Operating Systems–2: CS3523

January 2022

Programming Assignment 6: Paging

<u>Report</u>

For running extract the tar file, and open the directory in a terminal.

Type make qemu-nox in the terminal

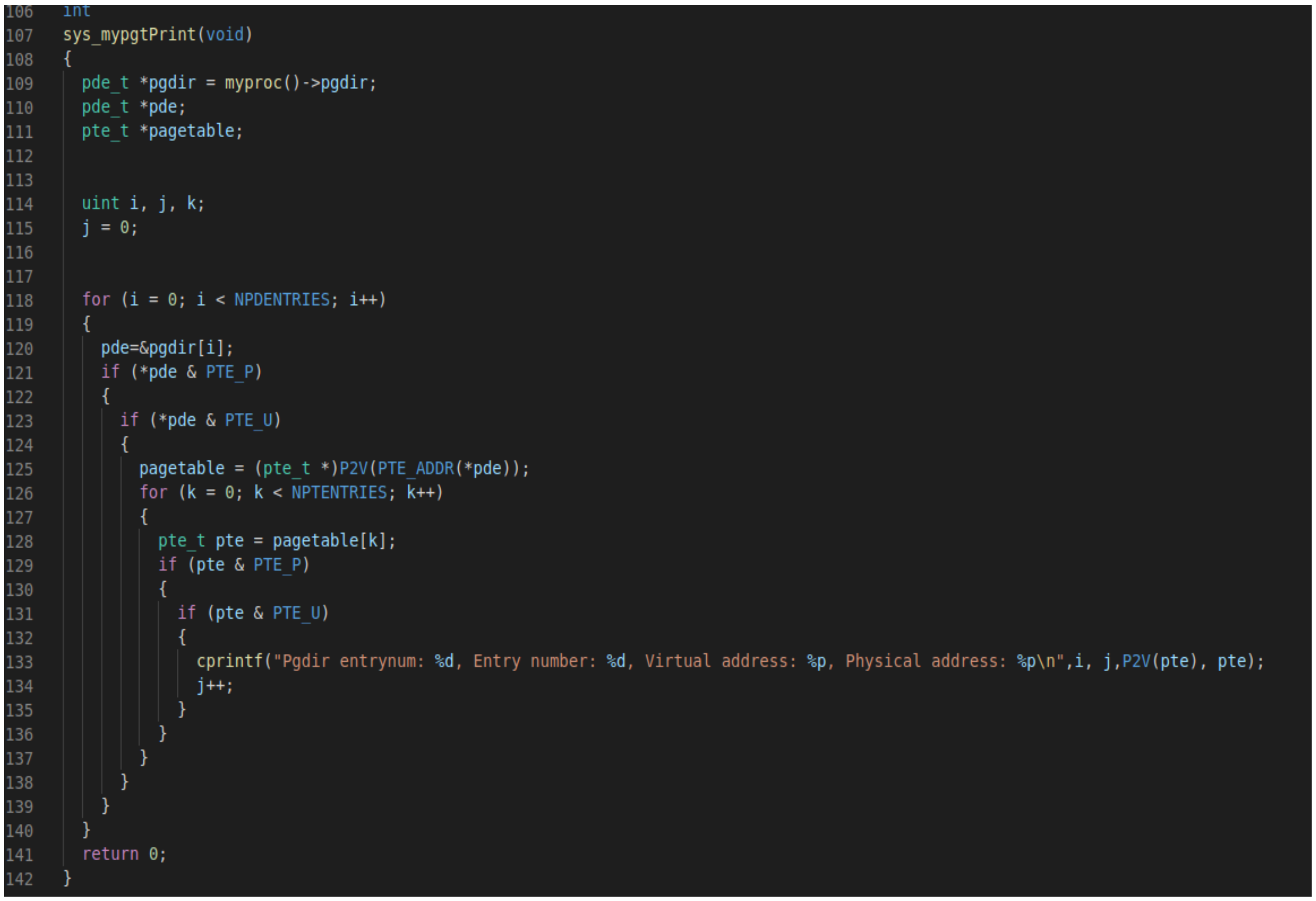
Then type mypgtprint for part-1

And mydemandPage for part-2

Part 1: Printing the page table entries

In this part we have to print the page table entries. For this we implement a user program and system call like assignment 2.

