Operating System Project 3-1

何孟遠
R11222007
December 28, 2023

1. What is your motivation, problem analysis, and plan?

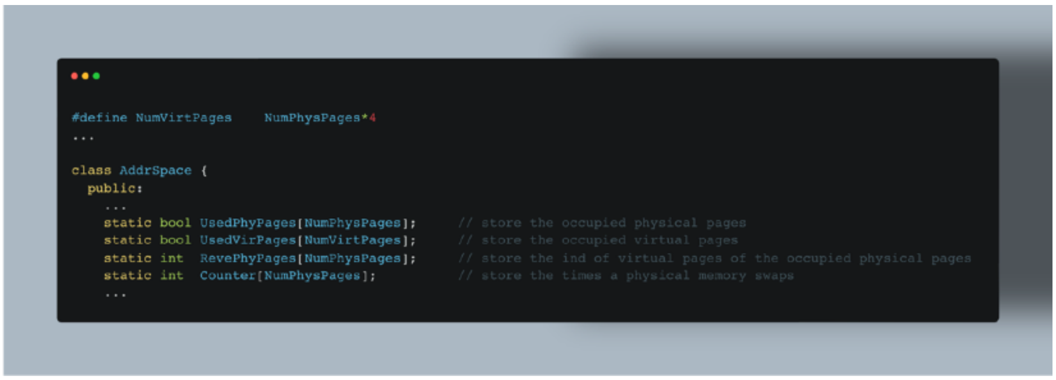
In this project, we want to use virtual memory to store the data larger than the physical memory size.

To do this, I do the following procedure:

- Design page tables, virtual tables, and physical tables. Every pointer in the page table either points to a physical table or a virtual table.
- Store the data one by one. If the memory still has a space, point the page table to a physical table; if don't, point to a virtual table.
- When executing, the CPU only gets the data inside a physical table. As a result, if the data is stored in a virtual table, the code needs to tell the program the exception, which is the page fault.
- As long as a page fault occurs, the exception handler needs to switch the data stored in the virtual memory to the physical memory. The code needs to decide which "victim" in the physical memory to switch out.
- Since I have to execute two programs concurrently, the code should clear the memory of the old thread.

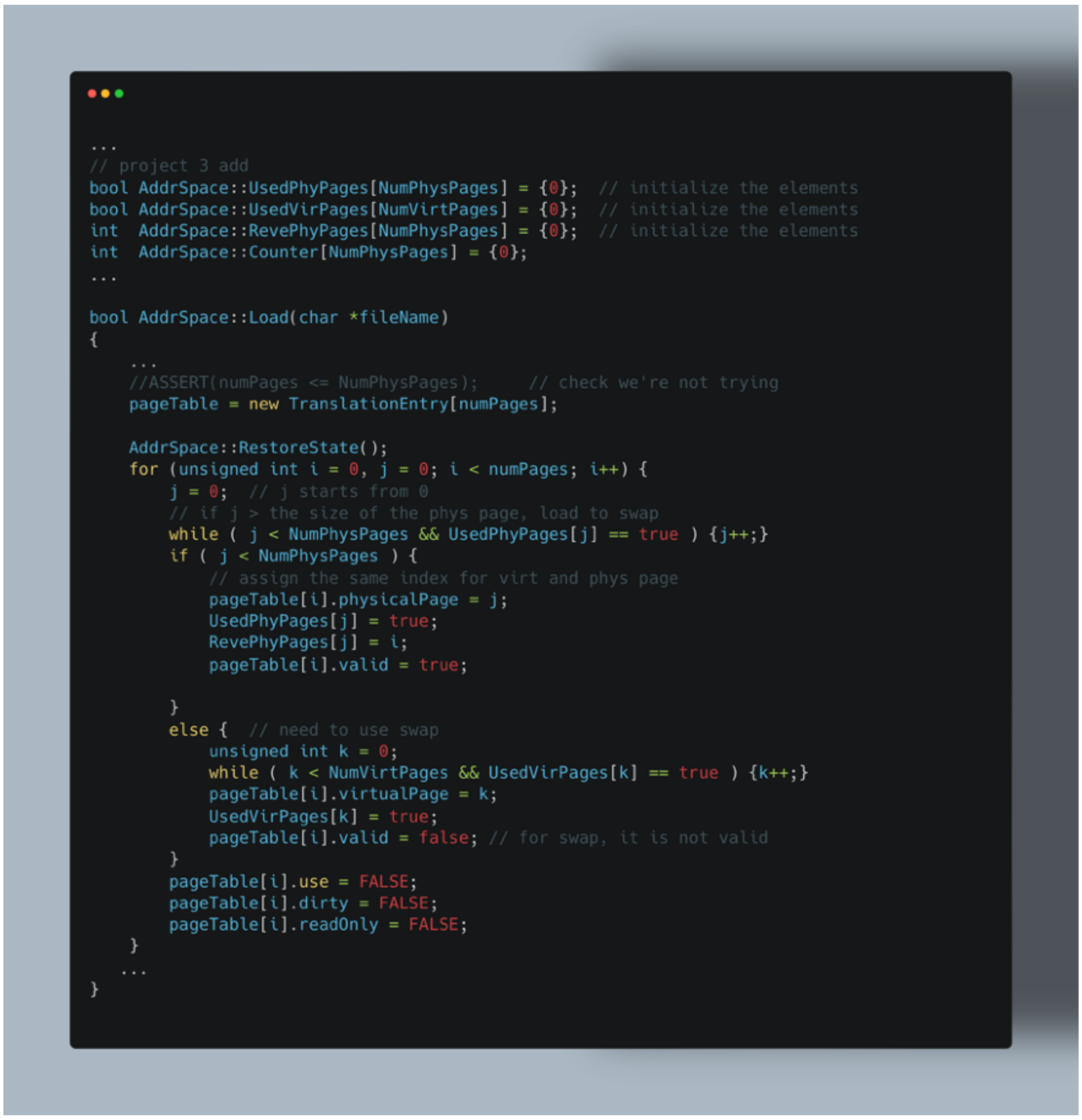
2. Explain the details of the code you added or modified.

