Week 10 Tutorial Answers

1. How is the assignment going?

   Does anyone have hints or advice for other students?

   Has anyone discovered interesting cases that have to be handled?

   Answer:

   Discussed in tutorial.

2. Write a C program, `print_diary.c`, which prints the contents of the file `$HOME/.diary` to stdout

   The lecture example getstatus.c shows how to get the value of an environment variable.

   snprintf is a convenient fucntion for constructing the pathname of the diary file.

   Answer:

```c
// print $HOME/.diary to stdout

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(int argc, char *argv[]) {
    char *home_pathname = getenv("HOME");
    if (home_pathname == NULL) {
        home_pathname = ".";
    }

    char *basename = ".diary";
    int diary_pathname_len = strlen(home_pathname) + strlen(basename) + 2;
    char diary_pathname[diary_pathname_len];
    snprintf(diary_pathname, sizeof diary_pathname, "%s/%s", home_pathname, basename);

    FILE *stream = fopen(diary_pathname, "r");
    if (stream == NULL) {
        perror(diary_pathname);
        return 1;
    }

    int byte;
    while ((byte = fgetc(stream)) != EOF) {
        fputc(byte, stdout);
    }
    fclose(stream);

    return 0;
}
```

3. Write a C program, `print_file_bits.c`, which given as a command line arguments the name of a file contain 32-bit hexadecimal numbers, one per line, prints the low (least significant) bytes of each number as a signed decimal number (-128..127).

   Answer:

```
// read 32-byte hexadecimal numbers from a file
// and print low (least significant) byte
// as a signed decimal number (-128..127)

#include <stdio.h>
#include <stdint.h>

int main(int argc, char *argv[]) {
    if (argc != 2) {
        fprintf(stderr, "Usage: %s <file>\n", argv[0]);
        return 1;
    }


    FILE *stream = fopen(argv[1], "r");
    if (stream == NULL) {
        perror(argv[1]);
        return 1;
    }

    int32_t number;
    while (fscanf(stream, "%x", &number) == 1) {

        // convert low byte to a signed number
        // simple assignment to a int8_t variable works on most platforms
        // but is not defined by the C standard

        int32_t low_byte = number & 0xff;
        if (low_byte & 1 << 7) {
            low_byte = -(1 << 8) + low_byte;
        }

        printf("%d\n", low_byte);
    }
    fclose(stream);

    return 0;
}
```

4. Assume we have 6 virtual memory pages and 4 physical memory pages and are using a least-recently-used (LRU) replacement strategy.
   What will happen if these virtualmemory pages were accessed?

```
5 3 5 3 0 1 2 2 3 5
```

Answer:

Courtesy the lab exercise:

```
$ dcc lru.c -o lru
$ ./lru
Simulating 4 pages of physical memory, 6 pages of virtual memory
5
Time 0: virtual page 5 loaded to physical page 0
3
Time 1: virtual page 3 loaded to physical page 1
5
Time 2: virtual page 5 -> physical page 0
3
Time 3: virtual page 3 -> physical page 1
0
Time 4: virtual page 0 loaded to physical page 2
1
Time 5: virtual page 1 loaded to physical page 3
2
Time 6: virtual page 2  - virtual page 5 evicted - loaded to physical page 0
2
Time 7: virtual page 2 -> physical page 0
3
Time 8: virtual page 3 -> physical page 1
5
Time 9: virtual page 5  - virtual page 0 evicted - loaded to physical page 2
```

