Problem 1 -- Short Answers

1A) Two processes are actively running on a single-CPU, X86-32 Linux system. By default all programs are compiled such that their text regions start at the same virtual address 0x08040000. Describe what happens in terms of memory performance at context switch time if (case 1) the CPU doesn't have the PCID feature vs (case 2) it does.

1B) On X86-32 Linux systems, the initial stack region is one page long and starts at 0xBFFFF000, i.e. the `vm_start` field of the associated `vm_area_struct` is 0xBFFFF000 and `vm_end` is 0xBFFFFFFF. After the process has been running for a while, a page fault is incurred. The faulting address is 0xBFFFEFFC. What happens?

1C) We've been playing CounterStrike where the executable file is located via a network filesystem on a remote server. Everything has been working fine. But unbeknownst to us, the network cable to file server got unplugged. Subsequently, we explore a feature of the game that we hadn't previously used. What do we expect will happen and why? Hint: the M-8.

1D) A particular executable file `/home/student/a.out` is being run as process 6767. The file has mode `755 (rwxr-xr-x)`. Process 555 does `open("/home/student/a.out",O_RDWR)`. Both 555 and 6767 are the same uid. What happens and why?

1E) On a 64-bit X86 system, the first PDE (index 0) of the PGD is all-0. This means that virtual memory from 0x0000 0000 0000 0000 to _____ is unmapped. Fill in the blank, and show work!

Problem 2 -- Page Tables Traceout

Recapping the lecture notes, in the X86-32 bit architecture:

1) Virtual and physical addresses are 32 bits.

2) The page size is 4096 bytes.

3) The page table structure is 2-level.

4) All Page Directories and Page Tables are stored in whole pages at page-aligned physical addresses. These directories/tables have 1024 entries of 4 bytes each.

5) We'll make a slight fib for the purposes of this assignment and give the bit-by-bit layout of PDEs and PTEs as:

```
Bit 31: Present
Bit 30: Supervisor(=1)/User(=0)
Bit 29: Read permission
Bit 28: Write permission
Bit 27: eXecute permission
Bit 26: Dirty (n/a for PDE)
Bit 25: Accessed (n/a for PDE)
Bits 24-20: Not used, always 0
Bits 19-0: Page Frame Number
```

6) We'll further fib and disregard the A and D bits for PDEs and assume they are always 0.

Now, we are going to trace out the creation of page tables and the on-demand paging-in and allocation of page frames by the kernel, starting from a single process that forks. We make the following assumptions:

A) The kernel has a contiguous pool of free page frames. At the moment that our example process comes to life, the lowest-numbered free page frame is at physical address 0X00100000. We have an unbounded number of free page frames starting with this address and working up. The kernel will allocate page frames sequentially from the pool (ascending address order)

B) Physical memory is plentiful, therefore the PFRA is not active, and pages, once allocated, will never be "stolen."

C) At address space creation time (exec), the kernel allocates just one single page frame to serve as the PGD for that process's address space. All other page frames, including those required to store Page Tables, are demand-paged. I.e. the page frames are not pre-allocated, but are allocated and "plumbed in" as page faults occur. The page frame needed for any page tables that are created on demand is allocated prior to the actual page frame to store the data.