- Design the page tables
    - addrspace.h



```
#define NumVirtPages    NumPhysPages*4
...

class AddrSpace {
  public:
    ...
    static bool UsedPhyPages[NumPhysPages];     // store the occupied physical pages
    static bool UsedVirPages[NumVirtPages];     // store the occupied virtual pages
    static int  RevePhyPages[NumPhysPages];     // store the ind of virtual pages of the occupied physical pages
    static int  Counter[NumPhysPages];          // store the times a physical memory swaps
    ...
```

- addrspace.cc



```
...
// project 3 add
bool AddrSpace::UsedPhyPages[NumPhysPages] = {0};  // initialize the elements
bool AddrSpace::UsedVirPages[NumVirtPages] = {0};  // initialize the elements
int  AddrSpace::RevePhyPages[NumPhysPages] = {0};  // initialize the elements
int  AddrSpace::Counter[NumPhysPages] = {0};
...

bool AddrSpace::Load(char *fileName)
{
    ...
    //ASSERT(numPages <= NumPhysPages);      // check we're not trying
    pageTable = new TranslationEntry[numPages];

    AddrSpace::RestoreState();
    for (unsigned int i = 0, j = 0; i < numPages; i++) {
        j = 0;  // j starts from 0
        // if j > the size of the phys page, load to swap
        while ( j < NumPhysPages && UsedPhyPages[j] == true ) {j++;}
        if ( j < NumPhysPages ) {
            // assign the same index for virt and phys page
            pageTable[i].physicalPage = j;
            UsedPhyPages[j] = true;
            RevePhyPages[j] = i;
            pageTable[i].valid = true;

        }
        else {  // need to use swap
            unsigned int k = 0;
            while ( k < NumVirtPages && UsedVirPages[k] == true ) {k++;}
            pageTable[i].virtualPage = k;
            UsedVirPages[k] = true;
            pageTable[i].valid = false; // for swap, it is not valid
        }
        pageTable[i].use = FALSE;
        pageTable[i].dirty = FALSE;
        pageTable[i].readOnly = FALSE;
    }
    ...
}
```