5. Discuss code supplied for the *lru* lab exercise.

```c
// Simulate LRU replacement of page frames

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <assert.h>


// represent an entry in a simple inverted page table

typedef struct ipt_entry {
    int virtual_page;        // == -1 if physical page free
    int last_access_time;
} ipt_entry_t;


void lru(int n_physical_pages, int n_virtual_pages);
void access_page(int virtual_page, int access_time, int n_physical_pages, struct ipt_entry *ipt);

int main(int argc, char *argv[]) {
    if (argc != 3) {
        fprintf(stderr, "Usage: %s <n-physical-pages> <n-virtual-pages>\n", argv[0]);
        return 1;
    }
    lru(atoi(argv[1]), atoi(argv[2]));
    return 0;
}


void lru(int n_physical_pages, int n_virtual_pages) {
    printf("Simulating %d pages of physical memory, %d pages of virtual memory\n",
          n_physical_pages, n_virtual_pages);
    struct ipt_entry *ipt = malloc(n_physical_pages * sizeof *ipt);
    assert(ipt);

    for (int i = 0; i < n_physical_pages; i++) {
        ipt[i].virtual_page = -1;
        ipt[i].last_access_time = -1;
    }

    int virtual_page;
    for (int access_time = 0; scanf("%d", &virtual_page) == 1; access_time++) {
        assert(virtual_page >= 0 && virtual_page < n_virtual_pages);
        access_page(virtual_page, access_time, n_physical_pages, ipt);
    }
}


// if virtual_page is not in ipt, the first free page is used
// if there is no free page, the least-recently-used page is evicted
//
// a single line of output describing the page access is always printed
// the last_access_time in ipt is always updated

void access_page(int virtual_page, int access_time, int n_physical_pages, struct ipt_entry *ipt) {

    // PUT YOUR CODE HERE TO HANDLE THE 3 cases
    //
    // 1) The virtual page is already in a physical page
    //
    // 2) The virtual page is not in a physical page,
    //    and there is free physical page
    //
    // 3) The virtual page is not in a physical page,
    //    and there is no free physical page
    //
    // don't forgot to update the last_access_time of the virtual_page

    printf("Time %d: virtual page %d accessed\n", access_time, virtual_page);
}
```

Answer:

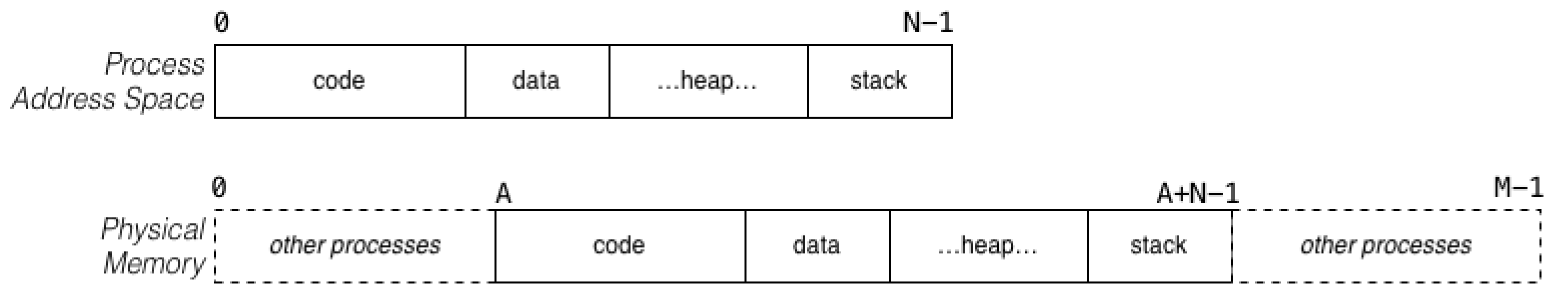
Discussed in tutorial.

6. Each new process in a computer system will have a new address space. Which parts of the address space contain initial values at the point when the process starts running? Code? Data? Heap? Stack? Which parts of the address space can be modified as the process executes?

Answer:

The Code region contains initialised values (i.e., the instructions for the program); these do not change as the process runs. The Data region contains a mix of initialised and uninitialised data objects; these objects can have their values updated as the process runs. The Heap and Stack regions contain no initial data when the process commences. The Heap changes when new data structures are [malloc]'d, and values are assigned to them. The Stack changes when functions are called and when they return.

7. One possible (and quite old) approach to loading programs into memory is to load the entire program address space into a single contiguous chunk of RAM, but not necessarily at location 0. For example:



Doing this requires all of the addresses in the program to be rewritten relative to the new base address.

