ELL783 - Operating Systems

Assignment 2 Report

Link to GitHub Repository

Sunav Kumar Vidhyarthi

2017ME10618

To start with, we wish to implement a virtual memory management scheme in this assignment. We will put a limit on the number of pages which can exist for a process in the physical memory first. More than that, the pages will be added to the swapFile. Second, we will implement FIFO, SCFIFO & NFU schemes for retrieving pages from the swapFile. Finally, we write the *myMemTest.c* file to test our implementation of the infrastructure.

1. Paging Framework

In this part we implement the paging infrastructure for xv-6 which involves storing pages in the swapFile of the process if more than 15 pages are assigned to the process.

1.1. Supplied File Framework

The supplied file framework can be accessed from the first commit of the repository.

1.2. Storing Pages in File

As instructed, we define the two macros MAX_TOTAL_PAGES and MAX_PSYC_PAGES in the *proc.h* file. *proc.h* file contains the process control block of the processes in xv-6.

```
#define MAX_PSYC_PAGES 15
#define MAX_TOTAL_PAGES 30
```

We enrich the same file with process metadata required for maintaining page counts, the *swapFile* and the information about presence of pages in the physical memory and the *swapFile*. The following snippet shows the *freePage{} struct* and the metadata in *proc struct.*

```
struct freePage {
  char *virtualAddress;
  int age;
  struct freePage *next;
  struct freePage *prev;
  uint swaploc;
};

struct proc {
  ...
  struct file *swapFile;

  struct freePage freePages[MAX_PSYC_PAGES];
  struct freePage swappedPages[MAX_PSYC_PAGES];
  struct freePage *head;
  struct freePage *tail;

  int mainMemoryPageCount;
  int totalSwapCount;
  int pageFaultCount;
  int totalSwapCount;
}
```

freePage{} is the structure for node of linked list of pages which are present in the main memory and the swap space. In the *proc* structure, we include the arrays of type *freePage{}* for both main memory and physical memory pages. This constitutes the list for main memory and swapped pages. Other metadata structures for keeping a count of number of page swaps, pages in main memory and swapped file, and page faults.

We use a flag in *mmu.h* file to signify removal of a page from the *swapFile*. We clear the PTE_P flag and establish the new PTE_PG flag.

```
#define PTE_PG          0x200   // Paged out to secondary storage
```

The following function sets the PTE_PG flag and clears the PTE_P flag in the **vm.c** file.

```
int checkAccesedBit(char *va){
  struct proc *proc = myproc();
  uint flag;
  pte_t *pte
  pte = walkpgdir(proc->pgdir,(void*)va,0);

  if(!pte)
    panic("checkAccesedBit: pte not found");

  flag = (*pte) & PTE_A;
  (*pte) &= ~PTE_A;
  return flag;
}
```

Paging is done and the corresponding pages are stored in the page table of respective processes. The address of the process's pages can be found in that table. The pointer to the head of the current process can be found in the CR3 register. As default state of affairs, xv-6 allocates memory without any constraint on the maximum number of pages which could be in the main memory. This assignment is done in the *proc.c* file. Hence, we put a limit in the loop in the *proc.c* file in *allocuvm()* function. These pages are assigned in the *allocuvm()* function. Similarly, *deallocuvm()* is modified to free the metadata associated with the currently running process.

1.3. Retrieving Pages on Demand

If there is a page fault, we need to confirm by issuing a check under the T_PGFLT flag in the *trap.c* file. This will check if the page causing page fault exist in the swapFile of the process or not. If it exists, the page needs to be retrieved and the page fault must be relieved. *swapPages()* receives the address of the page causing the page fault, updates the metadata of the process and swap the required page from the swapFile to main memory of the process. Which page is swapped against the page in the swapFile is decided from the scheme chosen for retrieving pages during the make? By default, the system will use the NFU strategy. Main piece of code reflecting the above changes in the *trap.c* file is shown below.

```
void
trap(struct trapframe *tf)
{
...
 case T_PGFLT:
    addr = rcr2();
    vaddr = &(myproc()->pgdir[PDX(addr)]);
    if(((int)(*vaddr) & PTE_P)!=0){
      if(((uint*)PTE_ADDR(P2V(*vaddr)))[PTX(addr)]&PTE_PG){
        swapPages(PTE_ADDR(addr));
        ++myproc()->pageFaultCount;
        return;
      }
    }
...
}
```

