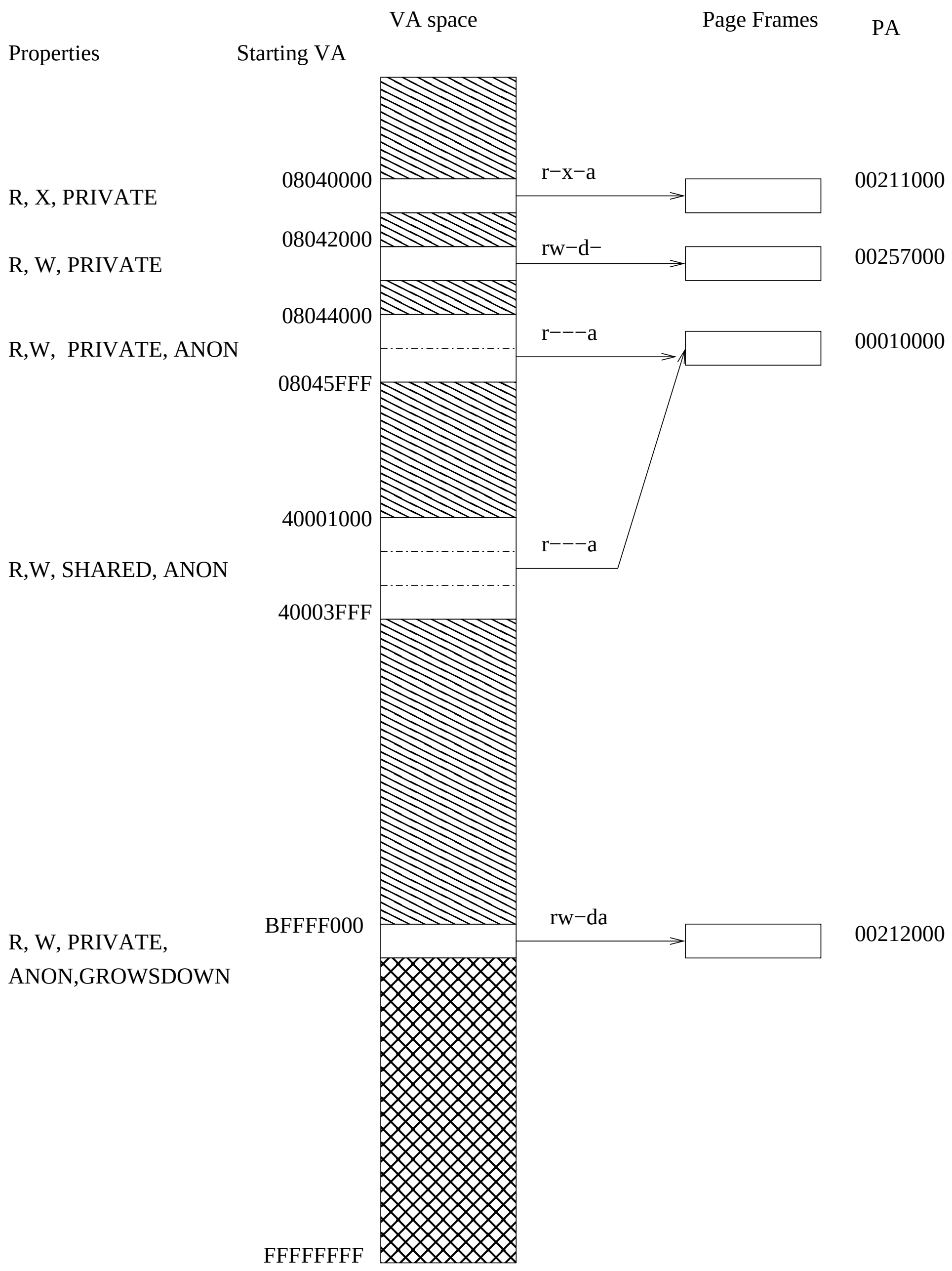
Problem 1 -- Page Tables

We envision a semi-hypothetical computer architecture and operating system which looks surprising like a simplified X86-32 / Linux. At a certain point during the execution of a certain program, we see the following:



In the diagram above, we see that there are five (5) regions in the address space of this process. We see that some, but not all, of the virtual pages have PTE mappings to page frames. The notation with the arrow implies the Present bit of the PTE, and above the line are the values of the R, W, X, D and A bits respectively. The Physical Addresses of the page frames are purely hypothetical. All addresses, PA or VA, are expressed as the full 32-bit value (i.e. not just the page number portion) and are in hexadecimal.