Consider the following piece of MIPS code, where `loop1` is located at `0x1000`, `end_loop1` is located at `0x1028`, and `array` is located at `0x2000`. If the program containing this code is loaded starting at address `A = 0x8000`, which instructions need to be rewritten, and what addresses are in the relocated code?

```
   li  $t0, 0
   li  $t1, 0
   li  $t2, 20  # elements in array
loop1:
   bge $t1, $t2, end_loop1
   mul $t3, $t1, 4
   la  $t4, array
   add $t3, $t3, $t4
   lw  $t3, ($t3)
   add $t1, $t1, $t3
   add $t1, $t1, 1
   j   loop1
end_loop1:
```

Answer:

`loop1` is located at `0x9000` (`0x1000+0x8000`)

`end_loop1` is located at `0x9028` (`0x1028+0x8000`)

`array` is located at `0xA000` (`0x2000+0x8000`)

The instructions that need to be rewritten are shown in red:

```
   li  $t0, 0
   li  $t1, 0
   li  $t2, 20  # elements in array
loop1:
   bge $t1, $t2, end_loop1
   mul $t3, $t1, 4
   la  $t4, array
   add $t3, $t3, $t4
   lw  $t3, ($t3)
   add $t1, $t1, $t3
   add $t1, $t1, 1
   j   loop1
end_loop1:
```

Note that the `beq` instruction doesn't change because it is implemented (in SPIM) via a relative change, not an absolute address, i.e., it says “jump 40 bytes ahead”, rather than “jump to the address of label `end_loop1`.”

8. What is the difference between a *virtual address* and a *physical address*?

Answer:

A *virtual address* is an offset within the address space of a process.

A *physical address* is an absolute offset within memory, which starts from 0 and runs to the size of the memory.

9. Consider a process whose address space is partitioned into 4KB pages and the pages are distributed across the memory as shown in the diagram below:

	[0]	[1]	[2]	[3]	[4]	[5]	[6]
Process Address Space	Code1	Code2	Data	Heap	*empty*	Stack1	Stack2

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
Physical Memory	Stack1	Code2	Stack2	-	Heap	-	Data	-	-	Code1

The low byte address in the process is 0 (in Code1) and the top byte address in the process is 28671 (max address in page containing Stack2).

For each of the following process addresses (in decimal notation), determine what physical address it maps to.

a. `jal func`, where the label `func` is at 5096
b. `lw $s0,($sp)`, where `$sp` contains 28668
c. `la $t0, msg`, where the label `msg` is at 10192

Answer:

For all of these mappings, we need to determine

- *vpage* = which virtual page the process address is located in
- *vbase* = base address of this virtual page
- *offset* = offset of process address within this virtual page
- *pbase* = base address of vpage loaded into memory

The physical address is then determined as *pbase* + *voffset*.

a. Address 5096 is in the Code2 page, which is located at address *vbase* = 4096 in memory (happens to be the same address in process space), and so *offset* = 1000. Thus the physical memory address of virtual address 5096 is 4096+1000 = 5096.

b. Address 28668 is located close to the top of the stack, so must be in the Stack2 virtual page, which has address *vbase* = 24576. So, *offset* = 28668-24576 = 4092. The Stack2 page is located in memory at *pbase* = 2*4096 = 8192. Thus, the physical memory address of virtual address 28668 is 8192+4092 = 12284.

c. The address 10192 is located in the Data virtual page, which has *vbase* = 8192; *offset* = 10192-8192 = 2000. The Data page is located in the memory at *pbase* = 6*4096 = 24576. Thus, the physical memory address of virtual address 10192 is 24576+2000 = 26576.

A useful trick with 4096-byte pages: it is useful to consider the addresses in hexadecimal (or in octal) 4096 is 2^{12}, which corresponds to three hexadecimal digits, or four octal digits. So, for example, address 10192 = 0x27d0 = 023720, in page Data where *vbase* = 0x2000 = 020000, and with *offset* = 0x7d0 = 03720; Data is located at *pbase* = 0x6000 = 060000, so the virtual address 0x27d0 = 023720 corresponds to the physical address 0x67d0 = 063720.

10. The *working set* of a process could be defined as the set of pages being referenced by the process over a small window of time. This would naturally include the pages containing the code being executed, and the pages holding the data being accessed by this code.

Consider the following code, which computes the sum of all values in a very large array:

```
int bigArray[100000];
// ...
int sum = 0;
for (int i = 0; i < 100000; i++)
    sum += bigArray[i];
```

Answer the questions below under the assumptions that pages are 4 KiB (4096 bytes), all of the above code fits in a single page, the `sum` and `i` variables are implemented in registers, and there is just one process running in the system.

a. How large is the working set of this piece of code?

b. Assuming that the code is already loaded in memory, but that none of `bigArray` is loaded, and that *only* the working set is held in memory, how many page faults are likely to be generated during the execution of this code?