1.4. Comprehensive Changes in Existing Parts

Changes in the *fork()* is done to ensure that the newly forked process must carry the pages in its parent's *swapFile.* All the metadata in Section 1.2, except the *pageFaultCount* and *swapPageCount* needs to be copied as well. This is done to enable **copy-on-write**. Main piece of code reflecting the above changes is shown below.

```
#ifndef NONE
    if(curproc->pid > 2){
      createSwapFile(np);
      char buf[PGSIZE/2] = "";
      int offset = 0;
      int nread = 0;
      while((nread == readFromSwapFile(curproc,buf, offset, PGSIZE/2))!=0)
        if(writeToSwapFile(np, buf, offset, nread) == -1){
          panic("fork: could not copy parent's swapFile");
        offset +=nread;
      }
    }

    for(i=0;i<MAX_PSYC_PAGES;i++){

      np->freePages[i].virtualAddress = curproc->freePages[i].virtualAddress;
      np->freePages[i].age = curproc->freePages[i].age;
      np->swappedPages[i].virtualAddress = curproc-
>swappedPages[i].virtualAddress;
      np->swappedPages[i].age = curproc->swappedPages[i].age;
      np->swappedPages[i].swaploc = curproc->swappedPages[i].swaploc;

    }
    ...
}
```

2. Page Replacement Schemes

A selection macro is added in the makefile as below.

```
ifndef SELECTION
    SELECTION := NFU
```

```
endif
```

Default SELECTION macro is set as NFU.

If **FIFO** scheme is used, not much is to be done as the *proc->head* already stores the head or the first page in the array which can we swapped with the page in the swapped space.

```
#elif FIFO
  ...
  while(i<MAX_PSYC_PAGES){
    if(proc->swappedPages[i].virtualAddress == (char*)0xffffffff){
      ...
      int num = writeToSwapFile(proc,(char*)PTE_ADDR(last-
>virtualAddress),i*PGSIZE, PGSIZE);
      ...
      pte_t *pte1 = walkpgdir(proc->pgdir, (void*)last->virtualAddress, 0);
      ...
      kfree((char*)PTE_ADDR(P2V_WO(*walkpgdir(proc->pgdir, last-
>virtualAddress, 0))));
      *pte1 = PTE_W | PTE_U | PTE_PG;
      ++proc->swapPageCount;
      ++proc->totalSwapCount;
      ...
    }
    ...
  }
```

If **SCFIFO** scheme is used, the we traverse the pages and send those pages to the end whose PTE_A flag is not set. PTE_A flag signifies if the page has been accessed or not. If the flag is not set the page is chosen for replacement.

```
#elif SCFIFO
  ...
  while(i<MAX_PSYC_PAGES){
    if(proc->swappedPages[i].virtualAddress == (char*)0xffffffff){
      ...
      int num = writeToSwapFile(proc, (char*)PTE_ADDR(proc->head-
>virtualAddress), i * PGSIZE, PGSIZE);
      ...
      pte_t *pte1 = walkpgdir(proc->pgdir, (void*)proc->head-
>virtualAddress, 0);
      ...
      kfree((char*)PTE_ADDR(P2V_WO(*walkpgdir(proc->pgdir, proc->head-
>virtualAddress, 0))));
      *pte1 = PTE_W | PTE_U | PTE_PG;
      ++proc->swapPageCount;
      ++proc->totalSwapCount;
      ...
    }
    ...
  }
```

In **NFU**, we modify the *trap.c* file to call *updateNFUState()* function such that the function is called whenever a timer interrupt is issued.

```
switch(tf->trapno){
  case T_IRQ0 + IRQ_TIMER:
    if(cpuid() == 0){
      acquire(&tickslock);
      #if NFU
        updateNFUState();
      #endif
      ticks++;
      wakeup(&ticks);
      release(&tickslock);
    }
    lapiceoi();
    break;
    ...
}
```

updateNFUState() increment the age of all the pages and resets if the page is accessed recently.

```
void updateNFUState(){
  ...
  for(p = ptable.proc; p< &ptable.proc[NPROC]; p++){
    if((p->state == RUNNING || p->state == RUNNABLE || p->state == SLEEPING)){
      for(i=0; i<MAX_PSYC_PAGES; i++){
        if(p->freePages[i].virtualAddress == (char*)0xffffffff)
          continue;
        p->freePages[i].age++;
        p->swappedPages[i].age++;
        ...
      }
    }
  }
  ...
}
```

