

2023 FALL OS Project 3 Part1

R12631055 林東甫

Motivation

```
Thread ../test/sort is executing.  
Thread ../test/matmult is executing.  
Unexpected user mode exception4  
Assertion failed: line 96 file ../userprog/exception.cc  
Aborted (core dumped)
```

In this project, we use demand paging to solve the limited memory issues. In a system that uses demand paging, the operating system copies a disk page into physical memory only if an attempt is made to access it and that page is not already in memory. To achieve this process a page table implementation is used. The page table maps logical memory to physical memory. The page table uses a bitwise operator to mark if a page is valid or invalid. A valid page is one that currently resides in main memory. An invalid page is one that currently resides in secondary memory. We will modify the following files to implement the demand paging system.

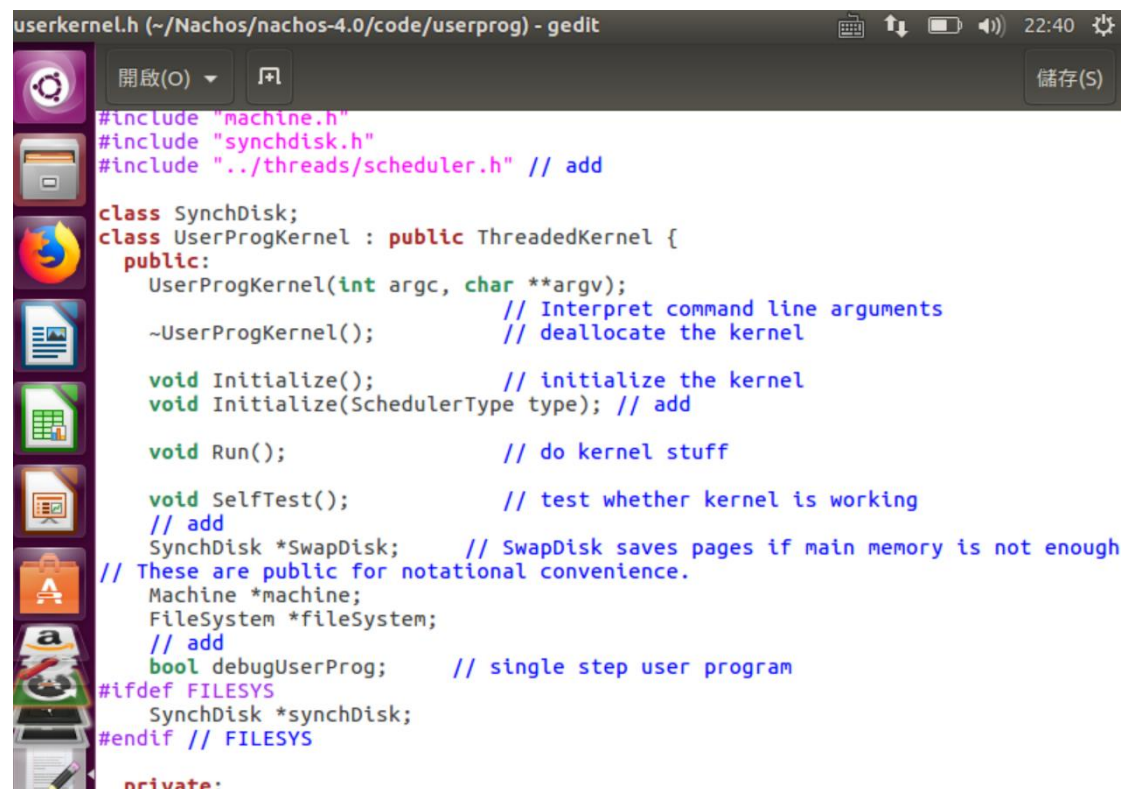
- /code/userprog/userkernel.*
- /code/machine/machine.*
- /code/userprog/addrspace.*
- /code/machine/translate.*

We implement page replacement algorithms, Least Recently Used (LRU).