Answer:

a. Two pages would be a sufficient working set: one page for the code, and one page for the "current" section of the array being scanned.

b. There are no page faults associated with the code. As the code scans the array, it will read the array contents page-by-page. When it reaches the end of one page of array data and needs the next page, this will generate a page fault. The number of page faults is thus the same as the number of pages required to store `bigArray`.

Pages for array = 100000 / *elements-per-page* = 100000 / (4 KiB/4) = 100000 / 1024 = 98.

11. Consider a (very small) virtual memory system with the following properties:

    - a process with 5 pages
    - a memory with 4 frames
    - page table entries containing (Status, MemoryFrameNo, LastAccessTime)
    - pages status is one of NotLoaded, Loaded, Modified (where Modified implies Loaded)

    Page table:

Page Table	[0]	[1]	[2]	[3]	[4]
	Status	Status	Status	Status	Status
	FrameNo	FrameNo	FrameNo	FrameNo	FrameNo
	LastAccess	LastAccess	LastAccess	LastAccess	LastAccess

    indexes are page numbers

    If all of the memory frames are initially empty, and the page table entries are flagged as NotLoaded, show how the page table for this process changes as the following operations occur:

```
read page0,  read page4,  read page0,  write page4,  read page1,
read page3,  read page2,  write page2,  read page1,  read page0,
```

    Assume that a LRU page replacement policy is used, and unmodified pages are considered for replacement before modified pages. Assume also that access times are clock ticks, and each of the above operations takes one clock tick.

Answer:

```
Initial state:
[0]         [1]         [2]         [3]         [4]
NotLoaded   NotLoaded   NotLoaded   NotLoaded   NotLoaded
-           -           -           -           -
-           -           -           -           -

After "read page0" (t=1):
[0]         [1]         [2]         [3]         [4]
Loaded      NotLoaded   NotLoaded   NotLoaded   NotLoaded
frame0      -           -           -           -
1           -           -           -           -

After "read page4" (t=2):
[0]         [1]         [2]         [3]         [4]
Loaded      NotLoaded   NotLoaded   NotLoaded   Loaded
frame0      -           -           -           frame1
1           -           -           -           2

After "read page0" (t=3):
[0]         [1]         [2]         [3]         [4]
Loaded      NotLoaded   NotLoaded   NotLoaded   Loaded
frame0      -           -           -           frame1
3           -           -           -           2

After "write page4" (t=4):
[0]         [1]         [2]         [3]         [4]
Loaded      NotLoaded   NotLoaded   NotLoaded   Modified
frame0      -           -           -           frame1
1           -           -           -           4

After "read page1" (t=5):
[0]         [1]         [2]         [3]         [4]
Loaded      Loaded      NotLoaded   NotLoaded   Modified
frame0      frame2      -           -           frame1
3           5           -           -           4

After "read page3" (t=6):
[0]         [1]         [2]         [3]         [4]
Loaded      Loaded      NotLoaded   Loaded      Modified   (memory
frame0      frame2      -           frame3      frame1      now
3           5           -           6           4           full)

After "read page2" (t=7):
[0]         [1]         [2]         [3]         [4]
NotLoaded   Loaded      Loaded      Loaded      Modified   (replace
-           frame2      frame0      frame3      frame1      LRU frame)
-           5           7           6           4

After "write page2" (t=8):
[0]         [1]         [2]         [3]         [4]
NotLoaded   Loaded      Modified    Loaded      Modified
-           frame2      frame0      frame3      frame1
-           5           8           6           4

After "read page1" (t=9):
[0]         [1]         [2]         [3]         [4]
NotLoaded   Loaded      Modified    Loaded      Modified
-           frame2      frame0      frame3      frame1
-           9           8           6           4

After "read page0" (t=10):
[0]         [1]         [2]         [3]         [4]
Loaded      Loaded      Modified    NotLoaded   Modified   (replace
frame3      frame2      frame0      -           frame1      LRU/unmodified
10          9           8           -           4           frame)
```

12. Some commands on Unix allow you to name the files that they operate on, e.g.,

```
$ cat file
```

Commands that read from their standard input allow you to specify which file they read their input from by redirecting their standard input, e.g.,

```
$ cat < file
```

Describe how each of these cases might be implemented. Assume that once the file is made accessible, it is scanned and copied to the standard output (file descriptor 1, or the `#define`'d constant `STDOUT_FILENO`) as follows:

```
int infd;              // input file descriptor
char buffer[BUFSIZ];   // input file buffer
int nread;             // # characters read

while ((nread = read (infd, buffer, BUFSIZ)) > 0)
    write (STDOUT_FILENO, buffer, nread);
```

Answer:

In both cases, the `/bin/cat` command is initiated by the shell.

