Process Control

Process Tree

- The set of parent and child processes is known as the *process tree*. Versions of the ps utility sometimes include ways to view this tree. The ‘-f’ option shows the parent PID (PPID) for each process.

MacBook-Pro-2:assignment-2 pgust bash phil$ ps -f

UID	PID	PPID	C	STIME	TTY	TIME	CMD
501	431	426	0	Sat08PM	ttys000	0:00.50	-bash
501	444	443	0	Sat08PM	ttys001	0:01.94	-bash
501	40567	444	0	3:51AM	ttys001	0:00.03	vi
501	52773	52772	0	Tue03PM	ttys002	0:00.55	-bash

- In this illustration, *bash* process 444 is the parent of the *vi* process 40567.

Processes and Process Control

Process Tree

- The pstree program, available on linux systems and also via brew on MacOS, can display the process tree of a process. A portable awk-based version is available at: https://www.serice.net/pstree/.
- For example, here is the process tree for the vi process.

```
% pstree -p 40567
-+= 00001 root /sbin/launchd
 \-+= 00319 phil /Applications/Utilities/Terminal.app/Contents/MacOS/Terminal -psn_0_57358
   \-+= 00431 root login -pf phil
     \-+= 444 phil -bash
       \--= 40567 phil vi
```

Processes and Process Control

Process Priorities

- Every process has a priority that determines how much time the process spends running compared to waiting.
- User processes have a default priority. It is possible for an administrator to modify the priority of a process using the *renice* command.
- It is also possible to view process priorities by adding the –l (lowercase L) option to *ps*.

Processes and Process Control

Signaling Processes

- The operating communicates with a process using a facility called signals.
- Signals may be generated by the operating system as a result of a low-level condition is of interest to a given process.
- Signals may also be generated by one running process and directed at another running process as a way to communicate with or to control the target process.
- A running process may also send a signal to itself through the operating system as a way for the process to modify or control its own execution.

Signals

Signaling Processes

- Some common signals (see /usr/include/sys/signal.h)

Signal	Value	Description
SIGHUP	1	Hangup (POSIX). Report that user's terminal is disconnected. Signal used to report the termination of the controlling process.
SIGINT	2	Interrupt (ANSI). Program interrupt. (ctrl-c)
SIGUSR1	10	User-defined signal 1
SIGUSR2	12	User-defined signal 2
SIGALRM	14	Alarm clock (POSIX). Indicates expiration of a timer. Used by the alarm() function.
SIGTERM	15	Termination (ANSI). This signal can be blocked, handled, and ignored. Generated by "kill" command.
SIGCONT	18	Continue (POSIX). Signal sent to process to make it continue.
SIGTSTP	20	Keyboard stop (POSIX). Interactive stop signal. This signal can be handled and ignored. (ctrl-z)

