Theo Song
Prof. Hakner
ECE 357
11/20/2019

Problem 1 - Page Tables

A)

08040000(text): Since the page frame that the region is pointing to has an executable opcode, this region would be a text region. Furthermore, when a text region is created during exec, the protections are set as PROT_READ|PROT_EXEC, while the flags are set as MAP_PRIVATE. (MAP_DENYWRITE is silently ignored during a mmap sys call, due to security reasons)

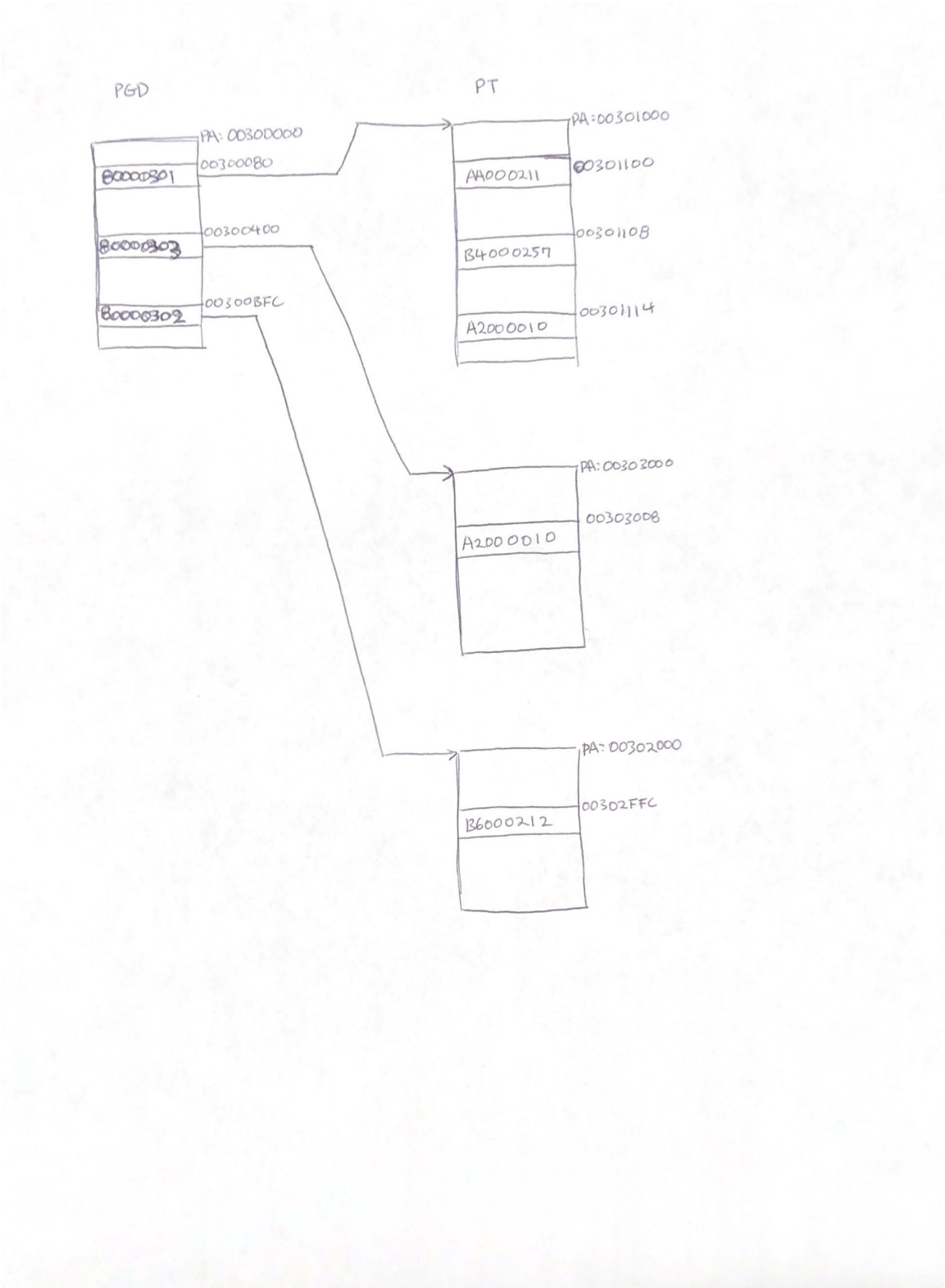
08042000(data): The first hint is that the page frame that this region is pointing to has some bytes, which are followed by all 0s, which indirectly implies that values might have been initialized. Also, when data regions are created, the flags are set as MAP_PRIVATE|MAP_DENYWRITE(explained above), while protections are set as PROT_READ|PROT_WRITE, which does match the properties of this region.