The first case is simple: the file name is available as `argv[1]`, so you simply need to use `open()` to obtain a file descriptor for it in the `main()` function of the *cat* command:

```
if ((infd = open (argv[1], O_RDONLY)) < 0)
    err (1, "%s", argv[1]);
```

The second case is more complicated. The *cat* command doesn't actually know the name of the file, so it needs to be given an open file descriptor for the file. The file descriptor is provided by the shell, which knows the command-line arguments as a sequence of `token[]` strings.

The shell invokes the *cat* command like

```
//// ... within the shell ... ////

char *tokens[];  // tokens (words) from the command line
char *environ[]; // environment definitions

// ... find full path name of command by scanning $PATH
// ... and storing the path name in cmdPath[]
// ... (see the Week08 lab for details)
char *cmdPath; // full path name of command (/bin/cat)

int pid = fork ();
if (pid == 0) {
    // get the file we'll be reading from
    int in;
    if ((in = open (tokens[2], O_RDONLY)) < 0)
        err (1, "%s", tokens[2]);

    // replace `stdin' by the open file descriptor
    dup2 (in, STDIN_FILENO);

    // execute the command, carrying new stdin across
    execve (cmdPath, token, environ);

    // we only reach here if execve(2) failed
    err (1, "%s", tokens[2]);
}
```

The exec'd process runs with its standard input connected to the `open`'d file. The `cat` command, seeing that it has no command-line arguments, reads from its standard input; before running the above code, it could do something like

```
if (argc == 1)
    infd = STDIN_FILENO;
```

Revision questions

The following questions are primarily intended for revision, either this week or later in session. Your tutor may still choose to cover some of these questions, time permitting.

19. Write a C program, compile.c, which is given 1+ command-line arguments which are the pathname of single file C programs. It should compile each program with dcc.

    It also should print the compile command to stdout.

```
$ dcc compile.c -o compile
$ ./compile file_sizes.c file_modes.c
/usr/local/bin/dcc file_modes.c -o file_modes
/usr/local/bin/dcc file_sizes.c -o file_sizes
```

Make sure you handle errors, for example, you should stop if any compile fails.

Answer:

```c
// compile .c files specified as command line arguments if needed
//
// if my_program.c is speicified as an argument
// /usr/local/bin/dcc my_program.c -o my_program
// will be executed unless my_program exists
// and my_program's modification time is more recent than my_program.c

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <spawn.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/wait.h>

void process_file(char *c_file);
void compile(char *c_file, char *binary);
char *get_binary_name(char *c_file);

int main(int argc, char *argv[]) {
    for (int arg = 1; arg < argc; arg++) {
        process_file(argv[arg]);
    }
    return 0;
}

// compile a C file
void process_file(char *c_file) {
    char *binary = get_binary_name(c_file);
    compile(c_file, binary);
    free(binary);
}


#define C_COMPILER "/usr/local/bin/dcc"

// compile a C file
void compile(char *c_file, char *binary) {
    pid_t pid;
    extern char **environ;
    char *cc_argv[] = {C_COMPILER, c_file, "-o", binary, NULL};

    // print compile command
    for (char **p = cc_argv; *p; p++) {
        printf("%s ", *p);
    }
    printf("\n");

    // run compile command
    if (posix_spawn(&pid, C_COMPILER, NULL, NULL, cc_argv, environ) != 0) {
        perror("spawn");
        exit(1);
    }

    int exit_status;
    if (waitpid(pid, &exit_status, 0) == -1) {
        perror("waitpid");
        exit(1);
    }

    if (exit_status != 0) {
        fprintf(stderr, "compile failed\n");
        exit(1);
    }
}


// give a string ending in .c
// return malloc-ed copy of string without .c

char *get_binary_name(char *c_file) {
    char *binary = strdup(c_file);
    if (binary == NULL) {
        perror("");
```

```
        perror("");
        exit(1);
    }

    // remove .c suffix
    char *last_dot = strrchr(binary, '.');
    if (last_dot == NULL || last_dot[1] != 'c' || last_dot[2] != '\0') {
        fprintf(stderr, "'%s' does not end in  .c\n", c_file);
        exit(1);
    }
    *last_dot = '\0';
    return binary;
}
```