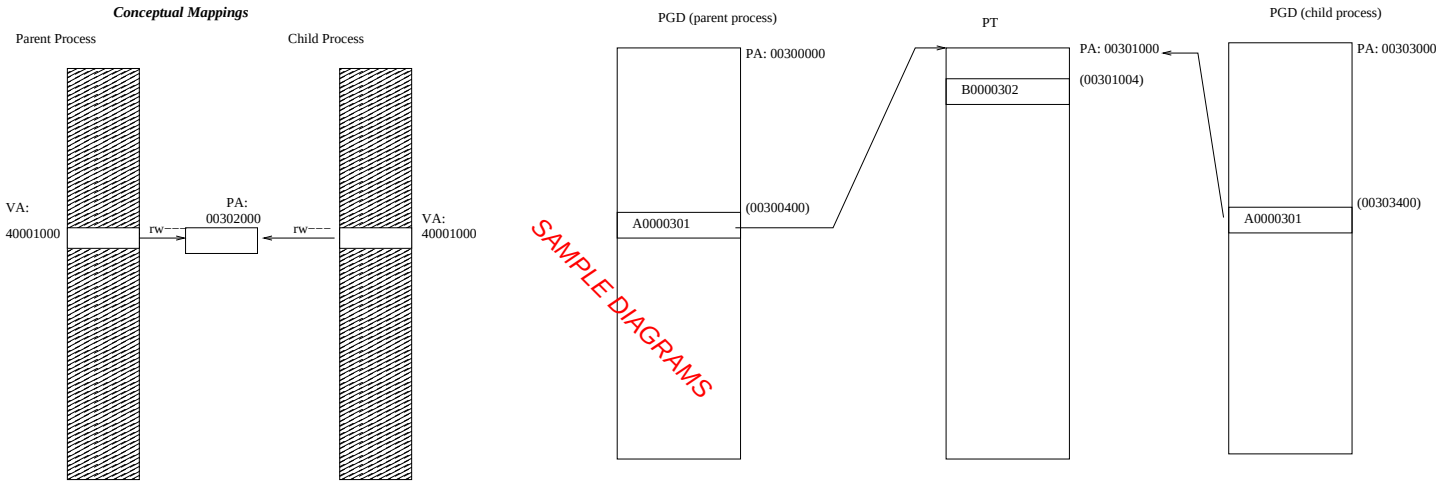
D) The kernel will lay out a process virtual address space starting at virtual address 0x08040000 for the text region. We assume that the text consumes just one page. The data and bss regions will be **contiguous** with the text region (no guard zones between memory regions); each region of course must be page-aligned. Therefore data will begin at 0x08041000. The stack will be allocated at the highest virtual address allowed for user processes (0xBFFFF000 - 0xBFFFFFFF) and grows down as needed on demand. To simplify the problem, we'll assume that the stack does not need to grow beyond the initial page.

E) To simplify further, we ignore the existence of the C library, instead assuming that the `main()` function is the first thing executed when the program is exec'd, and that the **text region fits within a single page**. We're also totally ignoring dynamically-loaded libraries (shared libraries). Obviously this is wrong, since the fork() system call code must exist someplace.

F) We'll assume that the compiler is set for no optimization, so operations occur just as written in the C source. Variables are allocated using sequential virtual addresses in the order in which they are declared in the source code (within their respective sections of course). There are no hidden allocations/declarations such as from system libraries.

G) We are concerned with user-mode pages and page directories, only, not kernel address space.

You will be making a diagram of the page table structures for **both processes** as they would exist at the line of code indicated. Here is a sample of what the diagram could look like:



NOTE: All values are in hexadecimal. All PDEs and PTEs not explicitly shown are 0

This sample diagram depicts a totally fictional process which had just one page and then forked. The parent's PGD is at PA 0x00300000, and there is a single virtual page at VA 0x40001000 which is mapped to PFN 0x00302. PFN 0x00301 is being used to hold one page table (which maps VA 0x40000000 - 0x403FFFFF) The virtual page in question has RW permissions, is a user-mode page, and has not been accessed (recently) or dirtied. It is apparently a VM_SHARED memory region, since the child process PGD at PFN 0x00303 is the same as the parent's, and both are sharing the PT at 0x00301.

Note: the example above is meant to show how the diagram should be constructed. **Other than the structure of the diagram, nothing about it is correct for this problem!**.

I have drawn the diagram two ways. On the left is a simplified or logical view, showing how the virtual address spaces are conceptually mapped to page frames. On the right side of the diagram is the physical view, showing the actual contents of physical memory containing the PGD and one PT.

Now, trace out the program below. Provide the state of the page table structure at the indicated point in the code, **both** the conceptual view and the physical view. Be sure to include the specific hexadecimal values of all valid PDEs and PTEs, including the actual physical addresses of each such entry.

Submission: This must be drawn using a drawing program in PDF, PNG, GIF, or JPEG format, or extremely neat handwritten work which is flawlessly scanned. No scribbles or crossouts! No scribbles or crossouts!

```
int b[40];
char d[]="ABC123";

main()
{
 int pid;
          d[3]=b[0];
          switch(pid=fork())
          {
                    case -1: perror("This won't happen");break
                    case 0: d[0]='Z';break;
                    default: b[0]++;
          }
          for(;;)
                    ;
  /* CONSIDER THE STATE OF THE PAGE TABLES IN BOTH PROCESSES AT THIS POINT */
}
```

Some hints: Make a sketch of the VA space of the process. Trace out each operation of the program in order and determine what virtual memory regions are accessed and for what (r/w/x). Then trace out how the kernel handled the demand-paging including the on-demand creation of page tables.

