ELL405 Operating Systems Assignment 2

Pratyush Pandey (2017EE10938)
Aniket Ulhasshetty (2018EE10443)

20^{nd} March, 2021

Contents

1 Memory Management in XV6

1.1 Initializing the memory subsystem

The basic understanding of the way memory management is done in XV6 can be obtained using the following summary point

- xv6 uses 32-bit virtual addresses, resulting in a virtual address space of 4GB. xv6 uses paging to manage its memory allocations. However, xv6 does not do demand paging, so there is no concept of virtual memory.
- xv6 uses a page size of 4KB, and a two level page table structure.
- Remember that once the MMU is turned on, for any memory to be usable, the kernel needs a virtual address and a page table entry to refer to that memory location. When main starts, it is still using entrypgdir which only has page table mappings for the first 4MB of kernel. If the kernel wants to use more than this 4MB, it needs to map all of that memory as free pages into its address space, for which it needs a larger page table. So, main first creates some free pages in this 4MB in the function kinit1, which eventually calls the functions freerange and kfree.
- The kernel uses the struct run data structure to address a free page. This structure simply stores a pointer to the next free page, and the rest of the page is filled with garbage.

- The function setupkvm works as follows. For each of the virtual to physical address mappings in kmap, it calls mappages. The function mappages walks over the entire virtual address space in 4KB page-sized chunks, and for each such logical page, it locates the PTE using the walkpgdir function. walkpgdir simply outputs the translation that the MMU would do. It uses the first 10 bits to index into the page table directory to find the inner page table. If the inner page table does not exist, it requests the kernel for a free page, and initializes the inner page table. Note that the kernel has a small pool of free pages setup by kinit1 in the first 4MB address space—these free pages are used to construct the kernel's page table. Once walkpgdir returns the PTE, mappages sets up the appropriate mapping using the physical address it has.

1.2 Creating user processes

- The function userinit creates the first user process. We will examine the memory management in this function. The kernel page table of this process is created using setupkvm as always. For the user part of the memory, the function inituvm (line 1903) allocates one physical page of memory, copies the init executable into that memory, and sets up a page table entry for the first page of the user virtual address space.
- If the child wants to execute a different executable from the parent, it calls exec right after fork. For example, the init process forks a child and execs the shell in the child. The exec system call copies the binary of an executable from the disk to memory and sets up the user part of the address space and its page tables.
- Exec first reads the ELF header of the executable from the disk and checks that it is well formed. It then initializes a page table, and sets up the kernel mappings in the new page table via a call to setupkvm. Then, it proceeds to build the user part of the memory image via calls to allocuvm and loaduvm for each segment of the binary executable. allocuvm allocates physical pages from the kernel's free pool via calls to kalloc, and sets up page table entries. loaduvm reads the memory executable from disk into the allotted page using the readi function.
- Next, exec goes on to build the rest of its new memory image. The guard page has no physical memory frame allocated to it, so any access beyond the stack into the guard page will cause a page fault. Then, the arguments to exec are pushed onto the user stack, so that the exec binary can access them when it starts.

2 Implementation

All the macros and small modifications have been added to the code, and described above, and can be seen using a git diff with the base code of XV6. However, the key/critical implementations have been written and explained. We make changes to the following files to add functionalities -