20. Consider the following edited output from the *ps* command running on one of the CSE servers:

```
  PID    VSZ   RSS TTY      STAT START   TIME COMMAND
    1   3316  1848 ?        Ss   Jul08   1:36 init
  321   6580  3256 pts/52   Ss+  Aug26   0:00 -bash
  334  41668 11384 pts/44   Sl+  Aug02   0:00 vim timing_result.txt
  835   6584  3252 pts/124  Ss+  Aug27   0:00 -bash
  857  41120 10740 pts/7    Sl+  Aug22   0:00 vi echon.pl
  924   6524  3188 pts/184  Ss   15:52   0:00 -bash
  938   3664    96 pts/184  S    15:52   0:00 /usr/local/bin/checkmail
 1199   6400  3004 pts/142  Ss   Oct05   0:00 -bash
 1381  41504 11436 pts/142  Sl+  Oct05   0:00 vim PageTable.h
 2558   3664    96 pts/120  S    13:47   0:00 /usr/local/bin/checkmail
 2912  41512 11260 pts/46   Sl+  Aug02   0:00 vim IntList.c
 3483  14880  5168 pts/149  S+   Sep20   0:00 gnuplot Window.plot
 3693  41208 11240 pts/120  Tl   13:50   0:00 vim trace4
 3742   6580  3320 pts/116  Ss+  Sep07   0:00 -bash
 5531   6092  2068 pts/158  R+   16:04   0:00 ps au
 5532   4624   684 pts/158  S+   16:04   0:00 cut -c10-15,26-
 5538   3664    92 pts/137  S    15:05   0:00 /usr/local/bin/checkmail
 6620   5696  3028 pts/89   S+   Aug13   0:00 nano PingClient.java
 7132  41516 11196 pts/132  Sl+  Sep08   0:00 vim board1.s
12256 335316 10436 ?        Sl   Aug14  15:01 java PingServer 3331
12272   4260  2816 ?        Ss   Aug02  10:34 tmux
12323  10276  4564 ?        S    Sep09   0:02 /usr/lib/i386-linux-gnu/gconf/gconfd-2
12461   4260  2808 ?        Ss   Sep02   5:42 tmux
13051  43448 13320 pts/110  Sl+  Sep05   0:02 vim frequency.pl
13200  47772 21928 ?        Ssl  15:19   0:02 gvim browser.cgi
13203  41756 11560 pts/26   Sl+  Aug12   0:02 vim DLList.h
13936  11872  6856 ?        S    Sep19   0:06 /usr/lib/gvfs/gvfs-gdu-volume-monitor
30383   7624  3828 pts/77   S+   Aug23 336:28 top
```

a. Where might you look to find out the answers to the following questions?

b. What does each of the columns represent?

c. What do the first characters in the `STAT` column mean?

d. Which process has consumed the most CPU time?

e. Why do some processes have no `TTY`?

f. When was this machine last re-booted?

Answer:

a. A good place to start looking would be `man ps`.

b.
- `PID` ... Process ID
- `VSZ` ... "Virtual (memory) SiZe": #KiB in the process's address space (code+data+stack)
- `RSS` ... "Resident Set Size": #KiB of process's address space currently loaded in memory
- `TTY` ... terminal ("TeleTYpewriter") that the process is connected to
- `STAT` ... current status
- `START` ... when the process started
- `TIME` ... total CPU time used by the process so far
- `COMMAND` ... the actual command + args

ps can output other information too.

c.
- `R` ... process is currently running (or runnable)
- `S` ... process is sleeping waiting on some event (e.g. i/o)
- `T` ... process is stopped, usually via job control (control-Z)

d. Clearly the *top* process, which has used almost 6 *hours* of CPU time.

e. Some processes (e.g., system daemons) are not attached to any terminal. Other processes may "lose" their terminal if they are run in background mode, they ignore `HUP` signals, and the process that started them exits. You can also avoid having a controlling terminal by connecting all of the processes i/o streams to a non-terminal device (like `/dev/null`).

f. Since the `init` process is the one that runs when the system first starts up, its starting time would indicate when the system was last re-booted: July 8.

