Problem 1 -- Page Tables

We envision a semi-hypothetical computer architecture (it is just an idealization of the actual X86-32 bit) with the following description of its page table structure (this is essentially identical to what is presented in lecture notes):

1) 32 bit virtual and physical addresses with 4K pages, 2-level page table structure split 10/10 bits.

2) The full physical address of the Page Global Directory (PGD) is contained in register CR3

3) All Page Directories and Page Tables are stored in whole pages at page-aligned physical addresses

4) Both Page Directory Entries and Page Table Entries have the following format:

```
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: eXecture 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
```

5) For Page Directory entries, the RWXD and A bits are not applicable and can be assumed as 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 using some simple, idealized examples. 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

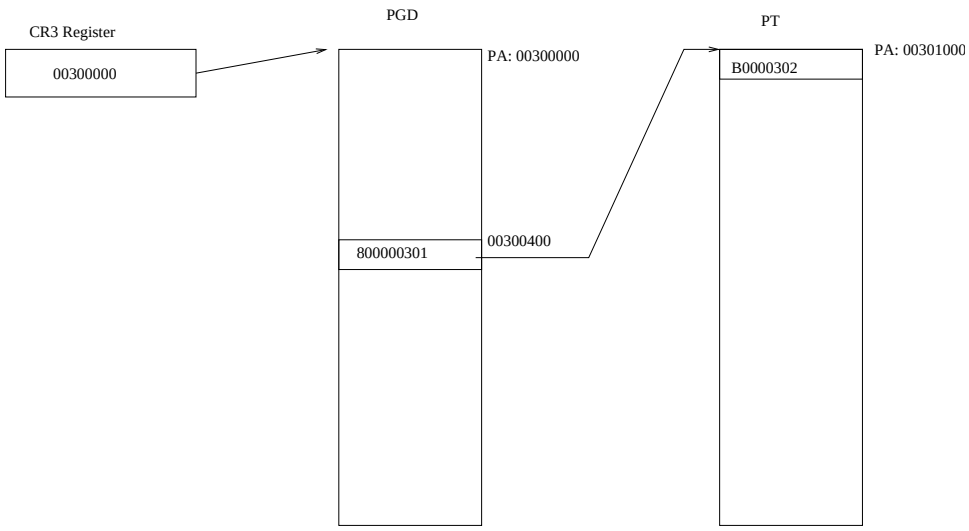
B) As the kernel requires page frames, it allocates them sequentially from this pool. The pool is contiguous and we won't run out of pages.

C) At process creation time, the kernel allocates a single page to serve as the PGD for that process's address space. All other pages, including pages 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.

D) The kernel will lay out a process virtual address space starting at virtual address 0x08040000 for the text region. The data and bss regions will be contiguous with the text region; each region of course must be page-aligned. The stack will be allocated at the higest virtual address allowed for user processes (0xBFFFF000 - 0xBFFFFFFF) and grows down as needed on demand.

E) We will assume that for our example case, the stack does not need to grow beyond a single page. We'll also simplify things and 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'll assume that the compiler is set for no optimization, so operations occur as written in the C source.

You will be making a diagram of the page table structure for a given process running a small program. Here is a sample of what the diagram would look like:



this sample diagram depicts a PGD at PA 0x00300000, and there is a single virtual page at VA 0x40000000 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 or dirtied. **Note:** the example above is meant to show how the diagram should be constructed. **It is not the correct beginning** for the solution of the actual problem below.

Now, trace out the program below. Provide the state of the page table structure at the indicated point in the code, including the value of register CR3, the contents of the PGD, and the contents of any and all intermediate Page Tables, in a form similar to the example above:

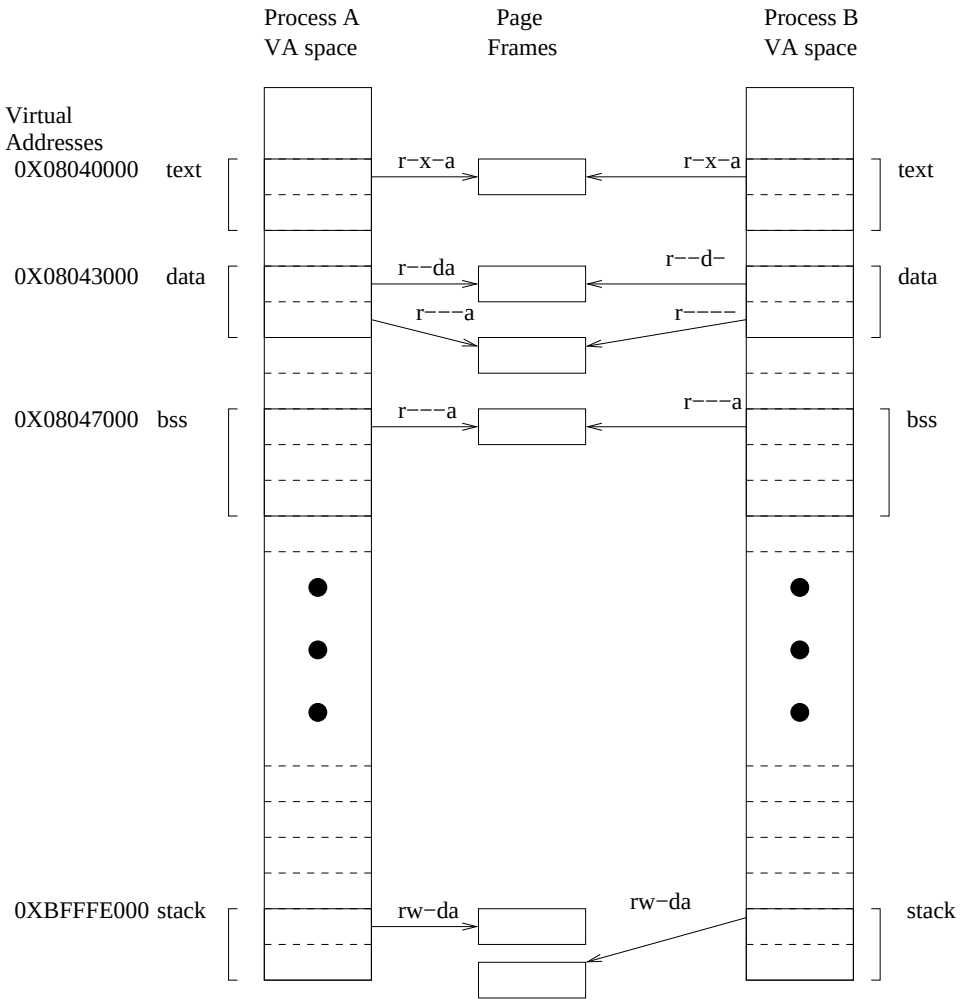
```
int b[40];
char d[]="ABC123";

main()
{
        f1();
        /* CONSIDER THE STATE OF THE PAGE TABLES AT THIS POINT */
}

f1()
{
        b[0]=d[3];
}
```

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.

Problem 2 -- Analysis of page mappings

In the sketch below, I am using the notation where page frames are represented as little boxes, without regard to the actual PFN. PTEs are represented as edges (arrows) with letters above indicating only some of the PTE flags (rwxda). The Present flag is not being shown explicitly since the presence of the arrow implies a valid PTE is present. The U/S flag is also not being shown. The multi-level structure of the page table has been simplified to show just the net result of mapping a VA to a PA.



Examining this sketch, tell me:

a) what can you reasonably infer about the ancestry relationship between processes A and B?

b) what virtual memory management optimization is being demonstrated here?

c) If process B were to write to VA 0x08044004, explain what would happen. Show how the page tables as depicted in the diagram would change.

{ attach separate page or mark up sketch above NEATLY }

Problem 3 -- Smear program

We're going to take advantage of the `mmap` system call to do something which is kind of awkward with the traditional read/write file I/O interface. I'll call the program `smear` and it will be invoked like this:

```
smear TARGET REPLACEMENT file1 {file 2....}
```

For each of the named files, search for every instance of the string given by the first argument and replace it with the replacement string given by the second argument. **Note:** to simplify the problem, both target and replacement strings must be the same number of characters.

Do this with `mmap`. Again, to simplify the problem, you aren't required to handle cases where the file is so large that it can't be mapped in its entirety into virtual address space (but still remember to check for and properly report any applicable system call errors)

Submit source code, and a screen shot with a small demonstration file showing that the intended search and replace took effect.