Week 10 Tutorial Questions

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?

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

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

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

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).
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?

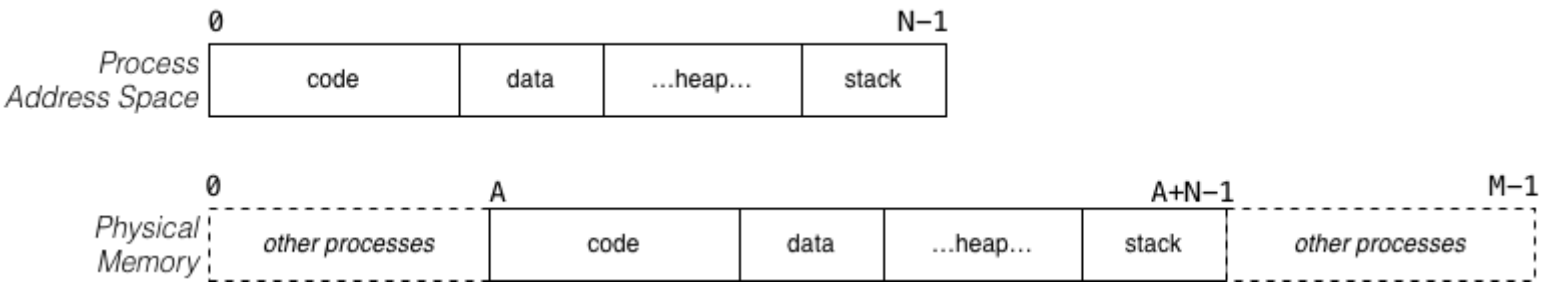
   ```
   5 3 5 3 0 1 2 2 3 5
   ```

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);
}
```

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?

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:



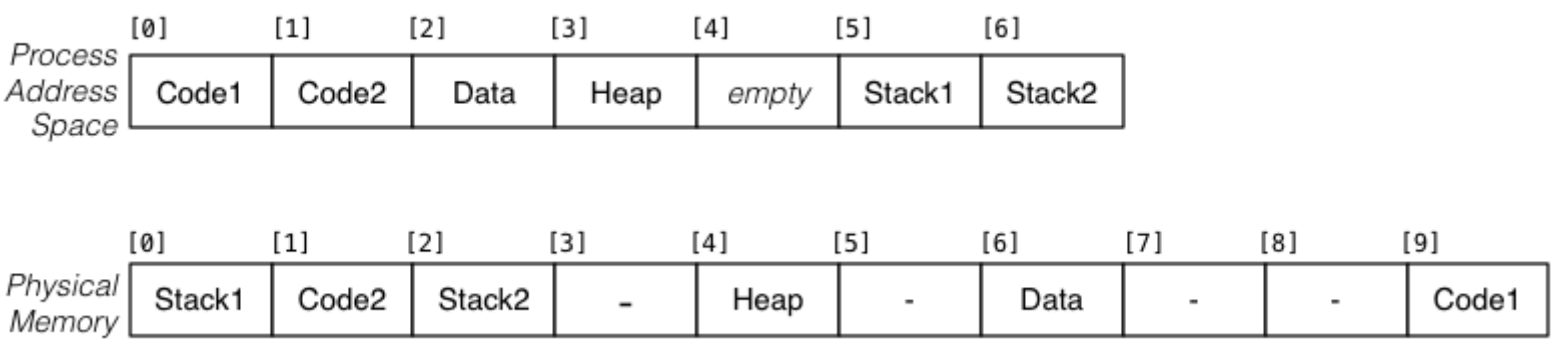
   Doing this requires all of the addresses in the program to be rewritten relative to the new base address.

   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:
```

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

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:



   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

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?

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]	*indexes are page numbers*
Status	Status	Status	Status	Status	
FrameNo	FrameNo	FrameNo	FrameNo	FrameNo	
LastAccess	LastAccess	LastAccess	LastAccess	LastAccess	

    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.

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);
    ```

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.

20. Consider the following edited output from the *ps(1)* 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?

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?

22. The *kill(1)* 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(1)*), and what is the default effect on the process receiving the signal:

a. `SIGHUP`

b. `SIGINT`

c. `SIGQUIT`

d. `SIGABRT`

e. `SIGFPE`

f. `SIGSEGV`

f. `SIGSEGV`

g. `SIGPIPE`

h. `SIGTSTP`

i. `SIGCONT`

23. The *sigaction(2)* 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(3)* 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.

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?

COMP1521 20T2: Computer Systems Fundamentals is brought to you by
the School of Computer Science and Engineering
at the University of New South Wales, Sydney.
For all enquiries, please email the class account at cs1521@cse.unsw.edu.au
CRICOS Provider 00098G