In *vm.c* the page having the maximum age is swapped.

```
while(i<MAX_PSYC_PAGES){
    if(proc->swappedPages[i].virtualAddress == (char*)0xffffffff){
      for(j=0; j<MAX_PSYC_PAGES; j++){
        if(proc->freePages[j].virtualAddress != (char*)0xffffffff){
          if(proc->freePages[j].age > maxAge){
              maxAge = proc->freePages[j].age;
              maxIndex = j;
            }
        }
      }
      ...
      pte_t *pte1 = walkpgdir(proc->pgdir, (void*)candidate-
>virtualAddress, 0);
      ...
      int test_num = writeToSwapFile(proc, (char*)PTE_ADDR(candidate-
>virtualAddress), i * PGSIZE, PGSIZE);
      ...
      kfree((char*)PTE_ADDR(P2V_WO(*walkpgdir(proc->pgdir, candidate-
>virtualAddress, 0))));
      *pte1 = PTE_W | PTE_U | PTE_PG;
      proc->swapPageCount+=1;
      proc->totalSwapCount+=1;
      ...
    }
    ...
  }
```

3. Testing

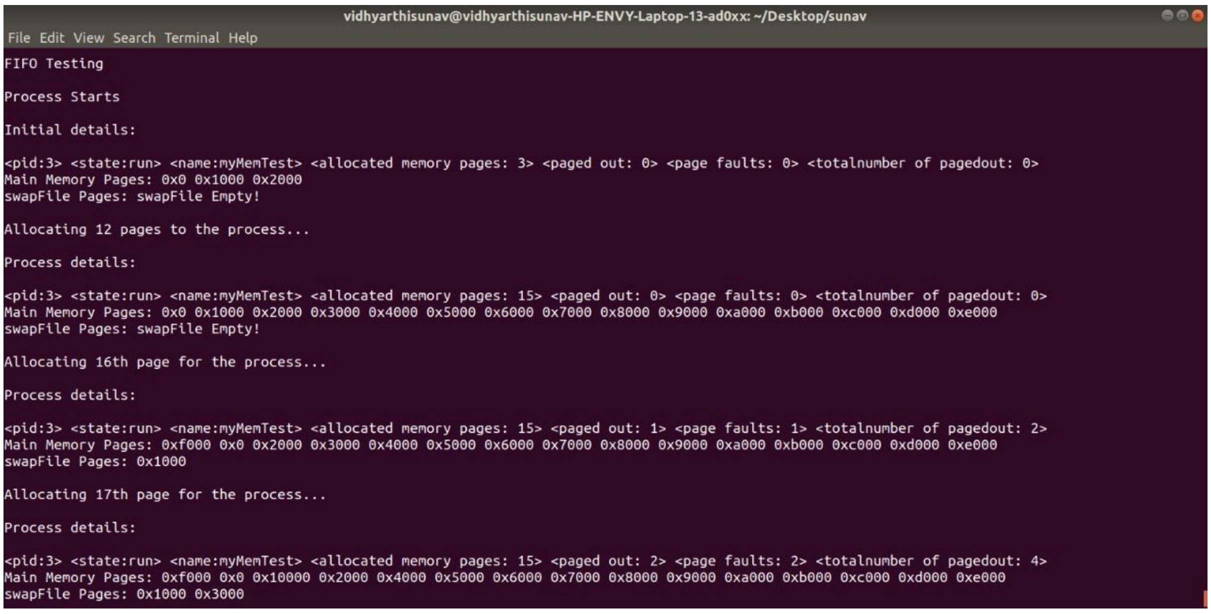
We write the *printCurrentProcessDetails()* and update the *procdump()* functions for the **Enhanced Process Details Viewer.** The following code snippets show implementation of these two functions.

```
void printProcessDetails(struct proc *proc)
{
  ...
  cprintf("\n<pid:%d> <state:%s> <name:%s> ", proc->pid, state, proc->name);
  cprintf("<allocated memory pages: %d> ", proc->mainMemoryPageCount);
  cprintf("<paged out: %d> ", proc->swapFilePageCount);
  cprintf("<page faults: %d> ", proc->pageFaultCount);
  cprintf("<totalnumber of pagedout: %d>\n", proc->totalSwapCount);
  ...
}
```

```
void
procdump(void)
{
  ...
  for(p = ptable.proc; p<&ptable.proc[NPROC]; p++){
    if(p->state == UNUSED)
      continue;
    printCurrentProcessDetails(p);
  }
  percentage = (freePageCounts.currentFreePages*100)/freePageCounts.initFreePa
ges;
  cprintf("\n\n Number of free physical pages: %d/%d ~ %d%% \n",freePageCounts
.currentFreePages,freePageCounts.initFreePages, percentage);
}
```