Implementation

We first create a new SynchDisk called SwapDisk in /code/userprog/userkernel.h to simulate the secondary storage. Pages demanded by the process are swapped from secondary storage to main memory.

/code/userprog/userkernel.h

A screenshot of a gedit text editor window titled 'userkernel.h (~/.Nachos/nachos-4.0/code/userprog)'. The window shows the header file for the user kernel. It includes 'machine.h', 'synchdisk.h', and 'threads/scheduler.h'. It defines a 'SynchDisk' class and a 'UserProgKernel' class that inherits from 'ThreadedKernel'. The 'UserProgKernel' class has public methods for initialization, running, and self-testing. It also contains pointers to 'Machine', 'FileSystem', and 'SynchDisk' objects, and a 'debugUserProg' boolean. The 'FILESYS' section is commented out.

```
userkernel.h (~/.Nachos/nachos-4.0/code/userprog) - gedit
#include "machine.h"
#include "synchdisk.h"
#include "../threads/scheduler.h" // add

class SynchDisk;
class UserProgKernel : public ThreadedKernel {
public:
    UserProgKernel(int argc, char **argv);
    ~UserProgKernel(); // deallocate the kernel

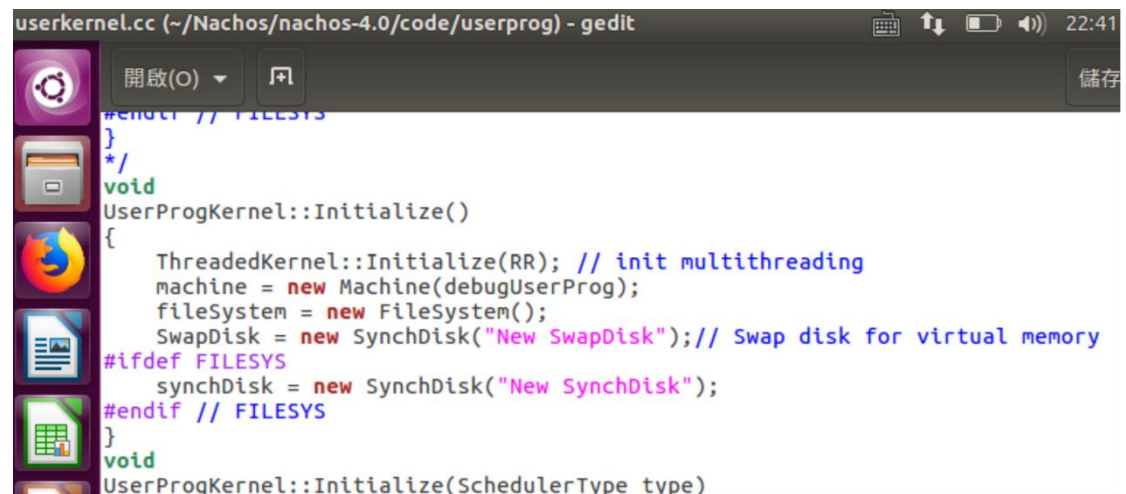
    void Initialize(); // initialize the kernel
    void Initialize(SchedulerType type); // add

    void Run(); // do kernel stuff

    void SelfTest(); // test whether kernel is working
    // add
    SynchDisk *SwapDisk; // SwapDisk saves pages if main memory is not enough
    // These are public for notational convenience.
    Machine *machine;
    FileSystem *fileSystem;
    // add
    bool debugUserProg; // single step user program
#ifdef FILESYS
    SynchDisk *synchDisk;
#endif // FILESYS

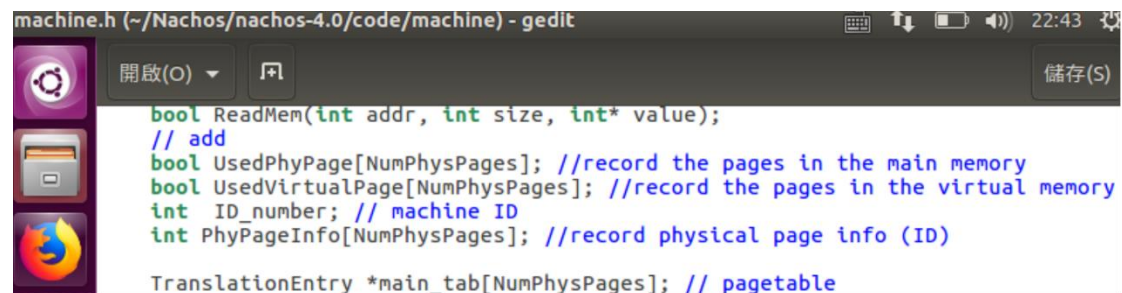
private:
```