We also know something about the actual contents of the page frames:

00211000 contains executable opcodes, with some 0 bytes at the end

00257000 contains the following bytes (in hex): 45 43 45 33 35 37, with all the rest being 0.

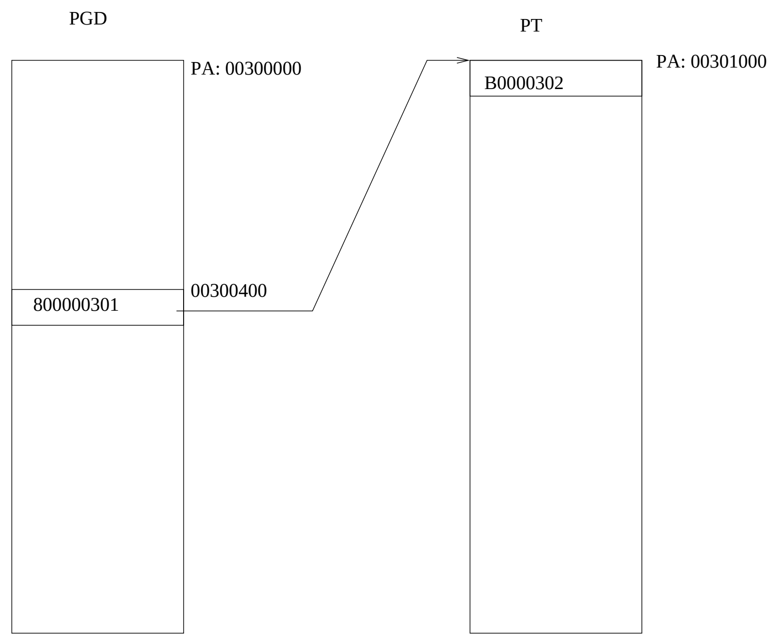
00010000 contains all 0

00212000 contains some 0 bytes, followed by unspecified bytes

Based on this information, we can at least partially reconstruct the sequence of events which got us here. Answer the following questions with reference to the above diagram:

A) Identify the type of each memory region, based on such clues as the region's properties.

B) Draw out the PGD and any PTs that the kernel would have allocated. Give the exact values (in hex) of the PDEs and PTEs that are non-zero, along with the exact Physical Address (in hex) of these PDE/PTEs. A partial example of this diagram is given below:



Note: this example is not at all correct with respect to this problem. It is merely given to illustrate how to construct the diagram. It illustrates a single mapping of VA 40000000 to PA 00302000, with PTE flags of RW.

Make the following assumptions:

i) Page frames used in creating the second-level page tables were allocated sequentially, starting with page frame 00300000

ii) The kernel does not pre-allocate any second-level page tables. They are allocated and "plumbed in" as needed during page fault resolution

iii) Both Page Table Entries (PTEs) and Page Directory Entries (PDEs) have the following bit-level structure:

```
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
```

iv) To simplify this problem, disregard the RWXDA bits in the PDEs within the PGD and assume they are 0. The P(resent) bit would be on for any PDE that is connected to an allocated second-level page table, and of course the PFN field of these PDEs needs to be considered.

v) We have no knowledge of the kernel's address space, therefore disregard and ignore it in your diagrams.

C) Compose a brief program (a main function and any appropriate declarations) which plausibly could have caused the above state to be reached. Of course, there are infinite correct solutions to this, but even more incorrect solutions! To simplify the problem,‘ pretend that the standard C library doesn't exist, and therefore main() is really the first function invoked, and furthermore no additional "hidden" variables exist. We'll disregard the awkward question of "how can we make a system call without the library?" and just assume it can be done, and furthermore that the text of this simple program could fit within 4096 bytes or less.

D) In conjunction with the above program, construct a narrative that explains each page fault that was incurred by the program, and how it was resolved (e.g. minor fault, major fault). Be sure to mention which Page Frame was allocated (if any) and, if a page-in from disk was needed, where did it come from?

E) What evidence is seen in the diagram above that the Page Frame Reclamation Algorithm (PFRA) has been recently active? Which page frame(s) *might* be a candidate for paging-out and reclamation?

Problem 2 -- 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, and write to the mapping corresponding to one byte beyond the last byte. Let's call this byte X. Now 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 all of these 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.