3.1. Sanity Check

FIFO Testing


vidhyarthisunav@vidhyarthisunav-HP-ENVY-Laptop-13-ad0xx: ~/Desktop/sunav
File Edit View Search Terminal Help

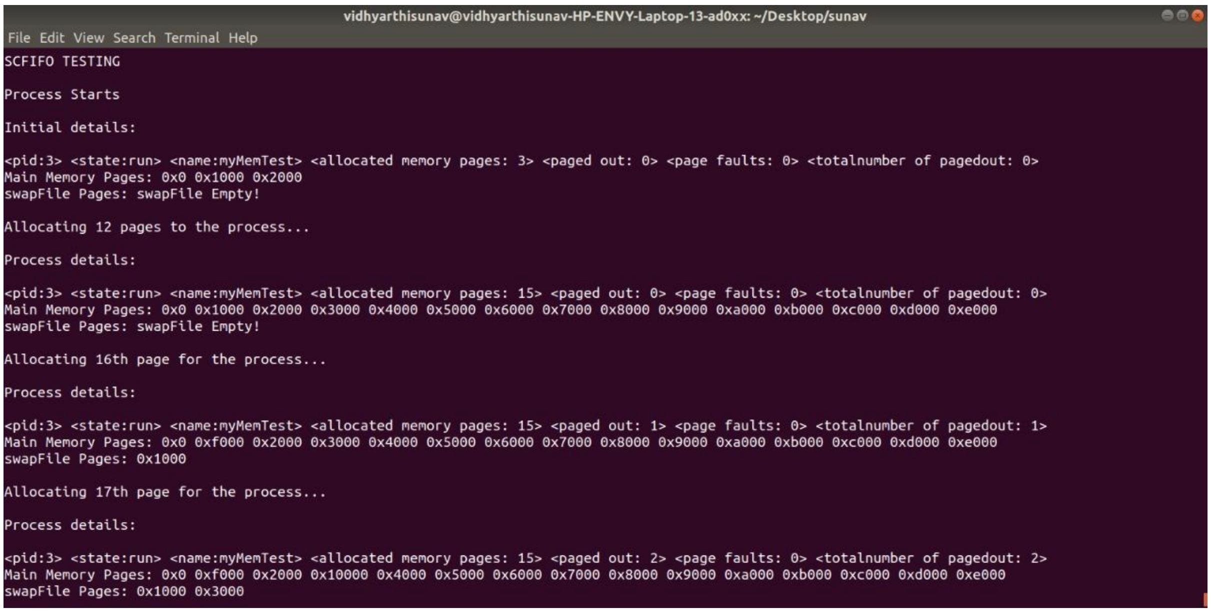
```
FIFO Testing

Process Starts

Initial details:

<pid:3> <state:run> <name:myMemTest> <allocated memory pages: 3> <paged out: 0> <page faults: 0> <totalnumber of pagedout: 0>
Main Memory Pages: 0x0 0x1000 0x2000
swapFile Pages: swapFile Empty!

Allocating 12 pages to the process...

Process details:

<pid:3> <state:run> <name:myMemTest> <allocated memory pages: 15> <paged out: 0> <page faults: 0> <totalnumber of pagedout: 0>
Main Memory Pages: 0x0 0x1000 0x2000 0x3000 0x4000 0x5000 0x6000 0x7000 0x8000 0x9000 0xa000 0xb000 0xc000 0xd000 0xe000
swapFile Pages: swapFile Empty!

Allocating 16th page for the process...

Process details:

<pid:3> <state:run> <name:myMemTest> <allocated memory pages: 15> <paged out: 1> <page faults: 1> <totalnumber of pagedout: 2>
Main Memory Pages: 0xf000 0x0 0x2000 0x3000 0x4000 0x5000 0x6000 0x7000 0x8000 0x9000 0xa000 0xb000 0xc000 0xd000 0xe000
swapFile Pages: 0x1000

Allocating 17th page for the process...

Process details:

<pid:3> <state:run> <name:myMemTest> <allocated memory pages: 15> <paged out: 2> <page faults: 2> <totalnumber of pagedout: 4>
Main Memory Pages: 0xf000 0x0 0x10000 0x2000 0x4000 0x5000 0x6000 0x7000 0x8000 0x9000 0xa000 0xb000 0xc000 0xd000 0xe000
swapFile Pages: 0x1000 0x3000
```

Initially the process is assigned three pages. We push 12 more pages and that makes allocated memory page count 15. At this point, all the pages lie in the main memory and the swapFile is empty.