Next, we initialized SwapDisk in /code/userprog/userkernel.cc.

A screenshot of a gedit text editor window titled 'userkernel.cc (~/.Nachos/nachos-4.0/code/userprog)'. The window shows the implementation of the 'UserProgKernel' class. It includes the 'FILESYS' section, which defines a 'SynchDisk' object named 'synchDisk' and initializes it with 'New SynchDisk'. The 'Initialize' method is implemented, calling 'ThreadedKernel::Initialize' and creating 'Machine', 'FileSystem', and 'SwapDisk' objects. The 'SwapDisk' object is initialized with 'New SwapDisk'. The 'FILESYS' section is commented out.

```
userkernel.cc (~/.Nachos/nachos-4.0/code/userprog) - gedit
#endif // FILESYS
}
*/
void
UserProgKernel::Initialize()
{
    ThreadedKernel::Initialize(RR); // init multithreading
    machine = new Machine(debugUserProg);
    fileSystem = new FileSystem();
    SwapDisk = new SynchDisk("New SwapDisk");// Swap disk for virtual memory
#ifdef FILESYS
    synchDisk = new SynchDisk("New SynchDisk");
#endif // FILESYS
}
void
UserProgKernel::Initialize(SchedulerType type)
```

We have to record the information about which frames of main and virtual memory are occupied, and the corresponding ID in /code/machine/machine.h.



```
machine.h (~/Nachos/nachos-4.0/code/machine) - gedit
開啟(O) 儲存(S)

bool ReadMem(int addr, int size, int* value);
// add
bool UsedPhyPage[NumPhysPages]; //record the pages in the main memory
bool UsedVirtualPage[NumPhysPages]; //record the pages in the virtual memory
int ID_number; // machine ID
int PhyPageInfo[NumPhysPages]; //record physical page info (ID)

TranslationEntry *main_tab[NumPhysPages]; // pagetable
```

Since we are handling virtual memory, the ASSERT to guarantee the number of pages does not exceed the number of physical pages in main memory in /code/userprog/addrspace.cc is no longer needed.

```
// remove
//ASSERT(numPages <= NumPhysPages);

// check we're not trying
// to run anything too big --
// at least until we have
// virtual memory
```

To implement virtual memory system. We modify the Load function in /code/userprog/addrspace.cc. The following for loop is used to find the