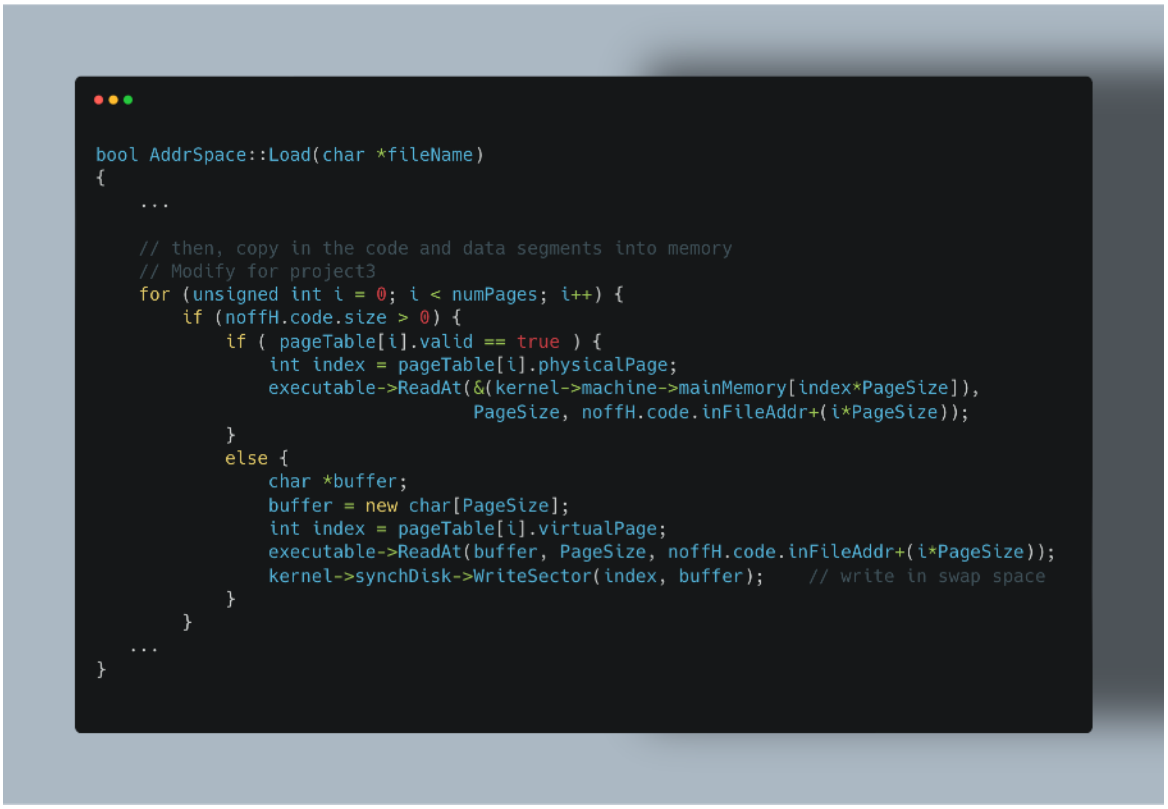
In addrspace.cc, since the kernel initially has no page table, I add AddrSpace::RestoreState() immediately after the pages are created.

The pagetable[i].valid flag represents whether it is stored in the physical memory. If true, the kernel will directly read/write memory to that page; if false, the translate.cc will raise an exception.

When deciding on the physical page for page tables, the code needs to pass the physical page that has been occupied, the same for the virtual table. So the code needs **UsedPhyPages** and **UsedVirPages** to record the status of the pages.

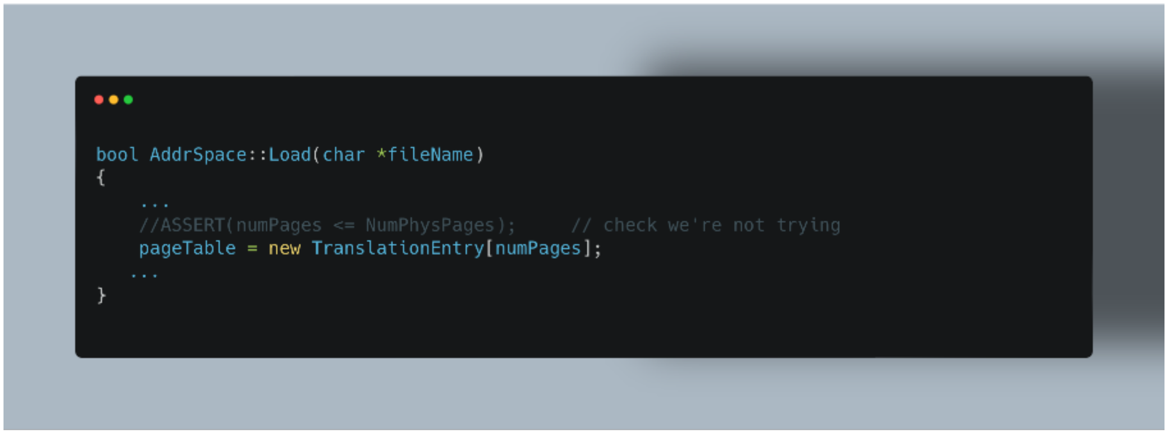
RevePhyPages and **Counter** are created for the swap, I will talk about it later.