We add 16th page and see the FIFO infrastructure working. As one more page is added, the page is swapped with the 1st page (0x0). So, 0xf000 lies on the first place. But then, there is a page fault for page 0x0 page. So, it is swapped with the 2nd page which is 0x1000. Hence, 0x1000 is in the swapFile and 0x0 is on the second place.

Now we add 17th page. 17th page is swapped with the 3rd page, i.e., 0x2000. So, 0x10000 (the 17th page) appears on the third place. But then, there is a page fault for the 3rd page because it contains user data. Consequently, 3rd page is swapped for the 4th page (which is empty). swapFile now contains the 2nd page (0x1000) and the 4th page (0x3000). Total page faults are 2. Total swap outs is 2 and total number of swaps made is 4.

SCFIFO Testing


vidhyarthisunav@vidhyarthisunav-HP-ENVY-Laptop-13-ad0xx: ~/Desktop/sunav
File Edit View Search Terminal Help

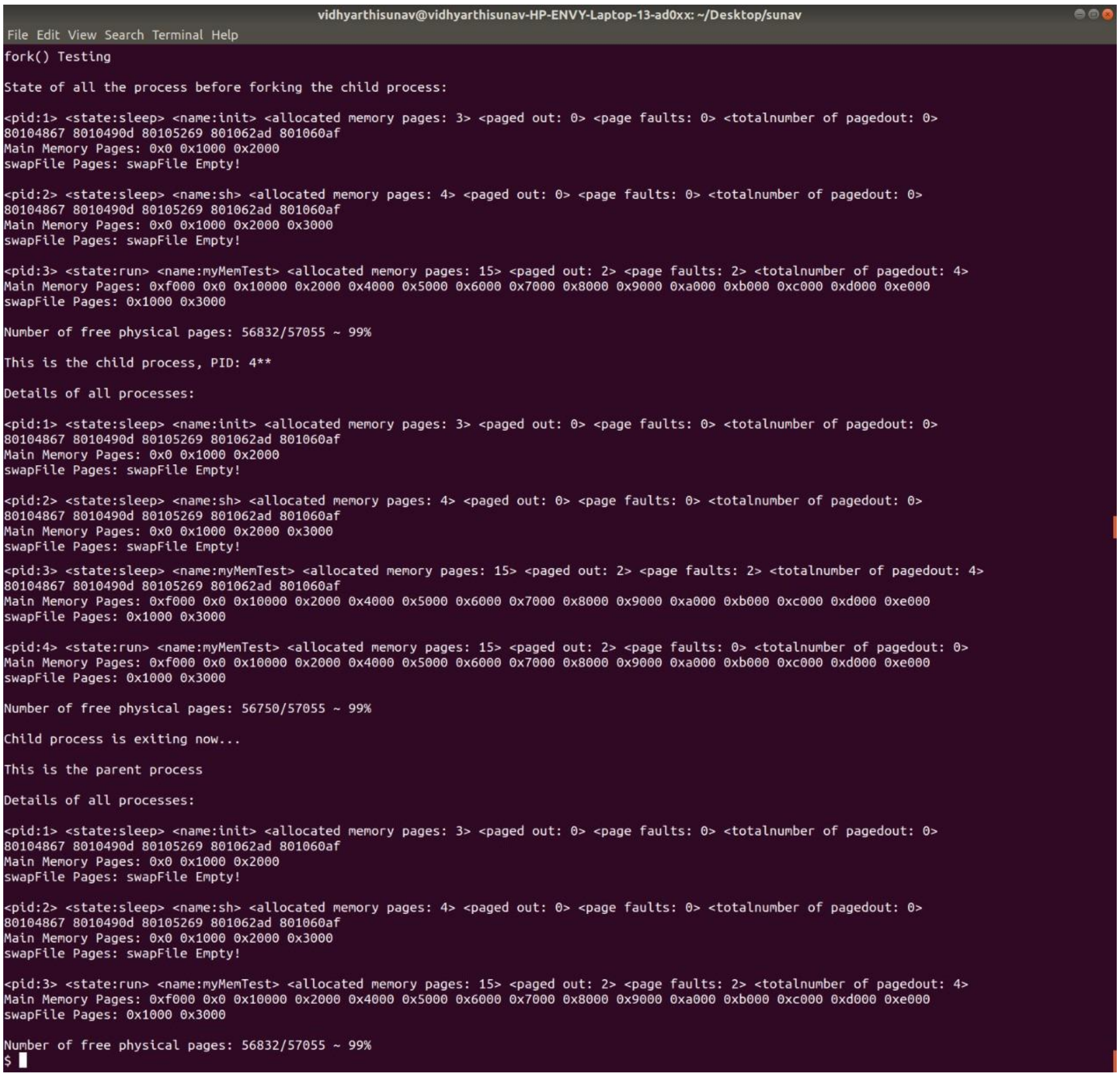
```
SCFIFO TESTING

Process Starts

Initial details:

<pid:3> <state:run> <name:myMemTest> <allocated memory pages: 3> <paged out: 0> <page faults: 0> <totalnumber of pagedout: 0>
Main Memory Pages: 0x0 0x1000 0x2000
swapFile Pages: swapFile Empty!

Allocating 12 pages to the process...

Process details:

<pid:3> <state:run> <name:myMemTest> <allocated memory pages: 15> <paged out: 0> <page faults: 0> <totalnumber of pagedout: 0>
Main Memory Pages: 0x0 0x1000 0x2000 0x3000 0x4000 0x5000 0x6000 0x7000 0x8000 0x9000 0xa000 0xb000 0xc000 0xd000 0xe000
swapFile Pages: swapFile Empty!

Allocating 16th page for the process...

Process details:

<pid:3> <state:run> <name:myMemTest> <allocated memory pages: 15> <paged out: 1> <page faults: 0> <totalnumber of pagedout: 1>
Main Memory Pages: 0x0 0xf000 0x2000 0x3000 0x4000 0x5000 0x6000 0x7000 0x8000 0x9000 0xa000 0xb000 0xc000 0xd000 0xe000
swapFile Pages: 0x1000

Allocating 17th page for the process...

Process details:

<pid:3> <state:run> <name:myMemTest> <allocated memory pages: 15> <paged out: 2> <page faults: 0> <totalnumber of pagedout: 2>
Main Memory Pages: 0x0 0xf000 0x2000 0x10000 0x4000 0x5000 0x6000 0x7000 0x8000 0x9000 0xa000 0xb000 0xc000 0xd000 0xe000
swapFile Pages: 0x1000 0x3000
```