21. The Unix/Linux shell is a text-oriented program that runs other programs. It behaves more-or-less as follows:

```
print a prompt
while (read another command line) {
    break the command line into an array of words (args[])
    // args[0] is the name of the command, a[1],... are the command-line args
    if (args[0] starts with '.' or '/')
        check whether args[0] is executable
    else
        search the command PATH for an executable file called args[0]
    if (no executable called args[0])
        print "Command not found"
    else
        execute the command
    print a prompt
}
```

a. How can you find what directories are in the `PATH`?

b. Describe the "`search the command PATH`" process in more detail. What the kinds of system calls would be needed to determine whether there was an executable file in one of the path directories?

Answer:

1. You can find which directories are in your path using either the *env* command or `echo $PATH`.
2. To check whether a given directory *D* contained an executable file *F*, you would need to:
   - form the complete file path "*D/F*"
   - use *stat* to get information about the file
   - check that it was a regular file (and e.g. not a directory)
   - check whether it is executable by owner, group or others

22. The *kill* command (run from the shell command-line) and the *kill()* system call can be used to send any of the defined signals to a specified process. For each of the following signals, explain the circumstances under which it might be generated (apart from *kill*), and what is the default effect on the process receiving the signal:

a. `SIGHUP`

b. `SIGINT`

c. `SIGQUIT`

d. `SIGABRT`

e. `SIGFPE`

f. `SIGSEGV`

g. `SIGPIPE`

h. `SIGTSTP`

i. `SIGCONT`

Answer:

a. `SIGHUP` ... example: SSH across network, and connection is lost; remote *sshd* process receives signal; default behaviour is to terminate process.

b. `SIGINT` ... example: typing control-C to a process running in foreground mode; default behaviour is to terminate the process.

c. `SIGQUIT` ... example: typing control-\ to a process running in foreground mode; default behaviour is to terminate the process and produce a core dump.

d. `SIGABRT` ... example: the process executes an *abort* function call (e.g., when an *assert* fails); default behaviour is to terminate the process.

e. `SIGFPE` ... example: process executes a division by zero; default behaviour is to terminate the process and produce a core dump.

f. `SIGSEGV` ... example: process makes a reference through an invalid pointer; default behaviour is to terminate the process and produce a core dump.

g. `SIGPIPE` ... example: a process writing to a pipe, where the process at the other end of the pipe has terminated; default behaviour is to terminate the process.

h. `SIGTSTP` ... example: typing control-Z to a process running in foreground mode; default behaviour is to stop the process.

i. `SIGCONT` ... example: put a process into background mode and then restart it (e.g. using *fg* from the shell); default behaviour is to make the process runnable

23. The *sigaction* function for defining signal handlers takes three arguments:

    - `int signum` ... the signal whose handler is being defined
    - `struct sigaction *act` ... pointer to a record describing how to handle the signal
    - `struct sigaction *oldact` ... pointer to a record describing how the signal was handled (set by *sigaction* if not `NULL`)

    The `struct sigaction` record includes a field of type `void (*sa_handler)(int)`.

    Describe precisely what this field is, and what its type signature means.

    Answer:

    The `void (*sa_handler)(int)` field holds a pointer to the signal handler function. They type signature tells us that `sa_handler` is a pointer to a function that takes an `int` argument (the signal) and doesn't return any value (`void`).

24. Consider the following program:

```c
// assume a bunch of #include's

static void handler (int sig)
{
    printf ("Quitting...\n");
    exit (0);
}

int main (int argc, char *argv[])
{
    struct sigaction act;
    memset (&act, 0, sizeof (act));
    act.sa_handler = &handler;
    sigaction (SIGHUP, &act, NULL);
    sigaction (SIGINT, &act, NULL);
    sigaction (SIGKILL, &act, NULL);
    while (1)
        sleep (5);
    return 0;
}
```

What does this program do if it receives

a. a `SIGHUP` signal?

b. a `SIGINT` signal?

c. a `SIGTSTP` signal?

d. a `SIGKILL` signal?

Answer:

a. on `SIGHUP`, the program prints `Quitting...` and finishes, with a zero exit status

b. on `SIGINT`, the program prints `Quitting...` and finishes, with a zero exit status

c. on `SIGTSTP`, the program is stopped (not terminated) and taken out of foreground mode

d. on `SIGKILL`, the program is terminated; you cannot handle a `SIGKILL`, so the `sigaction()` call actually failed for that case