08044000(bss): The page frame that this region is pointing to contains all 0, which implies that it is a uninitialized global variable. And acknowledging the fact that when bss regions are created, the protections are set as PROT_READ|PROT_WRITE and the maps are set as MAP_ANONYMOUS|MAP_PRIVATE, which does match the properties of this region.

40001000(bss/heap): The page frame that this region is pointing is the same as the previous region. Thus, this gives us a hint that this too might be a bss. However, the flag properties of this region has MAP_SHARED, not MAP_PRIVATE. Thus, one could assume that this region is created by the mmap() system call, with the flag declared as MAP_SHARED|MAP_ANONYMOUS, which would confirm that this is a bss region.

BFFFF000(stack): The page frame that this region is pointing to contains some 0 bytes, followed by unspecified bytes. Since stack grows towards lower memory address, this implies that this region is a stack. Furthermore, since the map property has a MAP_GROWSDOWN, which is only seen in stack region, this region is a stack region.

B)


PGD
PT
PA: 00300000
00300080
80000301
00300400
80000303
00300BFC
80000302
PA: 00301000
00301100
AA000211
00301108
B4000257
00301114
A2000010
PA: 00302000
00303008
A2000010
PA: 00302000
00302FFC
B6000212

C)

```
 1  char data[6] = {0x45, 0x43, 0x45, 0x33, 0x35, 0x37};
 2  char bss1[4096+100];
 3  char *bss2;
 4
 5  int main(void){
 6      //main will be written into stack which will dirty it
 7
 8
 9      data[5] = 0x37; //dirtying
10      bss2 = mmap(NULL,4096*2+100,PROT_READ|
PROT_WRITE,MAP_SHARED|MAP_ANONYMOUS,-1,0);
11
12      while(1){
13          bss1[4096+50]; //accessing
14          bss2[4096+100]; //accessing
15
16      }
17  }
```

Notes: assuming that the kernel chooses the address to be 40001000, when mmap()ing bss2. This program is reading bss1, bss2, and stack continuously, in order to make sure that the page is accessed recently, while the data is not, which would set the accessed bit even if the PFRA activity clears the accessed bit.

D)

As the new program executes, it will have its individual PGD, which would be pre-allocated without any page faults. The execution will start out by creating the text, data, bss, and stack region in the virtual memory with mmap.
As the program intends to obtain the initial opcode from the text region, it is demand paging an executable file from disk, since it has never been accessed before, which would make a major fault. This would be resolved by creating a new PT(PA:00301000) and by reading from a.out file into the text region.
Then, the main() will be added to the stack region, which would make a minor fault, since it is trying to write to an anonymous page that has not been previously written to. This would be resolved by allocating a 0-filled page(PA: 00302000) to this stack region.
Then, the data region will be written into, which would cause a minor fault, since this is a COW situation where the file is mapped MAP_PRIVATE and the process attempts to write. This would not be a major fault because the file is cached in memory previously from the text region. (PT's PA:00301000)
For the bss region, two minor faults would occur, one for the bss1 and another for bss2 (which have been mmapped). For bss1 (PT's PA: 00301000), although it is not writing into, it has to allocate a PTE (it has not been accessed before), it would make a minor fault, which would be resolved by allocating a 0-filled page. And for bss2, there would be a minor fault since it has not been accessed before, which would be resolved by creating a new PT(PA:00303000).

E)

Accessed bits are set by the hardware when it is accessed(obviously). And the Accessed bits are cleared in two situations. It could be cleared when a process is forked, but only for the PTEs of the child process. It could also be cleared when the kernel is scanning through page frames for re-use.
Looking at the diagram above, we could see that all the access bits are set(allowing us to see that it is not a just-forked() child process), except for the PTE of the second region. Looking more carefully into this PTE, we see that the dirty bit is set, which implies that the page was written to through this PTE, which would have made the accessed bit set as well. However, for some reason, the accessed bit is cleared, which is exactly the evidence that the PFRA has been recently active.
Thus, one could predict that after the page was written via the PTE of the second region(Starting VA: 08042000), the PFRA scanned through the PTEs, clearing the accessed bit. Then, other page frames have been accessed, while the second page frame has not been accessed recently, remaining the accessed bit as cleared. So the second page frame is the most likely candidate for paging-out and reclamation.