Here is the main system call code to print the page table inside sysproc.c :



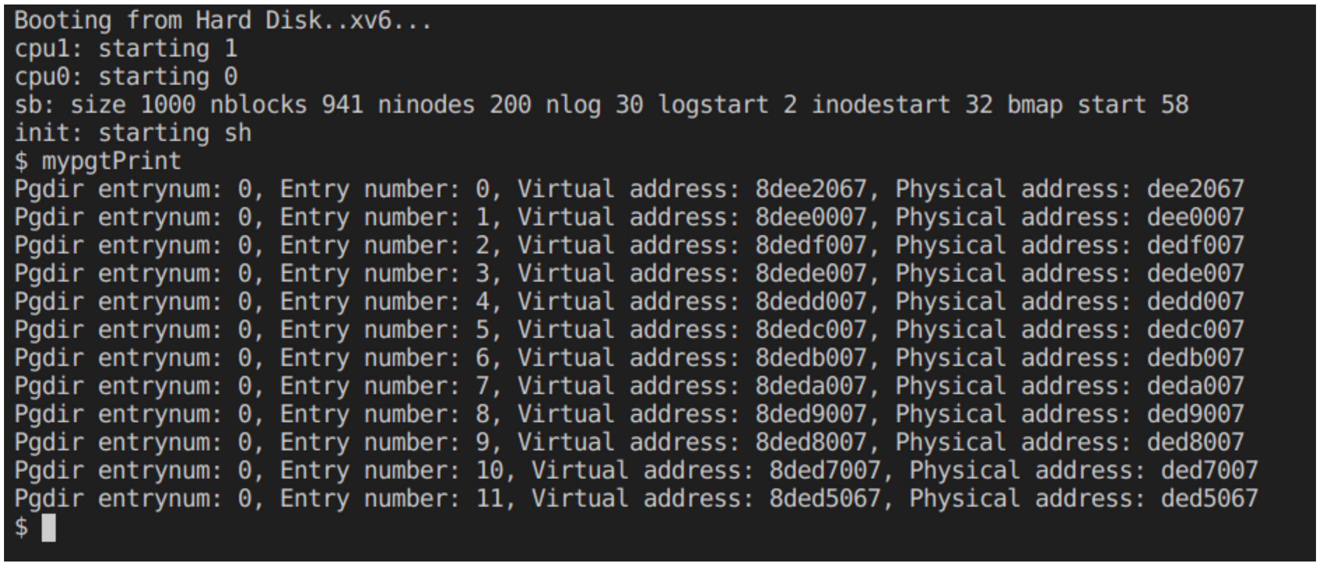
```
int
sys_mypgtPrint(void)
{
  pde_t *pgdir = myproc()->pgdir;
  pde_t *pde;
  pte_t *pagetable;


  uint i, j, k;
  j = 0;


  for (i = 0; i < NPDENTRIES; i++)
  {
    pde=&pgdir[i];
    if (*pde & PTE_P)
    {
      if (*pde & PTE_U)
      {
        pagetable = (pte_t *)P2V(PTE_ADDR(*pde));
        for (k = 0; k < NPTENTRIES; k++)
        {
          pte_t pte = pagetable[k];
          if (pte & PTE_P)
          {
            if (pte & PTE_U)
            {
              cprintf("Pgdir entrynum: %d, Entry number: %d, Virtual address: %p, Physical address: %p\n",i, j,P2V(pte), pte);
              j++;
            }
          }
        }
      }
    }
  }
  return 0;
}
```

So, we see clearly in the program we get the page directory of the current process. And declare variables for the page table. Then in two for loops we get two levels deeper to get each page table entry. The outer loop runs 1024 times (using a macro provided in xv6). Each of the pgdir[i] entries are the pagetables. But these are the physical addresses of the pagetables so we use different macros to obtain their virtual address. The user mode and valid mode bits are checked for the pagetables.

Then we go in the inner loop which runs on each of the 1024 pagetables which in turn have 1024 entries. Here again each of the pagetable[i] are the page table entries. We further check if these are the valid entries by using the same bits as earlier. Then lastly we print the page table entries.