2.1 fs.c

- `readFromSwapFile`: Returns sys_read (-1 when error)
- `writeToSwapFile`: Returns sys_write (-1 when error)
- `getFreeSlot`: Returns free slot in file (-1 when full)
- `writePageToFile`: Place data to file
- `readPageFromFile`: Read data from file (-1 if wrong page address)
- `copySwapFile`: 1 if swapFile not in use, copy contents

```c

/**
 * remove swap file of proc p
 * @param p
 * @return
 */
int removeSwapFile(struct proc* p){ ... }

/**
 * return as sys_read (-1 when error)
 * @param p
 * @param buffer
 * @param placeOnFile
 * @param size
 * @return
 */
int readFromSwapFile(struct proc * p, char* buffer, uint placeOnFile, uint size){ ... }

/**
 * return as sys_write (-1 when error)
 * @param p
 * @param buffer
 * @param placeOnFile
 * @param size
 * @return
 */
int writeToSwapFile(struct proc * p, char* buffer, uint placeOnFile, uint size){
    p->swapFile->off = placeOnFile;
    return filewrite(p->swapFile, buffer, size);
}

/**
 *
 * @param p
 * @param userPageVAddr
 * @param pgdir
 * @return
 */
int writePageToFile(struct proc * p, int userPageVAddr, pde_t *pgdir) {
    int freePlace = getFreeSlot(p);
    int retInt = writeToSwapFile(p, (char*)userPageVAddr, PGSIZE*freePlace, PGSIZE);
    if (retInt == -1)
        return -1;
    p->fileCtrlr[freePlace].state = USED;
    p->fileCtrlr[freePlace].userPageVAddr = userPageVAddr;
    p->fileCtrlr[freePlace].pgdir = pgdir;
    p->fileCtrlr[freePlace].accessCount = 0;
    p->fileCtrlr[freePlace].loadOrder = 0;
    return retInt;
}

/**
 *
 * @param p
 * @param ramCtrlrIndex
 * @param userPageVAddr
 * @param buff
 * @return
 */
int readPageFromFile(struct proc * p, int ramCtrlrIndex, int userPageVAddr, char* buff) { ... }

/**
 *
 * @param p
 * @return 0 on success
 */
int createSwapFile(struct proc* p){ ... }

/**
```

```
 *
 * @param fromP
 * @param toP
 */
void copySwapFile(struct proc* fromP, struct proc* toP){ ... }
```

2.2 kalloc.c

- `getFreePages`: Returns free pages
- `getTotalPages`: Returns total number of pages

2.3 proc.c

- `getPagedOutAmout`: flag indicating if page is swapped out
- `updateLap`: proc is either running, runnable or sleeping, update access counters
- `initSwapStructs`: Constructor/Initialiser

```
/**
 *
 * @param p
 */
void initSwapStructs(struct proc* p) {
    int i;
    for (i = 0; i < MAX_TOTAL_PAGES - MAX_PYSC_PAGES; i++)
        p->fileCtrlr[i].state = NOTUSED;
}
**
 *
 * @param p
 * @return
 */
int getPagedOutAmout(struct proc* p){

    int i;
    int amout = 0;

    for (i=0;i < MAX_PYSC_PAGES; i++){
        if (p->fileCtrlr[i].state == USED)
            amout++;
    }
    return amout;
}

/**
 *
 */
void updateLap(){
    struct proc *p;
    acquire(&ptable.lock);
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
        if (p->pid > 2 && p->state > 1 && p->state < 5) //proc is either running,
    runnable or sleeping
            updateAccessCounters(p); //implemented in vm.c
    }
    release(&ptable.lock);
}
```

2.4 vm.c

- `getPagePAddr`: flag indicating if page is swapped out
- `fixPagedOutPTE`: clear junk physical address, refresh CR3 register