- Store the data one by one.
    - addrspace.cc



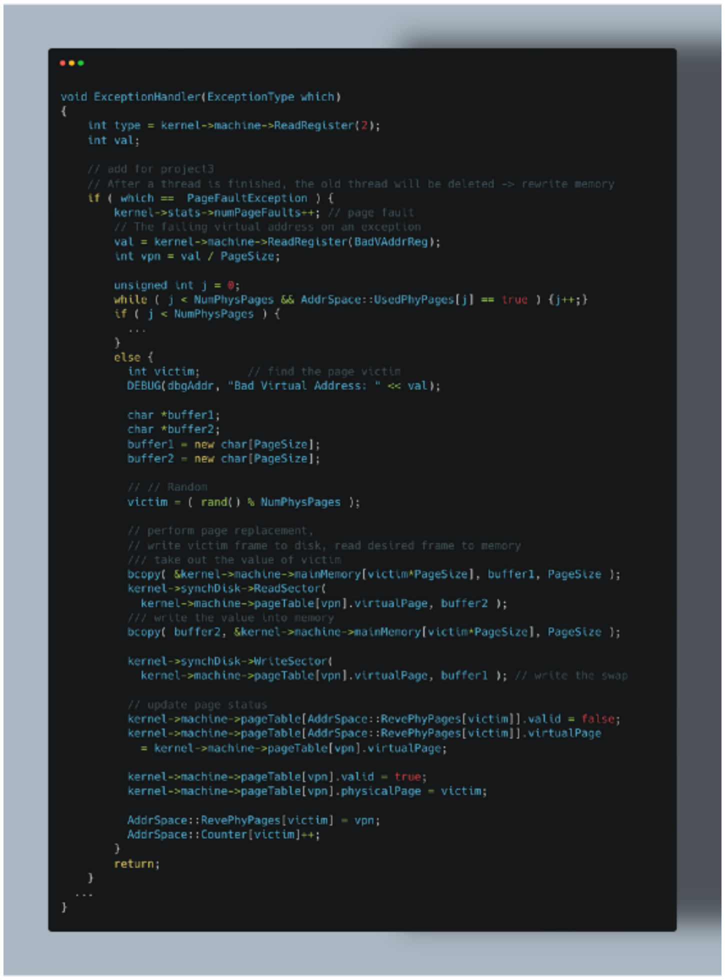
```
bool AddrSpace::Load(char *fileName)
{
    ...

    // then, copy in the code and data segments into memory
    // Modify for project3
    for (unsigned int i = 0; i < numPages; i++) {
        if (noffH.code.size > 0) {
            if ( pageTable[i].valid == true ) {
                int index = pageTable[i].physicalPage;
                executable->ReadAt(&(kernel->machine->mainMemory[index*PageSize]),
                                   PageSize, noffH.code.inFileAddr+(i*PageSize));
            }
            else {
                char *buffer;
                buffer = new char[PageSize];
                int index = pageTable[i].virtualPage;
                executable->ReadAt(buffer, PageSize, noffH.code.inFileAddr+(i*PageSize));
                kernel->synchDisk->WriteSector(index, buffer);    // write in swap space
            }
        }
    ...
}
```

- For the page fault to occur.



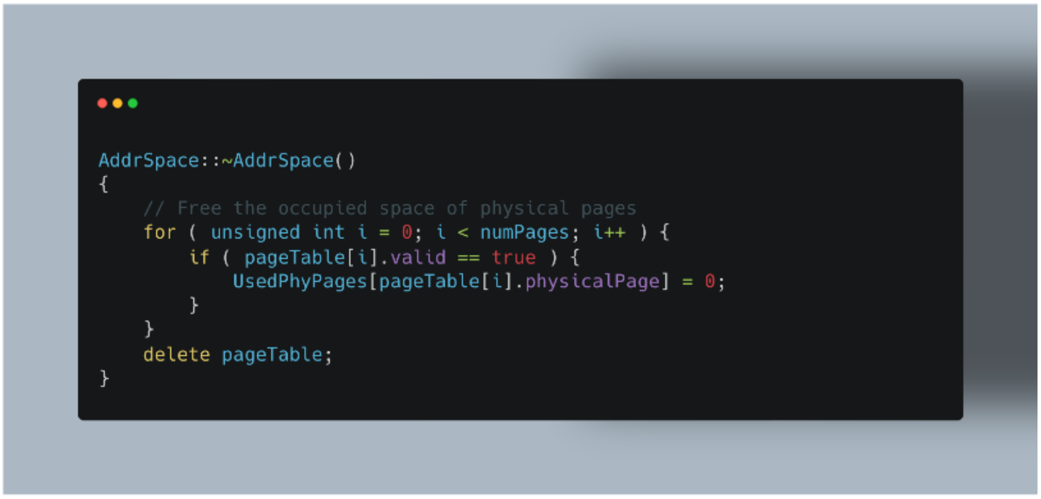
```
bool AddrSpace::Load(char *fileName)
{
    ...
    //ASSERT(numPages <= NumPhysPages);     // check we're not trying
    pageTable = new TranslationEntry[numPages];
   ...
}
```