Observations:

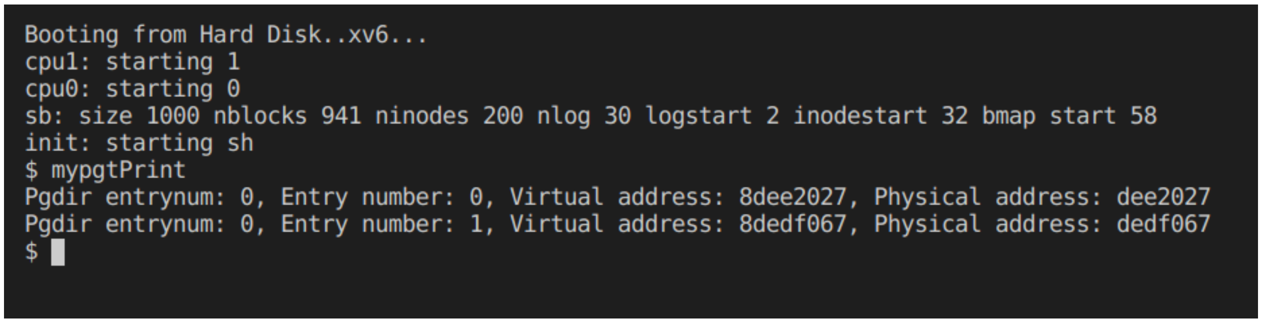
1. Global array declaration uses memory and this memory needs to be saved inside the memory. Thus page tables are used to map these array entries and page table entries increase because of the global array declaration.



```
Booting from Hard Disk..xv6...
cpu1: starting 1
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$ mypgtPrint
Pgdir entrynum: 0, Entry number: 0, Virtual address: 8dee2067, Physical address: dee2067
Pgdir entrynum: 0, Entry number: 1, Virtual address: 8dee0007, Physical address: dee0007
Pgdir entrynum: 0, Entry number: 2, Virtual address: 8dedf007, Physical address: dedf007
Pgdir entrynum: 0, Entry number: 3, Virtual address: 8dede007, Physical address: dede007
Pgdir entrynum: 0, Entry number: 4, Virtual address: 8dedd007, Physical address: dedd007
Pgdir entrynum: 0, Entry number: 5, Virtual address: 8dedc007, Physical address: dedc007
Pgdir entrynum: 0, Entry number: 6, Virtual address: 8dedb007, Physical address: dedb007
Pgdir entrynum: 0, Entry number: 7, Virtual address: 8deda007, Physical address: deda007
Pgdir entrynum: 0, Entry number: 8, Virtual address: 8ded9007, Physical address: ded9007
Pgdir entrynum: 0, Entry number: 9, Virtual address: 8ded8007, Physical address: ded8007
Pgdir entrynum: 0, Entry number: 10, Virtual address: 8ded7007, Physical address: ded7007
Pgdir entrynum: 0, Entry number: 11, Virtual address: 8ded5067, Physical address: ded5067
$ 
```

This illustrates the results.

2. Local Array: When a local array is declared these variables are limited just for the scope of the user program. Therefore, these entries are not mapped on the page table and the number of page table entries do not increase in this case.



```
Booting from Hard Disk..xv6...
cpu1: starting 1
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$ mypgtPrint
Pgdir entrynum: 0, Entry number: 0, Virtual address: 8dee2027, Physical address: dee2027
Pgdir entrynum: 0, Entry number: 1, Virtual address: 8dedf067, Physical address: dedf067
$ 
```

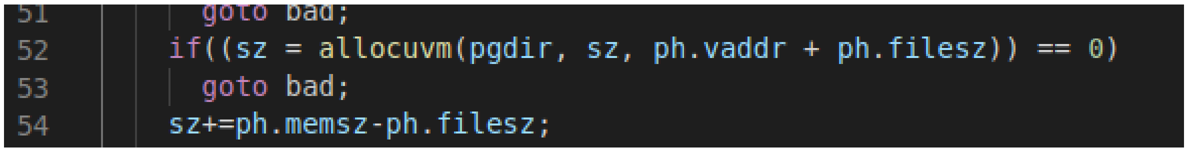
The image illustrates the observations.