- `fixPagedInPTE`: This method cannot be replaced with mappages because mappages cannot turn off PTE_PG bit
- `pageIsInFile`: PAGE IS IN FILE
- `getLIFO`
- `getSCFIFO`
- `getLAP`
- `getPageOutIndex`
- `updateAccessCounters`
- `getPageFromFile`
- `getFreeRamCtrlrIndex`
- `addToRamCtrlr`
- `swap`
- `isNONEpolicy`
- `removeFromRamCtrlr`
- Added a lot of additional code in other functions of the file vm.c

```

/**
 *
 * @param userPageVAddr
 * @param pgdir
 * @return
 */
int getPagePAddr(int userPageVAddr, pde_t * pgdir){
    pte_t *pte;
    pte = walkpgdir(pgdir, (int*)userPageVAddr, 0);
    if(!pte) //uninitialized page table
        return -1;
    return PTE_ADDR(*pte);
}

/**
 *
 * @param userPageVAddr
 * @param pgdir
 */
void fixPagedOutPTE(int userPageVAddr, pde_t * pgdir){
    ...
}

/**
 * This method cannot be replaced with mappages because mappages cannot turn off
   PTE_PG bit
 * @param userPageVAddr
 * @param pagePAddr
 * @param pgdir
 */
void fixPagedInPTE(int userPageVAddr, int pagePAddr, pde_t * pgdir){
    ...
}

/**
 *
 * @param userPageVAddr
 * @param pgdir
 * @return
 */
```

```c
int pageIsInFile(int userPageVAddr, pde_t * pgdir) {
    pte_t *pte;
    pte = walkpgdir(pgdir, (char *)userPageVAddr, 0);
    return (*pte & PTE_PG); //PAGE IS IN FILE
}

/**
 *
 * @return
 */
int getLIFO(){
    int i;
    int pageIndex = -1;
    uint loadOrder = 0;

    for (i = 0; i < MAX_PYSC_PAGES; i++) {
        if (myproc()->ramCtrlr[i].state == USED && myproc()->ramCtrlr[i].loadOrder > loadOrder) {
            loadOrder = myproc()->ramCtrlr[i].loadOrder;
            pageIndex = i;
        }
    }
    return pageIndex;
}

/**
 *
 * @return
 */
int getSCFIFO(){
    ...
}

/**
 *
 * @return
 */
int getLAP(){
    int i;
    int pageIndex = -1;
    uint minAccess = 0xffffffff;

    for (i = 0; i < MAX_PYSC_PAGES; i++) {
        if (myproc()->ramCtrlr[i].state == USED && myproc()->ramCtrlr[i].accessCount <= minAccess) {
            minAccess = myproc()->ramCtrlr[i].accessCount;
            pageIndex = i;
        }
    }
    return pageIndex;
}

/**
 *
 * @return
 */
int getPageOutIndex(){
#if LIFO
    return getLIFO();
#endif
#if SCFIFO
    return getSCFIFO();
#endif
#if LAP
    return getLAP();
#endif
    panic("Unrecognized paging machanism");
}

/**
 *
 * @param p
```

```
 */
void updateAccessCounters(struct proc * p){
 ...
}

/**
 *
 * @return
 */
int getFreeRamCtrlrIndex() {
    if (myproc() == 0)
        return -1;
    int i;
    for (i = 0; i < MAX_PYSC_PAGES; i++) {
        if (myproc()->ramCtrlr[i].state == NOTUSED)
            return i;
    }
    return -1; //NO ROOM IN RAMCTRLR
}

static char buff[PGSIZE]; //buffer used to store swapped page in getPageFromFile
    method

/**
 *
 * @param cr2
 * @return
 */
int getPageFromFile(int cr2){
...
}

/**
 *
 * @param pgdir
 * @param userPageVAddr
 */
void addToRamCtrlr(pde_t *pgdir, uint userPageVAddr) {
...
}

/**
 *
 * @param pgdir
 * @param userPageVAddr
 */
void swap(pde_t *pgdir, uint userPageVAddr){
...
}

/**
 *
 * @return
 */
int isNONEpolicy(){
#if NONE
    return 1;
#endif
    return 0;
}

 /**
 *
 * @param userPageVAddr
 * @param pgdir
 */
void removeFromRamCtrlr(uint userPageVAddr, pde_t *pgdir){
...
}

/**
 *
```

```
 * @param userPageVAddr
 * @param pgdir
 */
void removeFromFileCtrlr(uint userPageVAddr, pde_t *pgdir){
...
}


```

2.5 Miscellenous

- The functions `isdireempty` and `create` in `sysfile.c` were made non static and added to `defs.h` for access to members in other files
- Added Page Fault conditions in `trap.c`
- Added `myMemTest.c` in Makefile
- Added `myMemTest1.c` with additional tests
- Added additional selection macros in Makefile
- Added definitions in `proc.h`
- Added bits flags in `mmu.h`

3 myMemTest.c

3.1 Fork Test

```
 *
  Test used to check the swapping machanism in fork.
  Best tested when LIFO is used (for more swaps)
*/
void forkTest(){
  int i;
  char * arr;
  arr = malloc (50000); //allocates 13 pages (sums to 16), in lifo, OS puts page
    #15 in file.

  for (i = 0; i < 50; i++) {
    arr[49100+i] = 'A'; //last six A's stored in page #16, the rest in #15
    arr[45200+i] = 'B'; //all B's are stored in page #15.
  }
  arr[49100+i] = 0; //for null terminating string...
  arr[45200+i] = 0;

  if (fork() == 0){ //is son
    for (i = 40; i < 50; i++) {
      arr[49100+i] = 'C'; //changes last ten A's to C
      arr[45200+i] = 'D'; //changes last ten B's to D
    }
    printf(1, "SON: %s\n",&arr[49100]); // should print AAAAA..CCC...
    printf(1, "SON: %s\n",&arr[45200]); // should print BBBBB..DDD...
    printf(1,"\n");
    free(arr);
    exit();
  } else { //is parent
    wait();
    printf(1, "PARENT: %s\n",&arr[49100]); // should print AAAAA...
    printf(1, "PARENT: %s\n",&arr[45200]); // should print BBBBB...
    free(arr);
  }
}
/**
* The Output of the above test is as follows:
SON: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAACCCCCCCCCC
```

```
SON: BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBDDDDDDDDDD

PARENT: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
PARENT: BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB

* Hence the test is successful
*/