  - The page fault will occur when running Machine::Translate, then the error will go to the exception handler through Machine::RaiseExcpetion.
- The exception handler.
  - In exception.cc



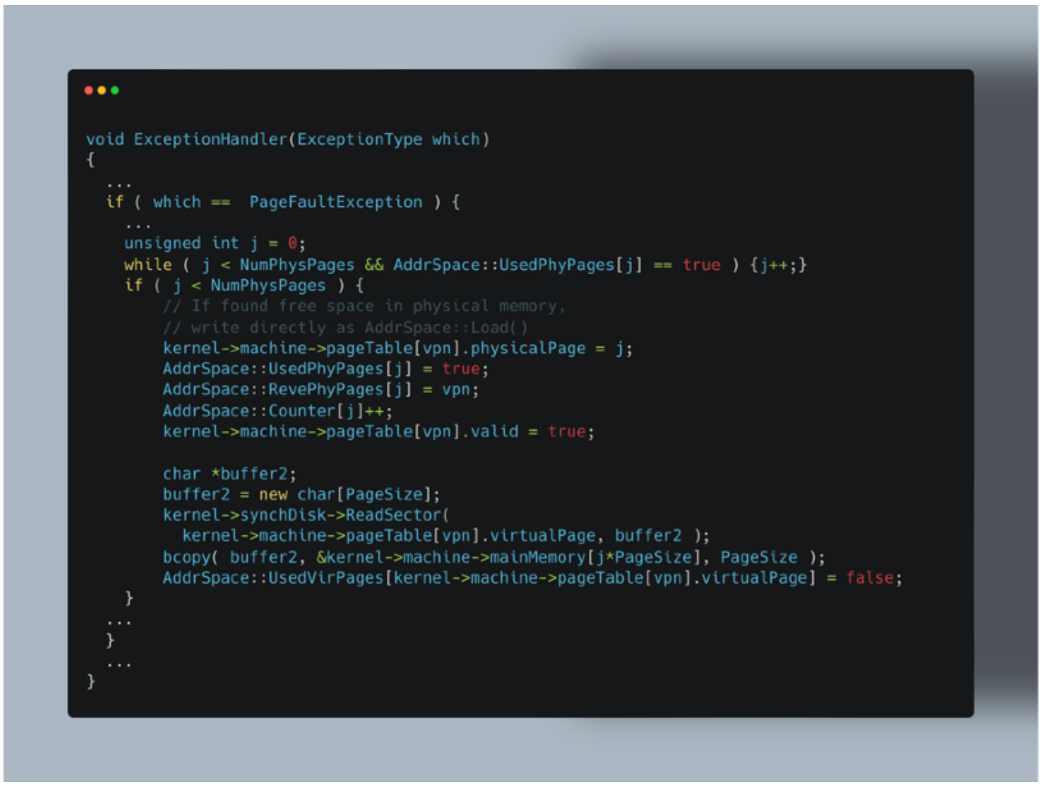
```
void ExceptionHandler(ExceptionType which)
{
    int type = kernel->machine->ReadRegister(2);
    int val;

    // add for project3
    // After a thread is finished, the old thread will be deleted -> rewrite memory
    if ( which ==  PageFaultException ) {
        kernel->stats->numPageFaults++; // page fault
        // The failing virtual address on an exception
        val = kernel->machine->ReadRegister(BadVAddrReg);
        int vpn = val / PageSize;

        unsigned int j = 0;
        while ( j < NumPhysPages && AddrSpace::UsedPhyPages[j] == true ) {j++;}
        if ( j < NumPhysPages ) {
          ...
        }
        else {
          int victim;       // find the page victim
          DEBUG(dbgAddr, "Bad Virtual Address: " << val);

          char *buffer1;
          char *buffer2;
          buffer1 = new char[PageSize];
          buffer2 = new char[PageSize];

          // // Random
          victim = ( rand() % NumPhysPages );

          // perform page replacement,
          // write victim frame to disk, read desired frame to memory
          /// take out the value of victim
          bcopy( &kernel->machine->mainMemory[victim*PageSize], buffer1, PageSize );
          kernel->synchDisk->ReadSector(
            kernel->machine->pageTable[vpn].virtualPage, buffer2 );
          /// write the value into memory
          bcopy( buffer2, &kernel->machine->mainMemory[victim*PageSize], PageSize );

          kernel->synchDisk->WriteSector(
            kernel->machine->pageTable[vpn].virtualPage, buffer1 ); // write the swap

          // update page status
          kernel->machine->pageTable[AddrSpace::RevePhyPages[victim]].valid = false;
          kernel->machine->pageTable[AddrSpace::RevePhyPages[victim]].virtualPage
            = kernel->machine->pageTable[vpn].virtualPage;

          kernel->machine->pageTable[vpn].valid = true;
          kernel->machine->pageTable[vpn].physicalPage = victim;

          AddrSpace::RevePhyPages[victim] = vpn;
          AddrSpace::Counter[victim]++;
        }
        return;
    }
  ...
}
```