3. On repeating the execution multiple time we find out that the physical addresses and local addresses do not change. This shows that

xv6 os allocates addresses and maps them to page table entries in a fixed manner i.e. without any changes in every boot up.

Part-2: Implement demand paging

For part-2 we have to include demand paging for the dynamically allocated variables. According to the hints we have to make changes inside the exec.c file to make page faults. Xv6 allocates the space for all the data beforehand. To change this we make changes in the exec.c allocuvm function as follows:



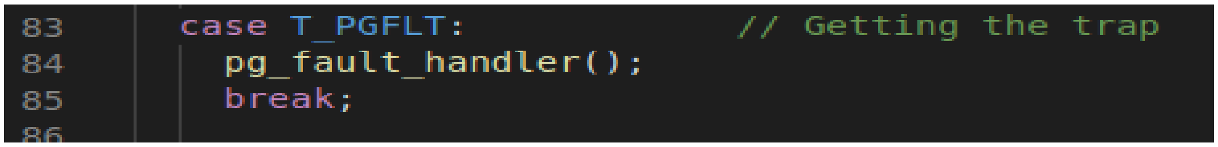
```
      goto bad;
    if((sz = allocuvm(pgdir, sz, ph.vaddr + ph.filesz)) == 0)
      goto bad;
    sz+=ph.memsz-ph.filesz;
```

Here earlier it was memsz instead of the filesz. The program headers include information about the various details of the program so that we can allocate and handle the memory accordingly. The memsz entry signifies how much data will actually be in the memory. And the filesz data is the size of the program on the disk. Therefore all the read only data size is included in the filesz entry and dynamic entries are present in the memsz section.

We change the allocation size such that only the read only code is loaded into the memory and we allocate the dynamic data on demand.

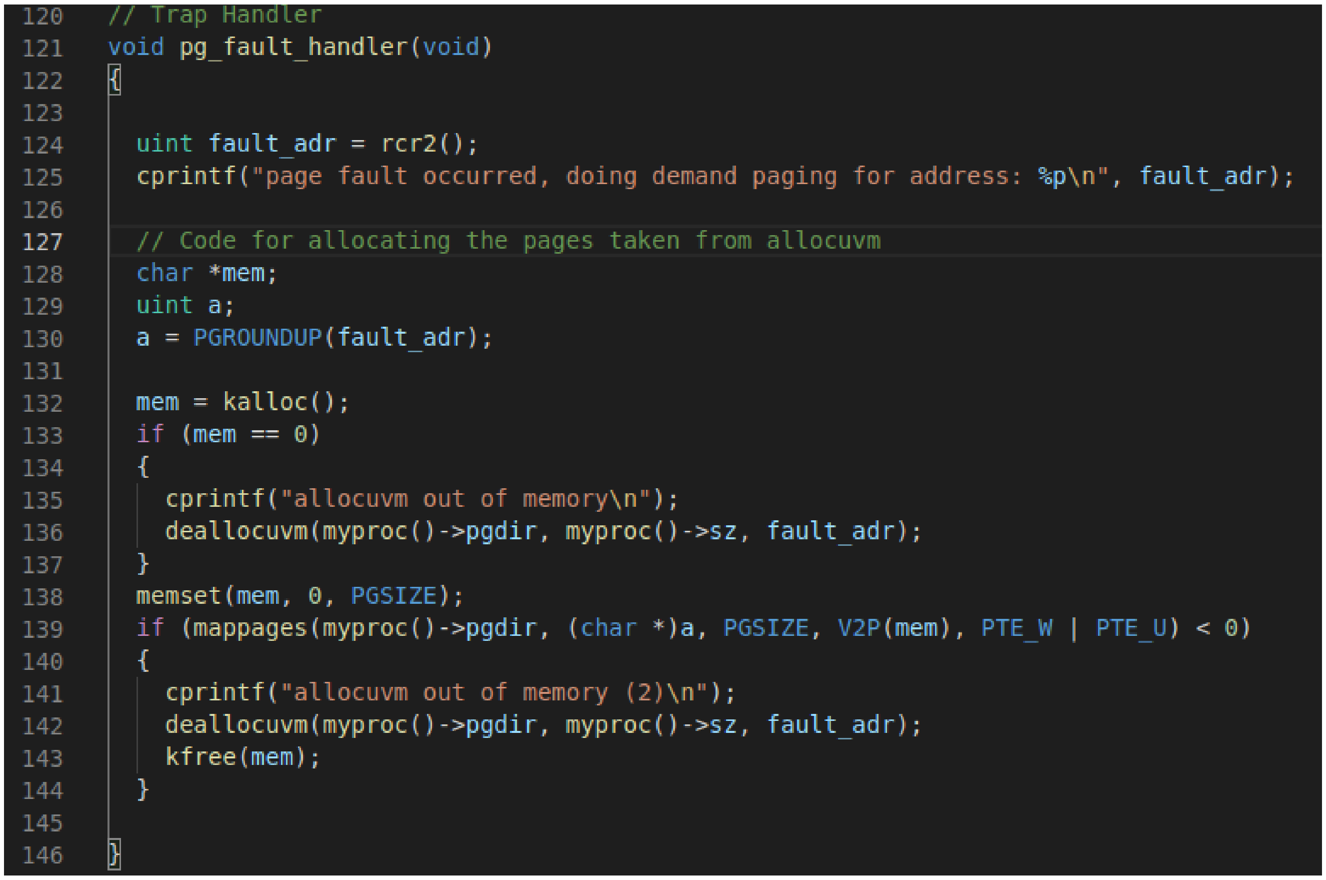
Handling Page faults:

Now when we make global arrays of size 3000 the dynamic memory is not allocated and this gives a page fault. Which is handled in trap. c File.



```
  case T_PGFLT:          // Getting the trap
    pg_fault_handler();
    break;
```

When the page fault occurs a trap with trap no. 14 is generated and trap handler function is called,

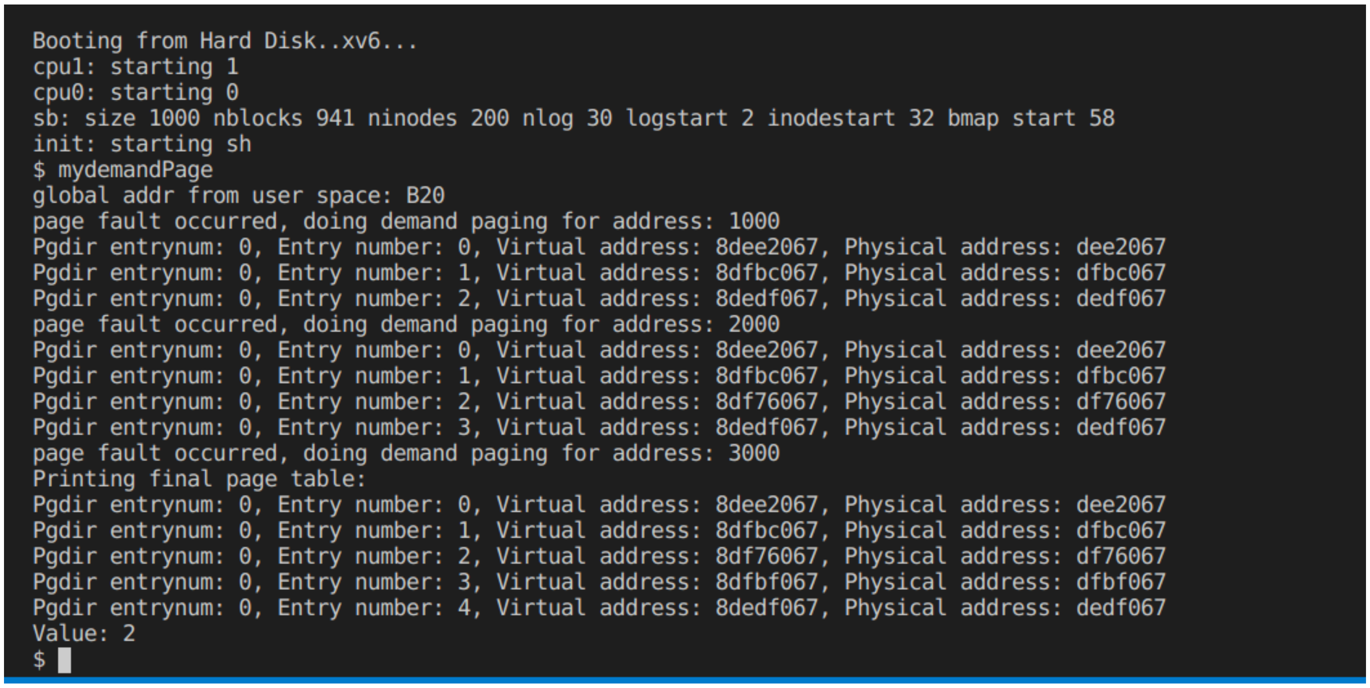


```
// Trap Handler
void pg_fault_handler(void)
{

  uint fault_adr = rcr2();
  cprintf("page fault occurred, doing demand paging for address: %p\n", fault_adr);

  // Code for allocating the pages taken from allocuvm
  char *mem;
  uint a;
  a = PGROUNDUP(fault_adr);

  mem = kalloc();
  if (mem == 0)
  {
    cprintf("allocuvm out of memory\n");
    deallocuvm(myproc()->pgdir, myproc()->sz, fault_adr);
  }
  memset(mem, 0, PGSIZE);
  if (mappages(myproc()->pgdir, (char *)a, PGSIZE, V2P(mem), PTE_W | PTE_U) < 0)
  {
    cprintf("allocuvm out of memory (2)\n");
    deallocuvm(myproc()->pgdir, myproc()->sz, fault_adr);
    kfree(mem);
  }

}
```