Initially the process is assigned three pages. We push 12 more pages and that makes allocated memory page count 15. At this point, all the pages lie in the main memory and the swapFile is empty.

We add 16th page and see the SCFIFO infrastructure working. As one more page is added, the page is supposed to be swapped with the 1st page (0x0), but as it's been recently used, it is swapped with second page instead. So, 0xf000 lies on the second place and the 2nd page (0x1000) is in swapFile. There is only one page out.

Now we add 17th page. 17th page is supposed to be swapped with the 3rd page, i.e., 0x2000. But again, as it has been recently accessed, 4th page (0x3000) is swapped and is contained with the 0x1000 page in the swapFile. There is no page fault and 2 page outs has been performed.

fork() Testing



```
vidhyarthisunav@vidhyarthisunav-HP-ENVY-Laptop-13-ad0xx: ~/Desktop/sunav
File Edit View Search Terminal Help
fork() Testing

State of all the process before forking the child process:

<pid:1> <state:sleep> <name:init> <allocated memory pages: 3> <paged out: 0> <page faults: 0> <totalnumber of pagedout: 0>
80104867 8010490d 80105269 801062ad 801060af
Main Memory Pages: 0x0 0x1000 0x2000
swapFile Pages: swapFile Empty!

<pid:2> <state:sleep> <name:sh> <allocated memory pages: 4> <paged out: 0> <page faults: 0> <totalnumber of pagedout: 0>
80104867 8010490d 80105269 801062ad 801060af
Main Memory Pages: 0x0 0x1000 0x2000 0x3000
swapFile Pages: swapFile Empty!

<pid:3> <state:run> <name:myMemTest> <allocated memory pages: 15> <paged out: 2> <page faults: 2> <totalnumber of pagedout: 4>
Main Memory Pages: 0xf000 0x0 0x10000 0x2000 0x4000 0x5000 0x6000 0x7000 0x8000 0x9000 0xa000 0xb000 0xc000 0xd000 0xe000
swapFile Pages: 0x1000 0x3000

Number of free physical pages: 56832/57055 ~ 99%

This is the child process, PID: 4**

Details of all processes:

<pid:1> <state:sleep> <name:init> <allocated memory pages: 3> <paged out: 0> <page faults: 0> <totalnumber of pagedout: 0>
80104867 8010490d 80105269 801062ad 801060af
Main Memory Pages: 0x0 0x1000 0x2000
swapFile Pages: swapFile Empty!