available space for the page of this process, and the while loop check whether the j-th frame is used. The index j will increase 1 is being used, until an empty frame or exceed the number of physical pages. There are two different cases in the following. When the main memory still have empty frame, then we can put the page into the main memory and update the information to the page table. This step is achieved by the function ReadAt. The other case will be the main memory is full. Then we have check the available virtual memory space by the similar while loop and write the page in to SwapDisk by the WriteSector function. For the Execute and SaveState function in /code/userprog/addrspace.cc, we add a flag in /code/userprog/addrspace.h to check whether the page table is successfully loaded to make the context-switch work.

```

for(unsigned int j=0,i=0;i < numPages ;i++){
    j=0;
    while(kernel->machine->UsedPhyPage[j]!=FALSE&&j<NumPhysPages){j++;}

    // main memory is enough, put the page to main memory
    if(j<NumPhysPages){
        kernel->machine->UsedPhyPage[j]=TRUE;
        kernel->machine->PhyPageInfo[j]=ID;
        kernel->machine->main_tab[j]=&pageTable[i];
        pageTable[i].physicalPage = j;
        pageTable[i].valid = TRUE;
        pageTable[i].use = FALSE;
        pageTable[i].dirty = FALSE;
        pageTable[i].readOnly = FALSE;
        pageTable[i].ID =ID;
        pageTable[i].LRU_counter++; // LRU counter when save in memory
        executable->ReadAt(&(kernel->machine->mainMemory
eSize)),PageSize, noffH.code.inFileAddr+(i*PageSize));
    }
    // main memory is not enough, use virtual memory
    else{
        char *buffer;
        buffer = new char[PageSize];
        tmp=0;
        while(kernel->machine->UsedVirtualPage[tmp]!=FALSE){tmp++;}
        kernel->machine->UsedVirtualPage[tmp]=true;
        pageTable[i].virtualPage=tmp; //record the virtual page we save
        pageTable[i].valid = FALSE; //not load in main memory
    }
}

```

Now the operating system is capable to put those pages that cannot put in main memory to virtual memory. In the following steps, we will implement page replacement algorithms, Least Recently Used (LRU) algorithms.

LRU

For the first page replacement algorithms, we implement Least Recently Used (LRU) algorithm. We implement by a hardware counter LRU_counter in /code/machine/traslate.h.

```
class TranslationEntry {
public:
    unsigned int virtualPage;    // The page number in virtual memory.
    unsigned int physicalPage;   // The page number in real memory (relative to
the
                                // start of "mainMemory"
    bool valid;                 // If this bit is set, the translation is ignored.
                                // (In other words, the entry hasn't been initialized.)
    bool readOnly;              // If this bit is set, the user program is not allowed
                                // to modify the contents of the page.
    bool use;                   // This bit is set by the hardware every time the
                                // page is referenced or modified.
    bool dirty;                 // This bit is set by the hardware every time the
                                // page is modified.
    // add
    int LRU_counter;            // counter for LRU

    int ID; // page table ID
};
```