Signals

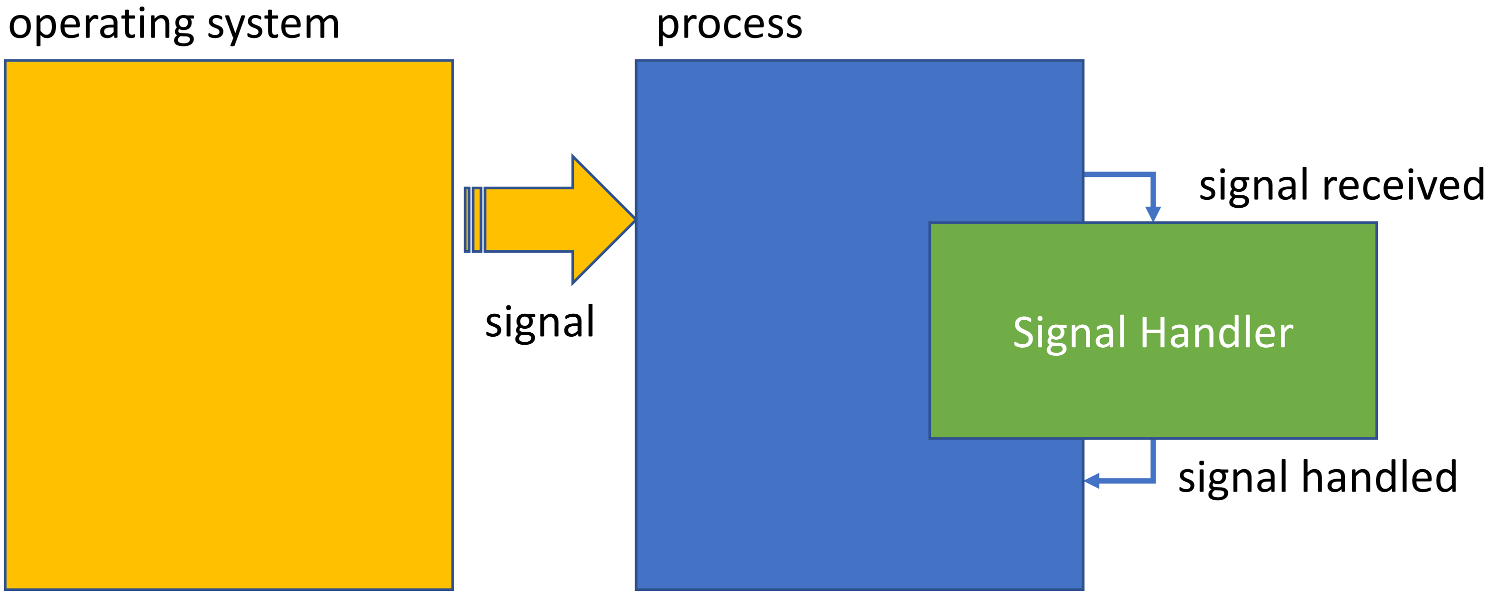
Signaling Processes

- When a signal is sent, the operating system interrupts the target process' normal flow of execution to deliver the signal.
- A signal can interrupt execution of a process during any *non-atomic* instruction, such as normal code execution or during I/O operations.
- If the target process has registered a *signal handler*, that routine is executed. Otherwise, the default signal handler for that signal executed.
- When the signal handler has completed, normal execution of the program resumes, or the process may terminate.

Signals

What are Signals?

- Handling a signal from the operating system.



Signals

What is a Signal Handler?

- A signal handler is a special function that is registered to handle a signal. The void function has a single integer parameter.
- Here is a typedef for a signal handler:

```
typedef void (*sig_t) (int);
```

Signals

What is a Signal Handler?

- You can define a single signal handler for the entire application, or specialized signal handlers for each type of signal.
- When the operating calls the signal handler, it passes the number corresponding to the signal being sent.
- Here is a signal handler that handles SIGTERM, the signal requesting to terminate a program a program normally.

```
/**
 * Handle SIGTERM as request to terminate the process normally.
 */
void SIGTERM_handler(int sig) {
    // code to handle the signal
}
```

Signals

Handling Signals

- In C, signal handlers can be installed with the *signal* or *sigaction* system call.
- If a signal handler is not installed for a particular signal, the default handler is used. Otherwise the signal is intercepted and the signal handler is invoked.
- Here is how to install our SIGTERM_handler signal handler:

  ```
  signal(SIGTERM, SIGTERM_handler);
  ```
- Now when we ask to shut down a program normally, this signal handler will be called, and can either cause the program to terminate, or do something else instead.

Processes and Process Control

Handling Signals

- Two special signal handlers are pre-defined that can also be used with *signal*():
  - SIG_DFL resets the default action for that signal
  - SIG_IGN causes future occurrences of the signal to be ignored and pending instance to be discarded
- There are two signals which cannot be intercepted and handled:
  - SIGKILL: Kill (9), unblockable (POSIX): cause immediate program termination.
  - SIGSTOP: Stop (17), unblockable (POSIX) Stop a process. This signal cannot be handled, ignored, or blocked.

Processes and Process Control

Handling Signals

- Signal handlers should be written in a way that does not result in any unwanted side-effects.
- Use of *non-reentrant* functions (e.g. functions that use static storage) inside signal handlers is also unsafe.
- In particular, the POSIX specification requires that all system functions directly or indirectly called from a signal function are async-signal safe (e.g. the system calls).
- Signal handlers can instead put the signal into a queue and immediately return.
- The main thread will then continue "uninterrupted" until signals are taken from the queue, such as in an event loop.

Processes and Process Control

Handling Signals

- Here is how a process can send a SIGALRM signal to itself in 1 second:

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

/** waiting for alarm */
bool waiting = true;

/** Handler for SIGALRM */
void SIGALRM_handler(int sig) {
    printf("time is up!\n");
    waiting = false;
}
```

```
int main() {
    // register SIGALRM handler
    signal(SIGALRM, SIGALRM_handler);

    // set up 1-second alarm
    alarm(1);

    // wait for alarm to go off
    while(waiting) {
        printf("waiting…\n");
    }
    printf("done waiting\n");
    return EXIT_SUCCESS;
}
```

Processes and Process Control

Sending Signals

- The *kill* system call in C sends a specified signal to a specified process, if permissions allow.

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

/** Handler for SIGUSR1 signal */
void SIGUSR1_handler(int sig) {
    printf("received SIGUSR1\n");
}
```

```
int main() {
    // register the SIGUSR1 handler
    signal(SIGUSR1, SIGUSR1_handler);

    // get id of this process
    pid_t pid = getpid();

    // send SIGUSR1 signal
    printf("sending SIGUSR1\n");
    kill(pid, SIGUSR1); // or raise(SIGUSR1)
    printf("sent SIGUSR1\n");

    return EXIT_SUCCESS;
}
```