Here is the page fault handler function. When the page fault occurs the faulting address is stored in the rcr2 register. Then later we allocate a page for thaat process using kalloc() function. Then this function returns us with a page. This page is intialized wih the memst() function.

This page is then mapped to the process's page directory using the mappages function. This code for allocating memory and mapping was taken from the allocuvm function present inside the vm.c file.

When mydemandpage is typed on the xv6 shell we see this output:



```
Booting from Hard Disk..xv6...
cpu1: starting 1
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$ mydemandPage
global addr from user space: B20
page fault occurred, doing demand paging for address: 1000
Pgdir entrynum: 0, Entry number: 0, Virtual address: 8dee2067, Physical address: dee2067
Pgdir entrynum: 0, Entry number: 1, Virtual address: 8dfbc067, Physical address: dfbc067
Pgdir entrynum: 0, Entry number: 2, Virtual address: 8dedf067, Physical address: dedf067
page fault occurred, doing demand paging for address: 2000
Pgdir entrynum: 0, Entry number: 0, Virtual address: 8dee2067, Physical address: dee2067
Pgdir entrynum: 0, Entry number: 1, Virtual address: 8dfbc067, Physical address: dfbc067
Pgdir entrynum: 0, Entry number: 2, Virtual address: 8df76067, Physical address: df76067
Pgdir entrynum: 0, Entry number: 3, Virtual address: 8dedf067, Physical address: dedf067
page fault occurred, doing demand paging for address: 3000
Printing final page table:
Pgdir entrynum: 0, Entry number: 0, Virtual address: 8dee2067, Physical address: dee2067
Pgdir entrynum: 0, Entry number: 1, Virtual address: 8dfbc067, Physical address: dfbc067
Pgdir entrynum: 0, Entry number: 2, Virtual address: 8df76067, Physical address: df76067
Pgdir entrynum: 0, Entry number: 3, Virtual address: 8dfbf067, Physical address: dfbf067
Pgdir entrynum: 0, Entry number: 4, Virtual address: 8dedf067, Physical address: dedf067
Value: 2
$ 
```

We can see the desired output from the assignment problem.

When we change the value of global array to 4000 and 5000 we observe thaat the page faults are increased since now new and more dynamic memory must be allocated.

Files changed:

Part 1:

syscall.h:

defs.h

syscalls.c

sysproc.c

usys.S

user.h

mypgtPrint.c

Makefile

For part-2

Makefile

mydemandPage.c

trap.c

exec.c