The translation by the LRU page replacement is implemented in code/machine/traslate.cc. First, we check the valid-invalid bit of the page table. If is not valid, it means the page is not in the main memory. Hence, we need to load from the secondary storage. There are two cases that may happen. First, if there are some empty frame in the main memory, then we can just load the page into it. The other case is that the main memory is full. In this cases, we create two buffers and runs least recently used (LRU) algorithm to find the victim. The ReadSector and WriteSector are used to read/write the temporarily saved pages found by the LRU algorithm. The victim is pull out from the main memory, and then swapped by our page into the corresponding frame. The page table will be updated correspondingly in both cases. In our implementation, LRU will perform linear search on LRU_counter to find the least recently used page.


```

} else if (!pageTable[vpn].valid) {
// not in main memory, demand paging
/* remove
    DEBUG(dbgAddr, "Invalid virtual page # " << virtAddr);
    return PageFaultException;
*/
//printf("Page fault\n");
kernel->stats->numPageFaults++; // page fault
j=0;
while(kernel->machine->UsedPhyPage[j]!=FALSE&&j<NumPhysPages){j++;}
// load the page into the main memory if the main memory is not full
if(j<NumPhysPages){
    char *buffer; //temporary save page
    buffer = new char[PageSize];
    kernel->machine->UsedPhyPage[j]=TRUE;
    kernel->machine->PhyPageInfo[j]=pageTable[vpn].ID;
    kernel->machine->main_tab[j]=&pageTable[vpn];
    pageTable[vpn].physicalPage = j;
    pageTable[vpn].valid = TRUE;
    pageTable[vpn].LRU_counter++; // counter for LRU

    kernel->SwapDisk->ReadSector(pageTable[vpn].virtualPage, buffer);
    bcopy(buffer,&mainMemory[j*PageSize],PageSize);
}
// main memory is full, page replacement
else{
    char *buffer1;
    char *buffer2;
    buffer1 = new char[PageSize];
    buffer2 = new char[PageSize];
    //Random
    //victim = (rand()%32);

    //LRU
    int min = pageTable[0].LRU_counter;
    victim=0;
    for(int index=0;index<32;index++){
        if(min > pageTable[index].LRU_counter){
            min = pageTable[index].LRU_counter;
            victim = index;
        }
    }
    pageTable[victim].LRU_counter++;

    //printf("Number %d page is swapped out\n",victim);

    // perform page replacement, write victim frame to disk, read
frame to memory
    bcopy(&mainMemory[victim*PageSize],buffer1,PageSize);
    kernel->SwapDisk->ReadSector(pageTable[vpn].virtualPage, buffer2);
    bcopy(buffer2,&mainMemory[victim*PageSize],PageSize);
    kernel->SwapDisk->WriteSector(pageTable[vpn].virtualPage,buffer1);

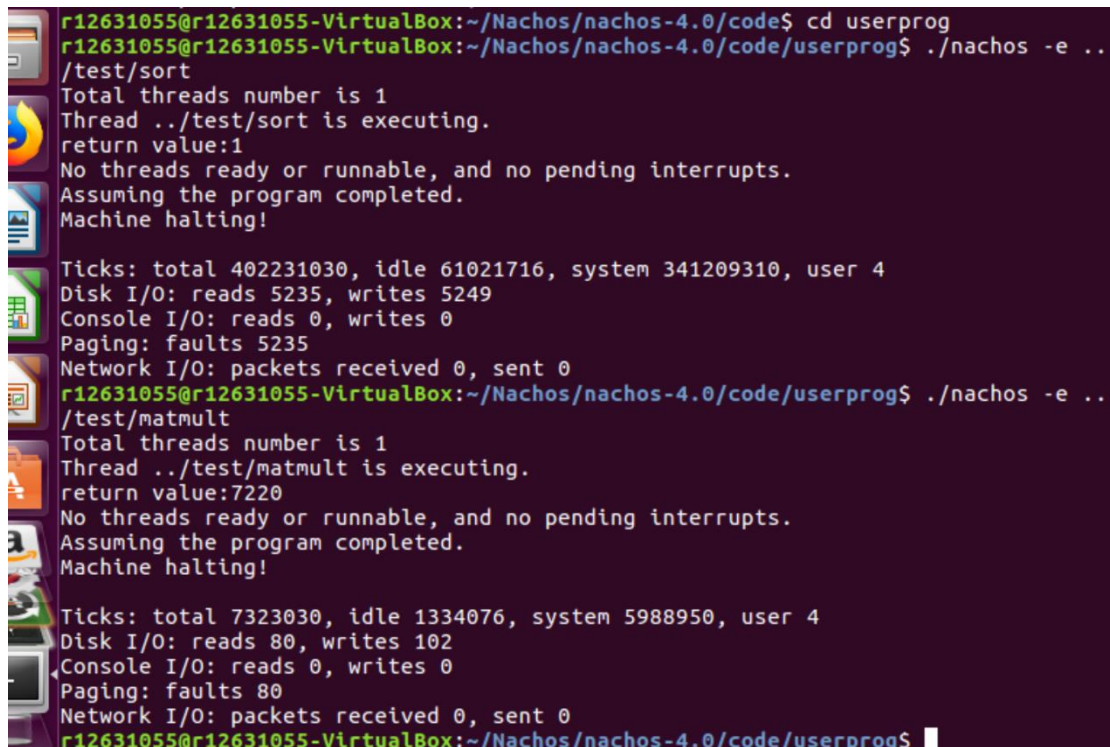
    main_tab[victim]->virtualPage=pageTable[vpn].virtualPage;

```

Results

1.

