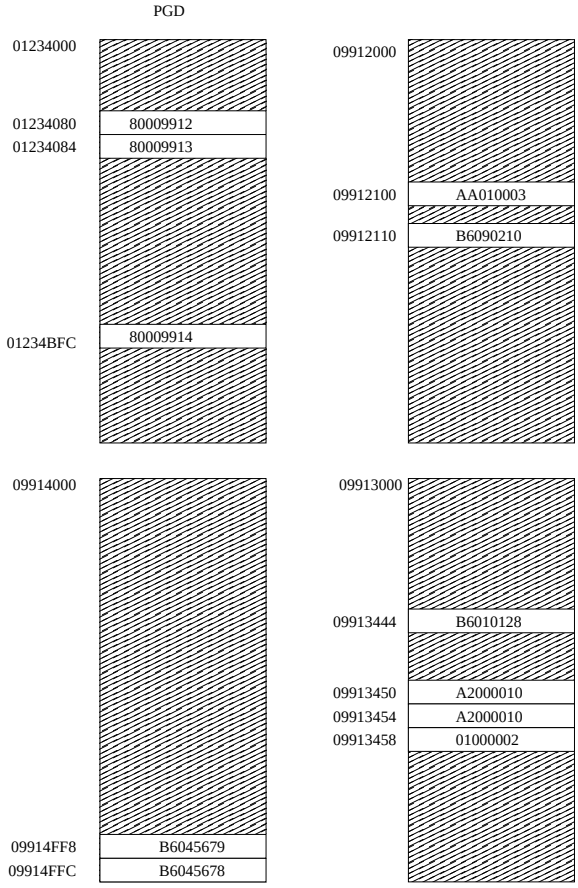
Problem 1 -- Page Tables

We are running on X86-32 Linux, but I have simplified the bit layout format of the PDE and PTE as follows:

```
Bit 31: Present
Bit 30: Supervisor(=1)/User(=0)
Bit 29: Read permission  (n/a for PDE)
Bit 28: Write permission (n/a for PDE)
Bit 27: eXecute permission (n/a for PDE)
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
```

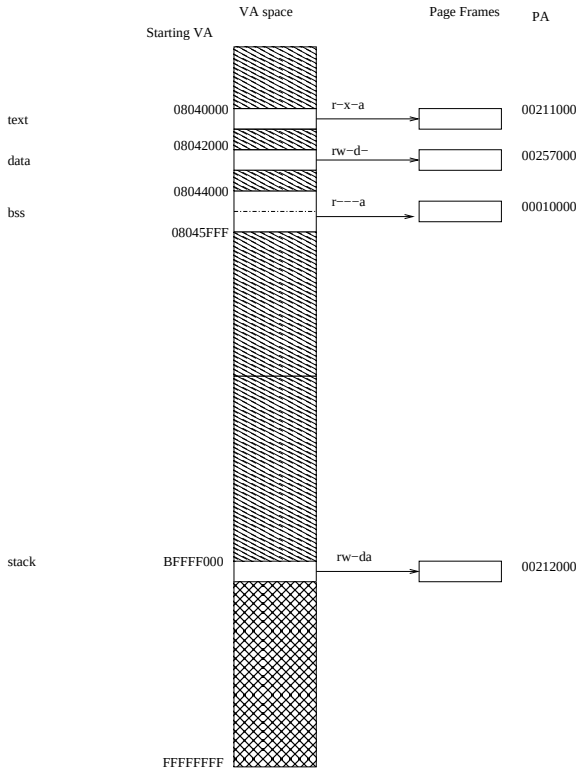
One of the simplifications I have made is to disregard the status of the RWXDA bits for PDEs. Below I am showing you the contents of certain page frames which comprise the page table structure for a given process:



In these diagrams, all numbers are in hexadecimal. The numbers to the left of the boxes are the 32-bit physical address of that 32-bit entry in the Page Table or Page Directory. The numbers within the box are the contents of that entry, in the format given above. Shaded areas of these page tables/directory indicate all-0 bytes.