```

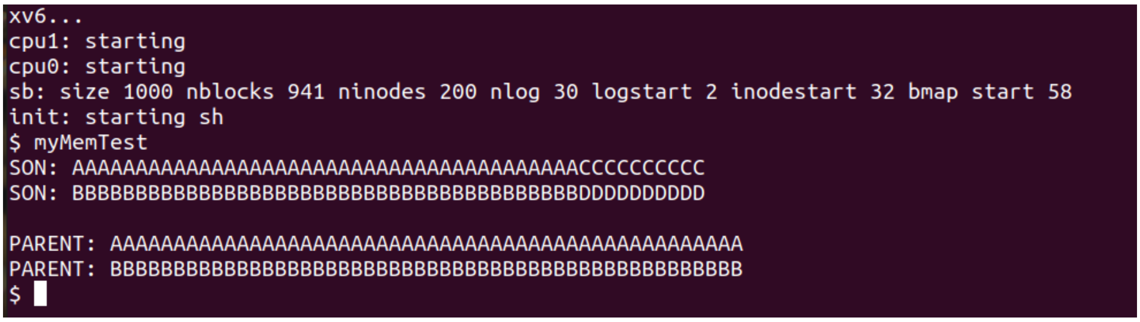

Figure 1: myMemTest output

3.2 Global protocols test

```
/*
Global Test:
Allocates 17 pages (1 code, 1 space, 1 stack, 14 malloc)
Using pseudoRNG to access a single cell in the array and put a number in it.
Idea behind the algorithm:
  Space page will be swapped out sooner or later with scfifo or lap.
  Since no one calls the space page, an extra page is needed to play with swapping
    (hence the #17).
  We selected a single page and reduced its page calls to see if scfifo and lap
    will become more efficient.
*/
void globalTest(){
  char * arr;
  int i;
  int randNum;
  arr = malloc(ARR_SIZE); //allocates 14 pages (sums to 17 - to allow more then one
     swapping in scfifo)
  for (i = 0; i < TEST_POOL; i++) {
    randNum = getRandNum(); //generates a pseudo random number between 0 and
    ARR_SIZE
     while (PGSIZE*10-8 < randNum && randNum < PGSIZE*10+PGSIZE/2-8)
       randNum = getRandNum(); //gives page #13 50% less chance of being selected
                               //(redraw number if randNum is in the first half of
    page #13)
     arr[randNum] = 'X';       //write to memory
  }
  free(arr);
}
/**
* The Output of the above test is as follows:
Results (for TEST_POOL = 500):
LIFO: 42 Page faults
LAP: 18 Page faults
SCFIFO: 35 Page faults
* Hence the test is successful
*/