The result is shown below, and it can be reproduced by the following command `./nachos -e ../test/sort` and `./nachos -e ../test/matmult`



```
r12631055@r12631055-VirtualBox:~/Nachos/nachos-4.0/code$ cd userprog
r12631055@r12631055-VirtualBox:~/Nachos/nachos-4.0/code/userprog$ ./nachos -e ../test/sort
Total threads number is 1
Thread ../test/sort is executing.
return value:1
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

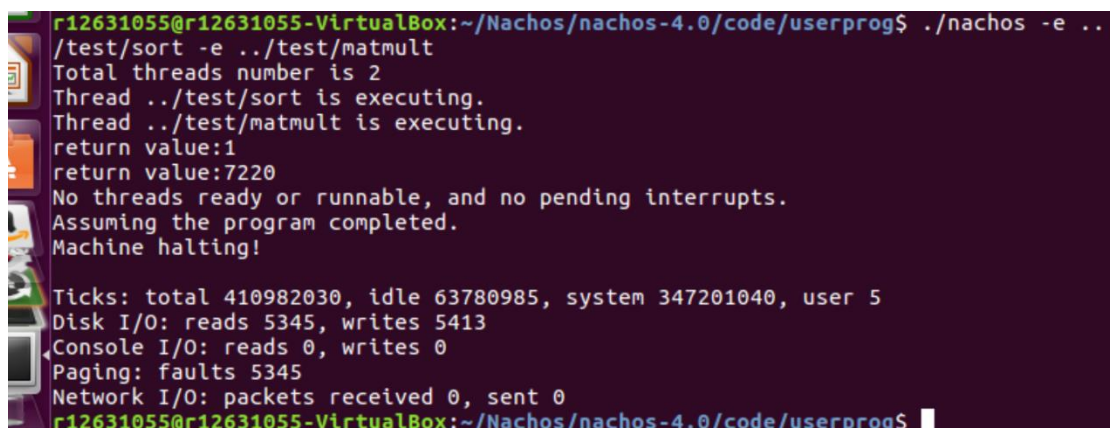
Ticks: total 402231030, idle 61021716, system 341209310, user 4
Disk I/O: reads 5235, writes 5249
Console I/O: reads 0, writes 0
Paging: faults 5235
Network I/O: packets received 0, sent 0
r12631055@r12631055-VirtualBox:~/Nachos/nachos-4.0/code/userprog$ ./nachos -e ../test/matmult
Total threads number is 1
Thread ../test/matmult is executing.
return value:7220
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 7323030, idle 1334076, system 5988950, user 4
Disk I/O: reads 80, writes 102
Console I/O: reads 0, writes 0
Paging: faults 80
Network I/O: packets received 0, sent 0
r12631055@r12631055-VirtualBox:~/Nachos/nachos-4.0/code/userprog$
```

The return value of sort is 1, which is the first element of sorted array `A[0]` as expected. The return value of matmult is 7220, which is the expected result.

2.

The result is shown below, and it can be reproduced by the following command `./nachos -e ../test/sort -e ../test/matmult`



```
r12631055@r12631055-VirtualBox:~/Nachos/nachos-4.0/code/userprog$ ./nachos -e ../test/sort -e ../test/matmult
Total threads number is 2
Thread ../test/sort is executing.
Thread ../test/matmult is executing.
return value:1
return value:7220
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 410982030, idle 63780985, system 347201040, user 5
Disk I/O: reads 5345, writes 5413
Console I/O: reads 0, writes 0
Paging: faults 5345
Network I/O: packets received 0, sent 0
r12631055@r12631055-VirtualBox:~/Nachos/nachos-4.0/code/userprog$
```

Reference:

<https://jasonblog.github.io/note/gunmake/shen ru xue xi makeming ling he makefile.html>

<https://morris821028.github.io/2014/05/30/lesson/hw-nachos4-2/>

<https://blog.terrynini.tw/tw/OS-NachOS-HW1/>

https://hackmd.io/@Z_yUjsyqRzaD5rSUQ6JOVw/S1hiIHr5D