<pid:2> <state:sleep> <name:sh> <allocated memory pages: 4> <paged out: 0> <page faults: 0> <totalnumber of pagedout: 0>
80104867 8010490d 80105269 801062ad 801060af
Main Memory Pages: 0x0 0x1000 0x2000 0x3000
swapFile Pages: swapFile Empty!
<pid:3> <state:sleep> <name:myMemTest> <allocated memory pages: 15> <paged out: 2> <page faults: 2> <totalnumber of pagedout: 4>
80104867 8010490d 80105269 801062ad 801060af
Main Memory Pages: 0xf000 0x0 0x10000 0x2000 0x4000 0x5000 0x6000 0x7000 0x8000 0x9000 0xa000 0xb000 0xc000 0xd000 0xe000
swapFile Pages: 0x1000 0x3000

<pid:4> <state:run> <name:myMemTest> <allocated memory pages: 15> <paged out: 2> <page faults: 0> <totalnumber of pagedout: 0>
Main Memory Pages: 0xf000 0x0 0x10000 0x2000 0x4000 0x5000 0x6000 0x7000 0x8000 0x9000 0xa000 0xb000 0xc000 0xd000 0xe000
swapFile Pages: 0x1000 0x3000

Number of free physical pages: 56750/57055 ~ 99%

Child process is exiting now...

This is the parent process

Details of all processes:

<pid:1> <state:sleep> <name:init> <allocated memory pages: 3> <paged out: 0> <page faults: 0> <totalnumber of pagedout: 0>
80104867 8010490d 80105269 801062ad 801060af
Main Memory Pages: 0x0 0x1000 0x2000
swapFile Pages: swapFile Empty!

<pid:2> <state:sleep> <name:sh> <allocated memory pages: 4> <paged out: 0> <page faults: 0> <totalnumber of pagedout: 0>
80104867 8010490d 80105269 801062ad 801060af
Main Memory Pages: 0x0 0x1000 0x2000 0x3000
swapFile Pages: swapFile Empty!

<pid:3> <state:run> <name:myMemTest> <allocated memory pages: 15> <paged out: 2> <page faults: 2> <totalnumber of pagedout: 4>
Main Memory Pages: 0xf000 0x0 0x10000 0x2000 0x4000 0x5000 0x6000 0x7000 0x8000 0x9000 0xa000 0xb000 0xc000 0xd000 0xe000
swapFile Pages: 0x1000 0x3000

Number of free physical pages: 56832/57055 ~ 99%
$
```

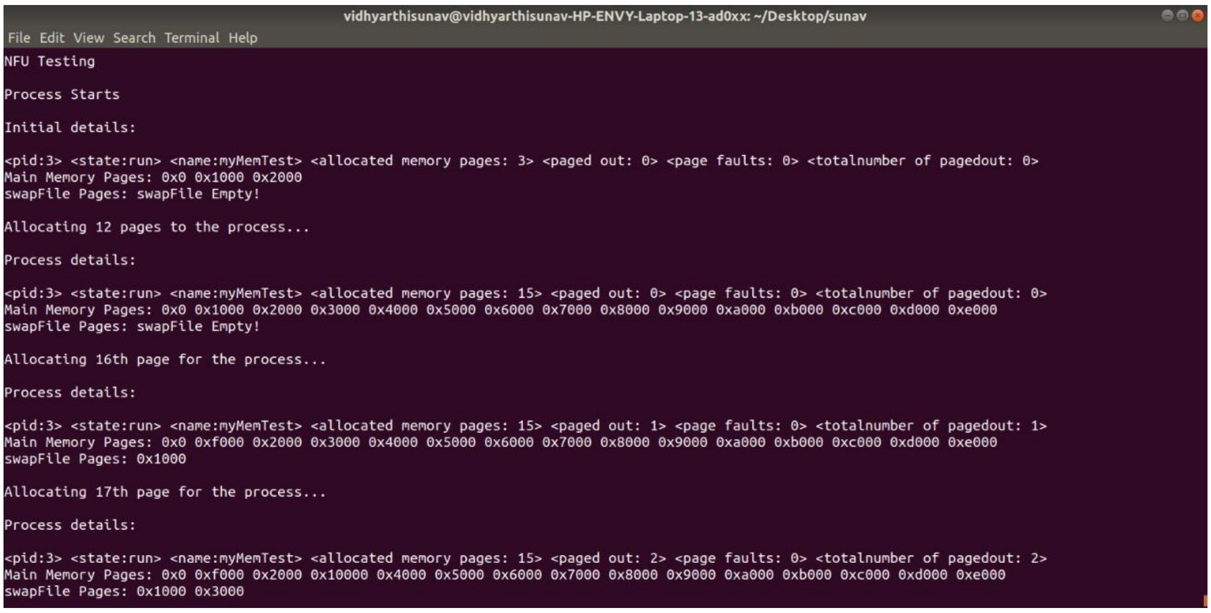
State of all the processes before forking the child process is given. myMemTest can be seen as a running process in the memory. myMemTest has 17 pages, 15 in the memory and 2 in the swap files. As of then, 2 page faults and total 4 swaps had been made (this is basically the myMemTest after the FIFO testing part).

