Homework 4

Maksim Levental

October 22, 2013

Problem 1

a

The size of the virtual address is 2^{32} addresses which equals 4294967296 virtually accessible byte addresses.

b

4294967296/2048 = 2097152 pages

c

512*1024*1024/2*1024 = 512*512 = 262144 page frames.

d

2^{11} = 2048 hence 11 bits.

e

32-11 = 21 so 21 bits for the page number.

f

i

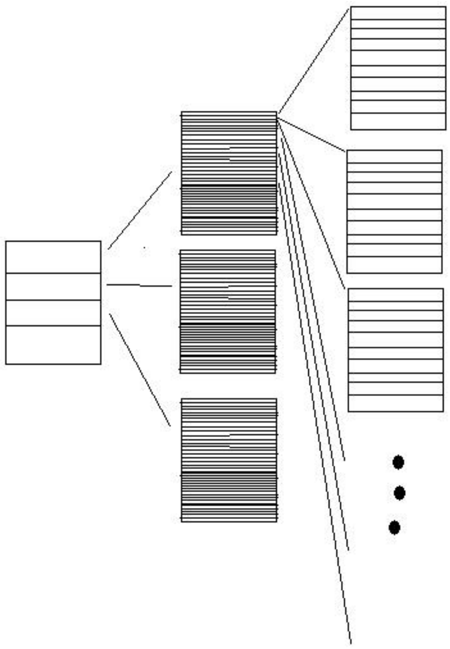
2097152 pages * 16 bytes = 32 MB.

ii

262144 page frames * 16 bytes = .25 MB.

iii

We will first divide the virtual address into 4 parts (segment table, page table dir, page table, offset). If we assume Segmentation with Paging the offset gets 11 bits, the page tables get 7 bits, the page directories get 7 bits, and the segment table gets 7 bits. So each level can index 2^7 sublevels, which is 128. So there are 128 entries in the Segment Table, then 128 page directories each with 128 entries, and finally 128 page tables per page directory each with 128 page table entries. This is 128+128*128+128*128*128 = 2113664 entries, which times 16 bytes is 32.25MB. Finally at the third level we can store all of the stack in one page table (because the stack needs only 120 entries ¡ 128) and all of text+data in 2 tables (because text+data need 230 entries ¿ 128). These page tables should be pointed to from the top and bottom of whichever page directory they're pointed to from. All other page directory and page table entries should point to null. The diagram below shows a schematic. The final level of page tables point to page frames in physical memory.



Problem 2

```c
int x = 5;
int main()
{
   int pid = fork();
   if (pid == 0)
   {
      printf("I'm the child\n");
```

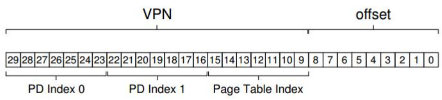
```
8        x = 10;
9        execvp(...);
10       return -1;
11     }
12     else if (pid > 0)
13     {
14       printf("I'm the parent\n");
15     }
16   return 0;
17   }
```

Line 4 causes a copy of the stack segment, because fork() is a function call. Line 8 is a write by the child so it causes a copy of the data segment to be made (because x is a global variable and so is in the data segment). Line 9 calls execvp so it causes a copy of the text segment to be made.

Problem 3

i

The virtual address is divided into 4 parts. Diagram:



ii

PD Index 0 indexes into the first Page Directory table, PD Index 1 indexes into the second Page Directory table, Page Table index indexes in the page table (selecting the page), and then finally offset indexes into the page.

iii

PD Index 0 is added to the base address stored in a system register in order to find the correct entry in the first Page directory. This entry then has a link to the base address of the second Page directory. To this base address is added PD Index 1 to find the correct entry for the Page Table. This entry contains a link to the Page Table to which the Page Table index is added to find the Page Table Entry. This entry then points to the correct page frame. The offest is then added to the base address of page frame in memory to get the correct memory address.

Problem 4

a

The block will be put into the first large enough spot: it will be put at address 31844.

b

The block will be put into the gap with the least amount of internal fragmentation: it will be put at address 44680.

c

The block will be put into the largest block: it will be put at address 1096.

d

The block will be put into the first fit after the current pointer (including checking the current pointer). Since it doesn't fit into the current pointed to gap, it gets put into the next one that it fits into, which is the one at 1096.

Problem 5

Fixed Partitioning with Relocation

$h + 1 <= VA <= g$

Assuming the call is made at runtime (eg a pointer is assigned an address by some expression) then a reference to $h + 1 <= VA <= g$ would be caught at runtime by checking the address registers stored in the PCB.

Assuming the call is made at compile time (eg a pointer is assigned an address constant) then a reference to $h + 1 <= VA <= g$ would be caught at relocation.

$max < VA$

Assuming the call is made at runtime (eg a pointer is assigned an address by some expression) then a reference to $max < VA$ would be caught at runtime by checking the limit register.

Assuming the call is made at compile time (eg a pointer is assigned an address constant) then a reference to $max < VA$ would be caught at relocation by checking the limit register.

Variable Sized Partitioning with Relocation

Same as in Fixed Size Partitioning with Relocation, because the only thing that changes is the size of the partition each process gets, not the protection scheme (assuming it's not Dynamically Variable).

Pure Paging

$h + 1 <= VA <= g$

Assuming the call is made at runtime (eg a pointer is assigned an address by some expression) then a reference to $h{+}1 <= VA <= g$ would result in a Run-time Translation error after checking the Page Table and getting a null pointer from the entry.

Assuming the call is made at compile time (eg a pointer is assigned an address constant) then a reference to $h{+}1 <= VA <= g$ would still result in a Run-time Translation error after checking the Page Table and getting a null pointer from the entry.

$max < VA$

Assuming the call is made at runtime or compile time then a reference to $max < VA$ would be the same as for $h + 1 <= VA <= g$, a page table query would result in null.

Pure Segmentation

$h + 1 <= VA <= g$

Assuming the call is made at runtime or compile time then a reference to $h + 1 <= VA <= g$ would result in a Run-time Translation error after checking the Segment Table and getting a null pointer from the entry.

$max < VA$

Assuming the call is made at runtime or compile time then a reference to $max < VA$ would be the same as for $h + 1 <= VA <= g$, a Segment table query would result in null.

Segmentation with Paging

Same as for Pure Segmentation.