Processes and Process Control

Sending Signals

- Similarly, the *kill* system command allows a user to send signals to processes.
- First get a process ID using the ps system command:

```
$ ps
 PID TTY           TIME CMD
 1229 ttys024    0:02.45 -bash
13953 ttys024    0:03.89 node server.js
10377 ttys026    6:39.91 mongod --noauth
 7029 ttys027    0:00.05 -bash
13911 ttys027    0:01.32 mongo
12034 ttys028    0:00.61 –bash
     …
```

Processes and Process Control

Sending Signals

- Next send termination signal number or name to the process:

```
kill -15 12034
kill –s TERM 12034
```

- If a process is “hung”, it can be ”killed” using SIGKILL(9), which cannot be caught or ignored by the process:

```
kill -9 12034
kill –s KILL 12034
```

Processes and Process Control

Sending Signals

- Here is an example of using signals to control a process.

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

/** define speeds */
enum {FAST = 0, SLOW = 1} speed = FAST;

/** in SIGTERM handler flag */
bool do_quit = false;
```

Processes and Process Control

Sending Signals

- Here is an example of using signals to control a process.

```
/** Handle SIGTERM as request to terminate the process normally.
 */
void SIGTERM_handler(int sig) {
    do_quit = true;
}

/** Handle SIGINT to change its repeat rate
 */
void SIGINT_handler(int sig) {
    speed = (speed == FAST) ? SLOW : FAST;
}
```

Processes and Process Control

Sending Signals

- Here is an example of using signals to control a process.

```
/** Setup and process signals.
 */
int main() {
    static const int speeds[] = {1, 5};
    static const char *speed_name[] = {"fast", "slow"};
    setlinebuf(stdout);  // set line buffering for stdout

    // Register signal and signal handler
    signal(SIGTERM, SIGTERM_handler);
    signal(SIGINT,  SIGINT_handler);

    // The process id
    pid_t pid = getpid();
```

Processes and Process Control

Sending Signals

- Here is an example of using signals to control a process.

```
while(true) {
    if (do_quit) { // handle request to quit
        printf("Do you really want to quit (y/n)? ");
        fflush(stdout);
        char linebuf[100];
        if (*fgets(linebuf, 100, stdin) == 'y') {
            printf("Exiting.\n");
            return EXIT_SUCCESS;
        }
        printf("Continuing.\n");
        do_quit = false;
    }
```

Processes and Process Control

Sending Signals

- Here is an example of using signals to control a process.

```
            printf("%s speed for process id %d\n", speed_name[speed], pid);
            sleep(speeds[speed]);  // Interruptible
        }
    }
```

Processes and Process Control

Sending Signals

- Typing certain key combinations at the controlling terminal of a running process cause the system to send it certain signals. These can be changed with the stty command.
  - Ctrl-C sends an INT signal ("interrupt", SIGINT); by default, this causes the process to terminate.
  - Ctrl-Z sends a TSTP signal ("terminal stop", SIGTSTP); by default, this causes the process to suspend execution.
  - Ctrl-\ sends a QUIT signal (SIGQUIT); by default, this causes the process to terminate and dump core.
  - Ctrl-T (not universally supported) sends an INFO signal (SIGINFO); by default, and if supported by the command, this causes the operating system to show information about the running command.

Processes and Process Control

Handling Signals in Bash

- Bash also provides a way to handle signals in scripts that is very similar to signal handling in C.
- The *trap* command specifies a command for a given signal. This script catches and ignores the SIGINT signal. It loops until it is externally terminated.

```
#!/bin/bash

# This script traps and ignores the INT signal
# and loops until it is externally terminated

#trap INT signal (CNTL-C)
trap 'echo " INT ignored" ' INT

# loop and print process id until terminated
while /usr/bin/true ; do
     echo $$   # process id
     sleep 1
done
```

Processes and Process Control

Handling Signals in Bash

- The action for the trap can be either an inline command or the name of a bash function

```
#!/bin/bash

# This script traps and ignores the INT signal
# and loops until it is externally terminated

# Handler for INT signal
INT_handler() {
     echo "INT ignored"
}

#trap INT signal (CNTL-C)
trap 'INT_handler' INT

# loop and print process id until terminated
while /usr/bin/true ; do
     echo $$   # process id
     sleep 1
done
```