    - Here, the code first uses a random algorithm to choose a victim. After the victim is chosen, the code needs to switch the data between the physical page and the swap.
    - **RevePhyPages** is useful for knowing which data page uses that victim's physical memory.
- Clear the memory of the old thread when finishing.
    - In addrspace.cc

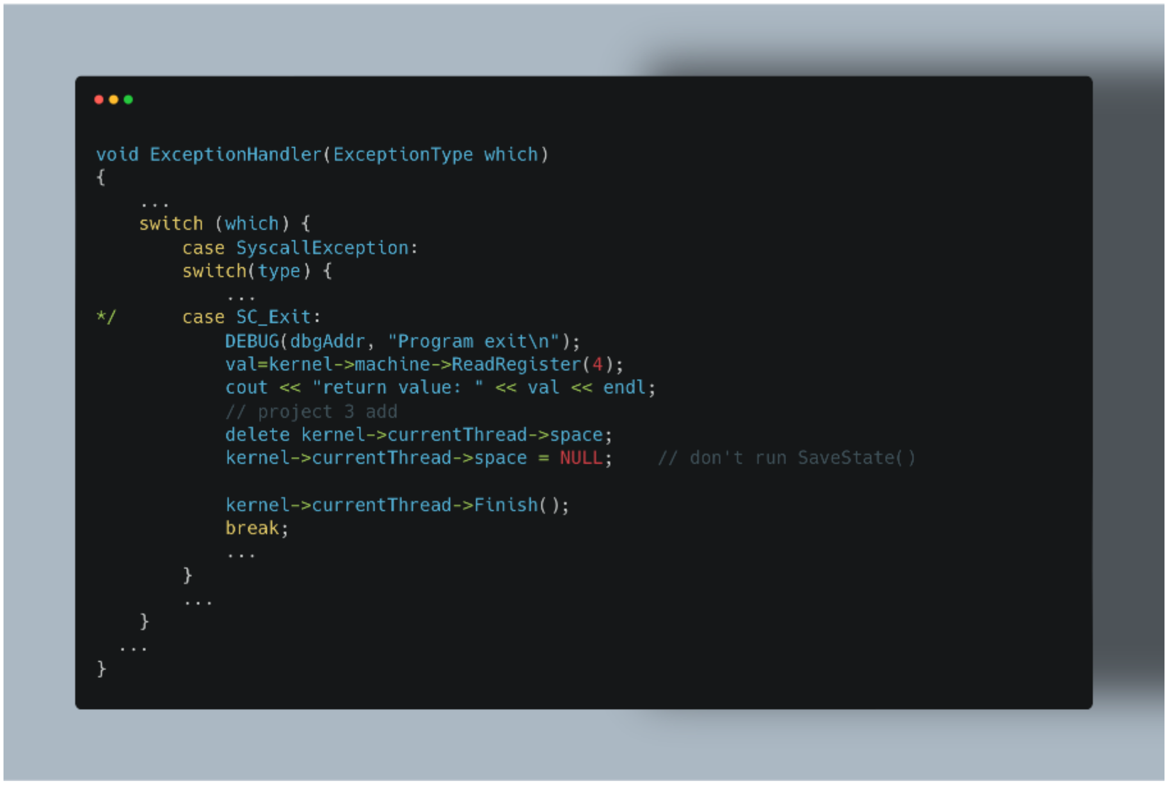


```
AddrSpace::~AddrSpace()
{
    // Free the occupied space of physical pages
    for ( unsigned int i = 0; i < numPages; i++ ) {
        if ( pageTable[i].valid == true ) {
            UsedPhyPages[pageTable[i].physicalPage] = 0;
        }
    }
    delete pageTable;
}
```