As child process is being forked. Details of all the processes can be seen on the screen. The new child process can be seen copying everything, including the pages in swapFile and total number of pages outs.

Number of page faults and total swaps made can be seen excluded in the details of newly forked child process.

As the child process exits, the process with id 4 cannot be seen any more in the all-process details.

NFU Testing



vidhyarthisunav@vidhyarthisunav-HP-ENVY-Laptop-13-ad0xx: ~/Desktop/sunav
File Edit View Search Terminal Help

```
NFU Testing

Process Starts

Initial details:

<pid:3> <state:run> <name:myMemTest> <allocated memory pages: 3> <paged out: 0> <page faults: 0> <totalnumber of pagedout: 0>
Main Memory Pages: 0x0 0x1000 0x2000
swapFile Pages: swapFile Empty!

Allocating 12 pages to the process...

Process details:

<pid:3> <state:run> <name:myMemTest> <allocated memory pages: 15> <paged out: 0> <page faults: 0> <totalnumber of pagedout: 0>
Main Memory Pages: 0x0 0x1000 0x2000 0x3000 0x4000 0x5000 0x6000 0x7000 0x8000 0x9000 0xa000 0xb000 0xc000 0xd000 0xe000
swapFile Pages: swapFile Empty!

Allocating 16th page for the process...

Process details:

<pid:3> <state:run> <name:myMemTest> <allocated memory pages: 15> <paged out: 1> <page faults: 0> <totalnumber of pagedout: 1>
Main Memory Pages: 0x0 0xf000 0x2000 0x3000 0x4000 0x5000 0x6000 0x7000 0x8000 0x9000 0xa000 0xb000 0xc000 0xd000 0xe000
swapFile Pages: 0x1000

Allocating 17th page for the process...

Process details:

<pid:3> <state:run> <name:myMemTest> <allocated memory pages: 15> <paged out: 2> <page faults: 0> <totalnumber of pagedout: 2>
Main Memory Pages: 0x0 0xf000 0x2000 0x10000 0x4000 0x5000 0x6000 0x7000 0x8000 0x9000 0xa000 0xb000 0xc000 0xd000 0xe000
swapFile Pages: 0x1000 0x3000
```

Initially the process is assigned three pages. We push 12 more pages and that makes allocated memory page count 15. At this point, all the pages lie in the main memory and the swapFile is empty.

We add 16th page and see the NFU infrastructure working. As one more page is added, the page is supposed to be swapped with the page that is not been accessed the longest. The parameter for measuring this is being set in the variable age, which is updated by the updateNFUState() function. The page is page 2 (0x1000) and it is swapped with the 16th page (0xf000). but as it's been recently used, it is swapped with second page instead. So, 0xf000 lies on the place of 2nd page (0x1000) and 2nd page (0x1000) lies in the swapFile now. Hence, there is only one page out and no page fault.

Now we add 17th page. Again, the page that is not been accessed the longest will be selected by NFU. The page in this case is page 4 (0x3000). As it is swapped with incoming page (page 17th i.e., 0x10000), it is swapped with the 4th page and lies on its place. Hence, there are page 0x1000 and 0x3000 in the swapFile and the page 0xf000 and 0x10000 lies on their place respectively. This constitutes two pages outs and no page fault.