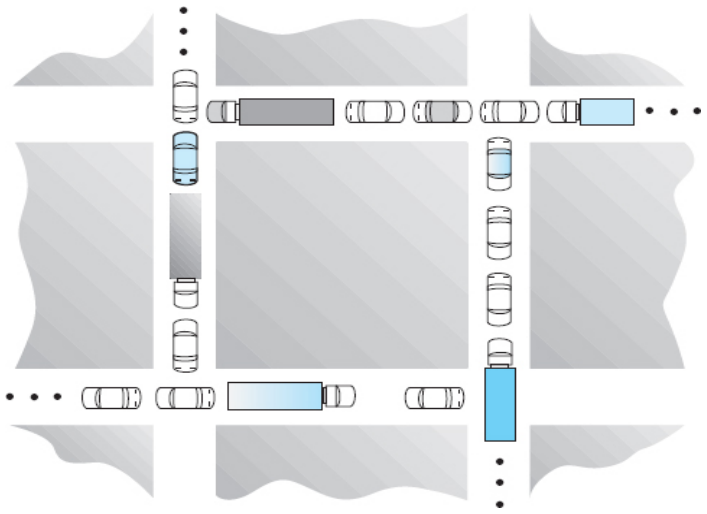
Problem Set 4: Deadlock & Main Memory

Write your answers for all parts in a file named **ps4.pdf**. You can create this file using any text editor, and then convert it to pdf.

I. Questions on Chapters 7 and 8

1. Consider the traffic deadlock depicted in the following figure.
   a. Show that the four necessary conditions for deadlock hold in this example.
   b. State a simple rule for avoiding deadlock in this system.



2. What is the difference between deadlock avoidance and deadlock preventions algorithms?

3. Consider a system consisting of four resources of the same type that are shared by three processes, each of which needs at most two resources. Show that the system is deadlock free.

4. Explain the difference between internal and external fragmentation in memory allocation. Which one happens with paging and which happens with segmentation?

5. Draw a diagram showing how virtual addresses are mapped to physical addresses using page tables. What is the time needed to access a memory location? How can we reduce this time? What is the average memory access time in this case?

6. Why are page sizes always powers of 2?

7. What is meant by "paging" the page table? And why do we do this? Are there other options for structuring page tables?

8. Explain why sharing a reentrant module is easier when segmentation is used than when pure paging is used.

II. Examining Memory of a Process

In this exercise, we will examine various parts of the memory created for a process. We will use various tools such as `gdb, top, ps,` and `proc` file system in **Linux**. The `proc` is a pseudo-file system, which provides an interface to kernel data structures. It contains a set of files located under `/proc`. Use "`man proc`" to learn more about this useful file system. For this exercise, we will focus on the file `/proc/<process_id>/maps`, which shows information about the mapped memory regions for the process with ID `process_id`.

Basic Memory

Write the following code in a file named `examine-memory.c` (can also be downloaded from the course web site).

```
#include <stdio.h>

int sum;

int func(int x)
{
     sum += x;
}

int main(int argc, char *argv)
{
     sum = 5;
     func(10);
     printf("Sum = %d\n", sum);
}
```

Then compile using -g flag:

```
$ gcc -g -o examine-memory examine-memory.c
```

And run using gdb:

```
$ gdb examine-memory
```

At the gdb prompt, issue the following commands:

```
(gdb) break main
(gdb) run
(gdb) display $pc
```

The command `break main` sets a breakpoint at the function main(), the command `run` runs the program within the debugger, and the command `display $pc` displays the content of the program counter. You should see something like the following (with different addresses):

```
Breakpoint 1, main (argc=1, argv=0x7fffffffdf58
                    "\260\342\377\377\377\177") at examine-memory.c:12
12        sum = 5;
(gdb) display $pc
1: $pc = (void (*)()) 0x400550 <main+15>
```

Use `ps –all` to find the process ID of `examine-memory`, and let us refer to it as `pid`. Open the file `/proc/<pid>/maps` and examine its contents. You may refer to the contents of this file when explaining some of your observations below.