    - In exception.cc.



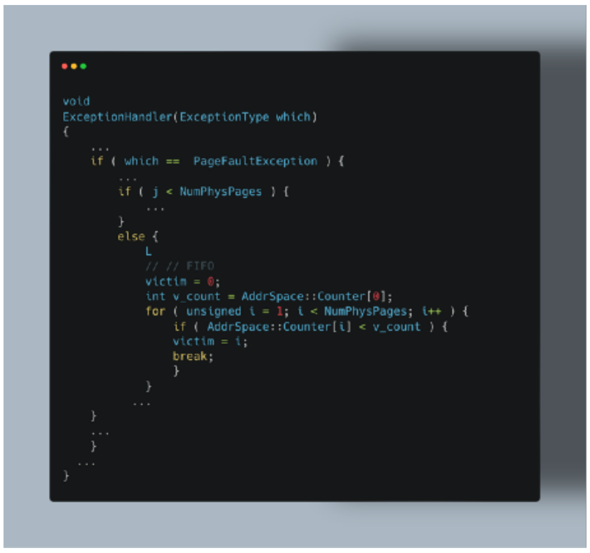
```
void ExceptionHandler(ExceptionType which)
{
  ...
  if ( which ==  PageFaultException ) {
    ...
    unsigned int j = 0;
    while ( j < NumPhysPages && AddrSpace::UsedPhyPages[j] == true ) {j++;}
    if ( j < NumPhysPages ) {
        // If found free space in physical memory,
        // write directly as AddrSpace::Load()
        kernel->machine->pageTable[vpn].physicalPage = j;
        AddrSpace::UsedPhyPages[j] = true;
        AddrSpace::RevePhyPages[j] = vpn;
        AddrSpace::Counter[j]++;
        kernel->machine->pageTable[vpn].valid = true;

        char *buffer2;
        buffer2 = new char[PageSize];
        kernel->synchDisk->ReadSector(
          kernel->machine->pageTable[vpn].virtualPage, buffer2 );
        bcopy( buffer2, &kernel->machine->mainMemory[j*PageSize], PageSize );
        AddrSpace::UsedVirPages[kernel->machine->pageTable[vpn].virtualPage] = false;
    }
  ...
  }
  ...
}
```



```
void ExceptionHandler(ExceptionType which)
{
    ...
    switch (which) {
        case SyscallException:
        switch(type) {
            ...
*/      case SC_Exit:
            DEBUG(dbgAddr, "Program exit\n");
            val=kernel->machine->ReadRegister(4);
            cout << "return value: " << val << endl;
            // project 3 add
            delete kernel->currentThread->space;
            kernel->currentThread->space = NULL;    // don't run SaveState()

            kernel->currentThread->Finish();
            break;
            ...
        }
        ...
    }
  ...
}
```