The CR3 register is pointing to physical address 01234000. We also observe that the page frame at 00010000 is filled entirely with 0 bytes.

Given this information, we can partially reconstruct what the virtual address space of the process must look like. We can't know everything, e.g. if a virtual page has never been accessed there would be no trace of it in the page tables. For example, we could draw a diagram like this:



Note that the actual numbers in the example above are fictitious. This is not the solution to this homework problem! It is drawn as an example of how we could diagram the virtual address space layout, along with any pages which happen to be mapped in.

A) Make a sketch similar to this, supported by the evidence given in the page table and page directory entries above. Leave any portions of VA space where we can't draw any conclusions blank or greyed out, as shown above. Based on the clues, find the likely text, data, bss and stack regions. These will be the only 4 regions present in the address space, and they will appear in that order of ascending virtual address (as they customarily do)

B) What is the Resident Set Size (RSS) of this process?

C) We can't say for certain how big the BSS region could be (there might be parts that we haven't accessed yet) but we can answer this: what is the minimum size (in pages) that the BSS region could be?

D) Has the PFRA stolen any pages from this process? If so, which ones? What evidence supports your conclusion?

Problem 2 -- Short Answers

2) For questions A-E below, each of which describe a Page Fault situation, describe in detail how the page fault is

resolved, keeping in mind our fundamental outcome categories of major fault, minor fault, or delivery of a specific signal (tell me which one). We will use the X86-32 Linux reference model, and all of these Page Faults are from user-mode processes.

A) The faulted address is a write access to 0xBFFF2FFC. The current beginning of the stack region is 0xBFFF3000.

B) The faulted address is a read access and falls within a file-mapped region. Neither our process nor anything else on the system has ever read that part of the file.

C) As above, but the file in question resides on a USB stick that was mounted as a volume into our filesystem. The user has accidentally unplugged the stick.

D) The faulted address is an instruction fetch. The VA corresponds to the BSS region of the process.

E) The faulted address is a read and falls within an mmap region with
`PROT_READ|PROT_WRITE,MAP_SHARED|MAP_ANONYMOUS`
Our process is the only process with this region (it is not actually shared with anyone else) and we have never accessed this virtual page before.

2F) We have mmap'd a file which is currently 65536 bytes long with

```
char *p=mmap(NULL,32768,PROT_READ|PROT_WRITE,MAP_SHARED,0,fd).
```

Another process has recently written 4096 bytes to offset 16384 of this file. Then we look at the memory region, e.g. with `p[16384]`. Describe the data structures which allow the kernel to present a consistent view between the memory mapping and the file and ensure that what we see in memory is what was written to the file and vice-versa.

Problem 3 -- mmap test programs

A sophisticated programming technique is to create "test programs" which probe for the existence of certain features on a target platform, or which "prove" that something works a certain way. For example, many open-source programs use the GNU `autoconf` utility to automatically test for things such as the size of a `long`, or the presence of a specific library version, and create header files so that code can be compiled automatically on a variety of platforms.

In this assignment, you will be creating test programs to discover the answers to a variety of questions having to do with the virtual memory system and the `mmap` system call. Of course, you (should) already know the answers....they are probably in this unit's lecture notes, but your task is to create a program or system of programs to learn each answer through computational experimentation and **without user intervention**. To be a true test program, it must be capable of **determing the answer through conditional test, not simply printing out a foregone conclusion!** For example, the following is a valid test:

```
if (3+1==0) printf("ints are 3 bits long\n");
```

While the following is bogus:

```
/* Make sure to set this #define so the right answer is printed LOL */
#if INTSIZE==3
printf("ints are 3 bits long\n");
#endif
```

In addition to printing helpful messages for the benefit of a human observer, each test program will return a specific exit code as specified below. This would, if this assignment had any real benefit outside of this class, allow your test programs to be used in an autoconf-like scripting environment.