You can also get similar information about from `gdb` using the following:

```
(gdb) info proc mapping
```

for more info on `proc` within `gdb`, issue the command:

```
(gdb) help info proc
```

In addition, you can use `gdb` to display the values of all machine registers as follows:

```
(gdb) info registers
```

Using the info in `/proc` and/or `gdb`, answer the following:

(1) Checking stack and PC:
   a. What are the start and end addresses of the stack?
   b. What are the values of the stack pointer ($sp) and frame pointer ($fp)? What is the difference between stack pointer and frame pointer?
   c. What is the name of the register that stores the value of the Program Counter ($pc)?

(2) Now continue running a few steps of the program by issuing `next 4` at the (gdb) prompt, which executes the next four lines of code (after the break point that we inserted). What is the value of the Program Counter? Why is the value of last program counter displayed vastly different from the previous ones? Issue `continue` so that the program exits (but still remains in `gdb`).

(3) Delete the previous breakpoint (command `delete`) and add another one at the beginning of function func() and run your program again. What are the addresses of x and sum (command `display &x`)? Why do we have a large difference between these addresses?

Virtual Memory

For this question, you need at least three terminal windows. In one terminal window, run

```
top -u userID
```

where userID is your user id. You should see a list of your processes listed and updated periodically. We are interested in the columns labeled VIRT, RES, SHR, which are shown by default. We are also interested in knowing the number of minor and major page faults, which are not shown by default. To show the page faults, press 'f' while top is running. Then scroll down until you see nMaj (major page faults) and press 'd' to select it (* will appear next to indicate it is chosen). Then scroll down again until you see nMin (minor page faults) and press 'd' to select it as well. Then press ESC. For more info, read the man page for `top`. Keep this window opened and visible for the rest of this question.

In a second terminal window, write the following code in a file called virtual-memory.c (can also be downloaded from the course web site).

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

#define MEGABYTE 1024*1024
#define GIGABYTE 1024*1024*1024

int main()
{
  char *ptr;
  long i = 0;
  while (1)
  {
     struct timespec sleepValue;
     sleepValue.tv_nsec = 5000000; // sleep for 5ms
     sleepValue.tv_sec = 0;  // sleep for 1 sec
     ptr  = malloc(GIGABYTE); // you use MEGABYTE if small memory
     if (ptr == NULL) {
         printf ("%ld GB could be allocated before malloc failed\n",
                    i*sizeof(size_t));
         return(0);
     }

     //memset(ptr, i, GIGABYTE); //Line A (uncomment for Q6).

     nanosleep(&sleepValue, NULL);
     i++;
  }
  return (0);
}
```

(4) Compile the above code and name the output as virtual-memory. Now, run virtual-memory. While the program is running, observe the output of top in the other window, paying attention to the columns VIRT, RES, SHR, %CPU, nMaj, and nMin, and how they change over time. Write down your observations and explain what happen.

(5) Observe the process ID of the virtual-memory process, from the window running the top command. Let say it is pid. While virtual-memory is still running, issue the following command few (3—5) times with a few seconds of interval in between.

```
cp /proc/<pid>/maps maps.<n>
```

where `<pid>` is the process ID of your process, and `<n>` is 1, 2, …, 5.

You should now have few copies of the maps file of the process, each is a snapshot at different time. Compare the content of the maps files (the Linux `diff` command can be useful here). Explain the differences between the maps files.

Wait until your program terminates. How many bytes can be allocated in your system? Is it the same as the amount of physical memory your system has?

(6) Uncomment Line A in virtual-memory.c (`mmset()`). Read the man page of `mmset()` to know what it does. Compile and run the program. Observe the output of top in the other window, paying attention to the columns VIRT, RES, SHR, %CPU, nMaj, and nMin, and how they change over time. Write down your observations and explain what happen, comparing your observations this time with what you observed in question 5.
For example, were you able to allocate more/less memory in this case compared to the previous one? How about page faults?

III. Deliverables

Submit ps4.pdf on Canvas.