Problem 3 -- mmap vs read for grep

A common UNIX utility is the `grep` command which searches for patterns in files. We are going to write a variant of this program. Unlike the traditional grep command which was meant for line-delimited text files, we are going to be searching for binary patterns. This will allow us to find, for example, the signatures of a.out files, JPEGs, TIFs, PDFs, etc. Here is the syntax of this command:

```
SYNOPSIS:
        bgrep {OPTIONS} -p pattern_file {file1 {file 2} ...}
        bgrep {OPTIONS} pattern {file1 {file2} ...}
DESCRIPTION:
        Searches for a binary pattern in one or more input files.
        The pattern may be specified in one of two ways:
         1)A file may be specified with the -p option.  The contents
           of that file are the exact binary pattern
         2)The pattern can be specified as the first argument after the
           option flags.  For binary patterns that contain non-printable
           characters, this may be awkward and method (1) may be easier
         These two methods are mutually exclusive.  If there is no -p
         option flag, then the first argument after the option flags is
         the pattern.

        If no files are named then bgrep's sole input file is standard input.
        bgrep without any options or arguments at all is a syntax error.


        bgrep reports, for each instance of a match, the name of the file
        which matches (standard input is reported as <standard input>)
        along with the byte offset of the start of the match, and surrounding
        context if requested with the -c option
OPTIONS:
        -p pattern_file
                 Read the pattern from pattern_file

        -c context_bytes
                 When a match is found, also output the binary context
                 surrounding the match for context_bytes before and after.
                 If there are fewer than context_bytes either before or after,
                 that part of the output is truncated.  The context should
                 be presented as both characters and hex codes.  See example.
RETURNS:
        If any system call errors occur, bgrep returns -1.  Otherwise,
        if at least one instance of the pattern is found, bgrep returns 0,
        otherwise returns 1.
```

This is a tremendous pain in numerous areas to implement with conventional I/O using the read system call, requiring some tricky buffer manipulations. The pattern finding itself is not that bad, but capturing the leading and trailing context for all cases, and doing so with efficient use of system calls and memory, is a difficult problem. However, we are going to take the easy route and use the `mmap` system call to search each file as a block of memory.

However, there are limitations with this approach. You will not be able to pipe into this command, because pipes can

not be mmap'd. You will not be able to use this program from standard input if it is the terminal, because terminals are character device files and these can't be mmap'd. If you can't open or mmap a given input source, report an error and skip it, but continue searching the other inputs (if applicable). Read the definition of the program's return value carefully.

The size of a file being examined could change while you are scanning it. If the file grows, you aren't required to detect that and scan the new part (you could get stuck in an endless loop that way too). But, if the file shrinks, that could cause `SIGBUS`. You must handle that, close the file in question and move on to the other files. Since this is an error, the return code will be -1 (255) regardless if any other files matched. Here are some example runs:

```
$ ./bgrep -c 6 JFIF f1.jpg  f2.gif f3.gif f4.tif f5.tif
f1.jpg:6 ? ? ? ? ? ? J F I F ? ? ? ? ? , ?    FF D8 FF E0 00 10 4A 46 49 46 00 01 01 01 01 2C
f4.tif:7206 ? ? ? ? ? ? J F I F ? ? ? ? ? H ?    FF D8 FF E0 00 10 4A 46 49 46 00 01 02 01 00
f4.tif:2947840 F I G F G I J F I F E C D E D B A    46 49 47 46 47 49 4A 46 49 46 45 43 44 45
f5.tif:493928 ? ? ? ? ? ? J F I F ? ? ? ? ? ? ?    FF D8 FF E0 00 10 4A 46 49 46 00 01 01 00
$ echo $?
0
$ ./bgrep FOOBAR badfile.txt
Can't open badfile.txt for reading:No such file or directory
$ echo $?
255
$ ./bgrep NOSUCHPATTERN f1.jpg
$ echo $?
1
#Note in the example below, no -c option, so it just prints the byte offset of the match
$ ./bgrep pattern largefile1 largefile2  largefile3
largefile1:1024
largefile1:16384
#This example assumes that largefile2 gets truncated while it is being searched
SIGBUS received while processing file largefile2
largefile3:6
$ echo $?
255
#Below I create a pattern file with the "magic number" of an executable file
#Note that there is no newline character
$ perl -e 'print "\177ELF\001";' >pattern.bin
$ hexdump -c pattern.bin
0000000 177   E   L   F 001
0000005
#And here I find that magic number at the start of the bgrep executable
$ ./bgrep -c 8 -p pattern.bin ./bgrep
 ./bgrep:0 ? E L F ? ? ? ? ? ? ? ? ?    7F 45 4C 46 01 01 01 00 00 00 00 00 00 00
```