```

4 myMemTest1.c

Here we have additional tests

4.1 SCFIFO test

```
int i, j;
  char *arr[14];
  char input[10];

  // TODO delete
  printf(1, "myMemTest: testing SCFIFO... \n");

  // Allocate all remaining 12 physical pages
  for (i = 0; i < 12; ++i) {
    arr[i] = sbrk(PGSIZE);
    printf(1, "arr[%d]=0x%x\n", i, arr[i]);
  }
  printf(1, "Called sbrk(PGSIZE) 12 times - all physical pages taken.\nPress any key...\n");
  gets(input, 10);

  /*
  Allocate page 15.
  For this allocation, SCFIFO will consider moving page 0 to disk, but because it has been accessed, page 1 will be moved instead.
  Afterwards, page 1 is in the swap file, the rest are in memory.
  */
  arr[12] = sbrk(PGSIZE);
  printf(1, "arr[12]=0x%x\n", arr[12]);
  printf(1, "Called sbrk(PGSIZE) for the 13th time, no page fault should occur and one page in swap file.\nPress any key...\n");
  gets(input, 10);

  /*
  Allocate page 16.
  For this allocation, SCFIFO will consider moving page 2 to disk, but because it has been accessed, page 3 will be moved instead.
  Afterwards, pages 1 & 3 are in the swap file, the rest are in memory.
  */
  arr[13] = sbrk(PGSIZE);
  printf(1, "arr[13]=0x%x\n", arr[13]);
  printf(1, "Called sbrk(PGSIZE) for the 14th time, no page fault should occur and two pages in swap file.\nPress any key...\n");
  gets(input, 10);

  /*
  Access page 3, causing a PGFLT, since it is in the swap file. It would be
  hot-swapped with page 4. Page 4 is accessed next, so another PGFLT is invoked,
  and this process repeats a total of 5 times.
  */
  for (i = 0; i < 5; i++) {
    for (j = 0; j < PGSIZE; j++)
      arr[i][j] = 'k';
  }
  printf(1, "5 page faults should have occurred.\nPress any key...\n");
  gets(input, 10);

  /*
  If DEBUG flag is defined as != 0 this is just another example showing
  that because SCFIFO doesn't page out accessed pages, no needless page faults occurr.
  */
  if(DEBUG){
    for (i = 0; i < 5; i++) {
      printf(1, "Writing to address 0x%x\n", arr[i]);
      arr[i][0] = 'k';
    }
    //printf(1, "No page faults should have occurred.\nPress any key...\n");
    gets(input, 10);
```

```c
  }

  if (fork() == 0) {
    printf(1, "Child code running.\n");
    printf(1, "View statistics for pid %d, then press any key...\n", getpid());
    gets(input, 10);

    /*
    The purpose of this write is to create a PGFLT in the child process, and
    verify that it is caught and handled properly.
    */
    arr[5][0] = 'k';
    printf(1, "A Page fault should have occurred in child proccess.\nPress any key
    to exit the child code.\n");
    gets(input, 10);

    exit();
  }
  else {
    wait();

    /*
    Deallocate all the pages.
    */
    sbrk(-14 * PGSIZE);
    printf(1, "Deallocated all extra pages.\nPress any key to exit the father code
    .\n");
    gets(input, 10);
  }

```

The result is shown below

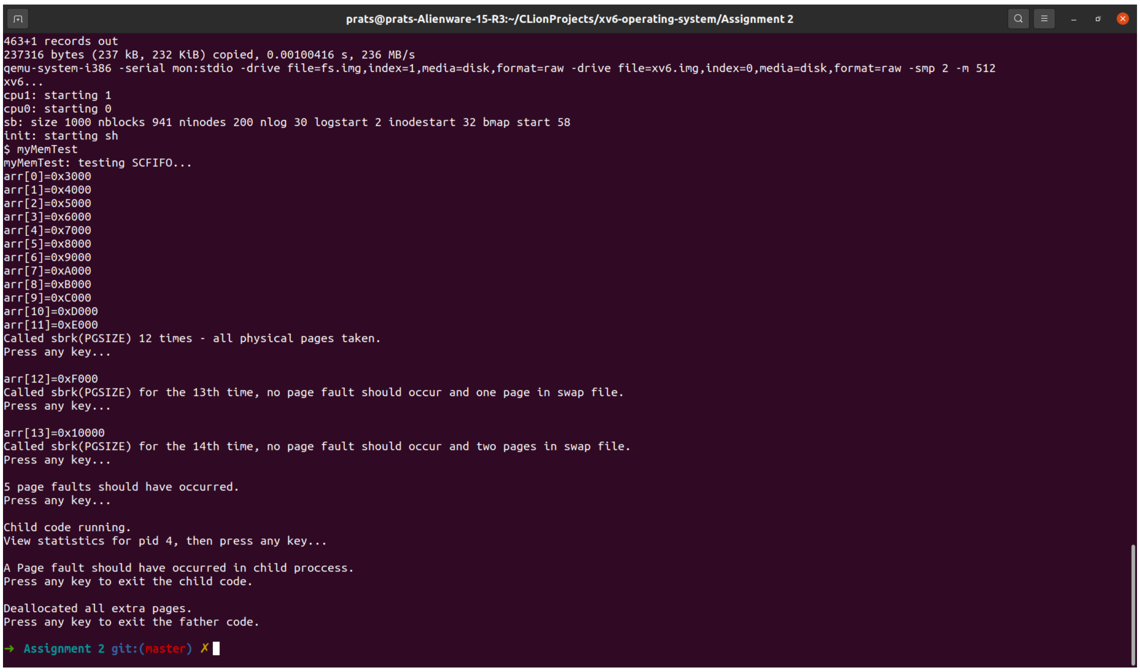

Figure 2: Extra SCFIFO test output