Test #1 - PROT_READ Violation

If a memory region is mmap'd with PROT_READ, but an attempt is made to write via that memory mapping, does this succeed, fail, or cause a signal? Test program will return: 0 if it succeeds (the value in memory changes), 255 if it fails (value remains the same) without causing a signal, or if a signal is received, exit value will be the signal number. To do this, you'll have to trap (handle) all possible signals.

```
$ ./mtest 1
Executing Test #1 (write to r/o mmap):
map[3]=='A'
writing a 'B'
Signal [redacted, figure it out yourself!] received
$ echo $?
{a number corresponding to the signal number is echoed here}
```

Test #2 - writing to MAP_SHARED

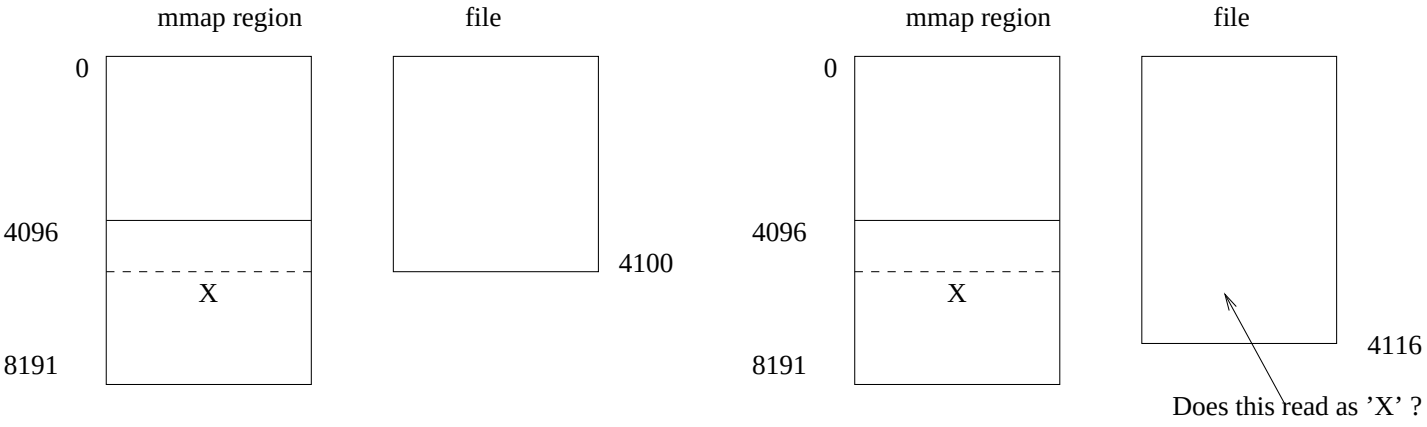
If one maps a file with MAP_SHARED and then writes to the mapped memory (you can test by writing a single byte), is that update visible when accessing the file through the traditional lseek(2)/read(2) system calls? Exit value 0 if your test shows that the file's byte changes, 1 if the byte remains the same.

Test #3 - writing to MAP_PRIVATE

Same test as above, except for MAP_PRIVATE.

Test #4 - writing into a hole

Create a small file with length that is not a multiple of the page size, map it MAP_SHARED. Refer to this illustration:



In this example, the file is 4101 bytes long (the file offset of the last byte is 4100) and the mmap region is 8192 bytes

long. The memory beyond the dashed line can be read or written, even though it does not correspond to a currently valid part of the file, because it falls within the same page. Verify that you can read this area of memory (it should read back as all 0 bytes).

Now let's call the byte at offset 4101 byte 'X' and write something there, perhaps the letter 'X' in fact. Then increase the size of the file by say 16 bytes by lseek'ing and writing one byte, thus creating a "hole" near the end of the file. Explore the file again and check to see if byte X is now visible in the file through the `read` system call. Exit code 0 if it is visible, 1 if it is not.

NOTE: You could implement 4 separate programs, or you could combine tests 1-4 into one test program which accepts a single command-line argument that is the test number to run. Either way, your test program(s) must be entirely self-contained. Do not rely on any prior manual setup (such as creating test files). You can assume that the current working directory in which the program is run has the proper permissions. You are encouraged to have additional debugging output which helps you visualize the contents of memory and file as you go along. However, the test results must be reflected in the exit codes as described above.