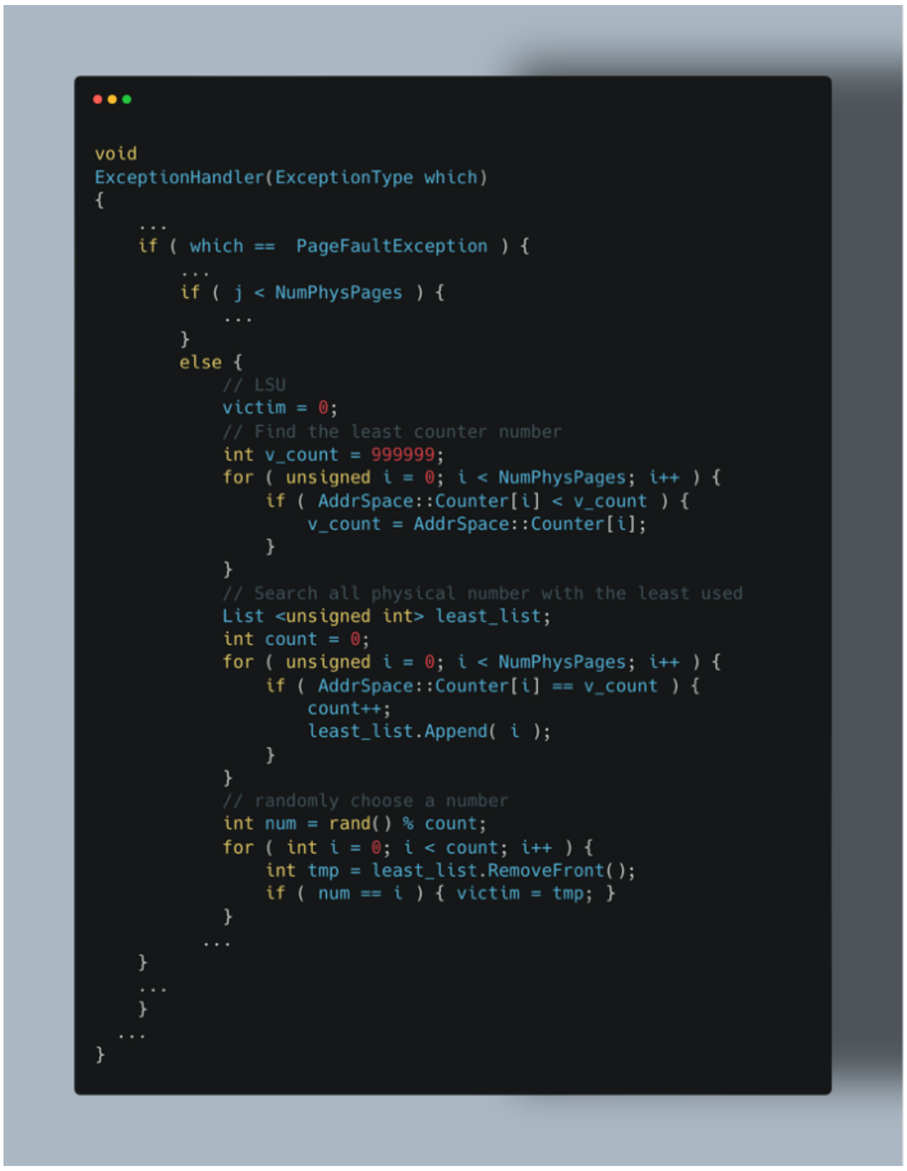
By setting kernel->currentThread->space = NULL, the SaveState() in Scheduler::Run() will not execute, hence kernel->machine->pageTable will use the pageTable in the second execution instead of the old one. If this hasn't been done, the output of the program will not correct even if we execute the same program twice.

- Design the different algorithms for deciding victim.
    - In exception.cc. Since the Random algorithm has been used for the figure above, I will not list it again here.
        - FIFO



```
void
ExceptionHandler(ExceptionType which)
{
    ...
    if ( which ==  PageFaultException ) {
        ...
        if ( j < NumPhysPages ) {
            ...
        }
        else {
            L
            // // FIFO
            victim = 0;
            int v_count = AddrSpace::Counter[0];
            for ( unsigned i = 1; i < NumPhysPages; i++ ) {
                if ( AddrSpace::Counter[i] < v_count ) {
                victim = i;
                break;
                }
            }
          ...
    }
    ...
    }
  ...
}
```

- LSU



```
void
ExceptionHandler(ExceptionType which)
{
    ...
    if ( which ==  PageFaultException ) {
        ...
        if ( j < NumPhysPages ) {
            ...
        }
        else {
            // LSU
            victim = 0;
            // Find the least counter number
            int v_count = 999999;
            for ( unsigned i = 0; i < NumPhysPages; i++ ) {
                if ( AddrSpace::Counter[i] < v_count ) {
                    v_count = AddrSpace::Counter[i];
                }
            }
            // Search all physical number with the least used
            List <unsigned int> least_list;
            int count = 0;
            for ( unsigned i = 0; i < NumPhysPages; i++ ) {
                if ( AddrSpace::Counter[i] == v_count ) {
                    count++;
                    least_list.Append( i );
                }
            }
            // randomly choose a number
            int num = rand() % count;
            for ( int i = 0; i < count; i++ ) {
                int tmp = least_list.RemoveFront();
                if ( num == i ) { victim = tmp; }
            }
          ...
    }
    ...
    }
  ...
}
```

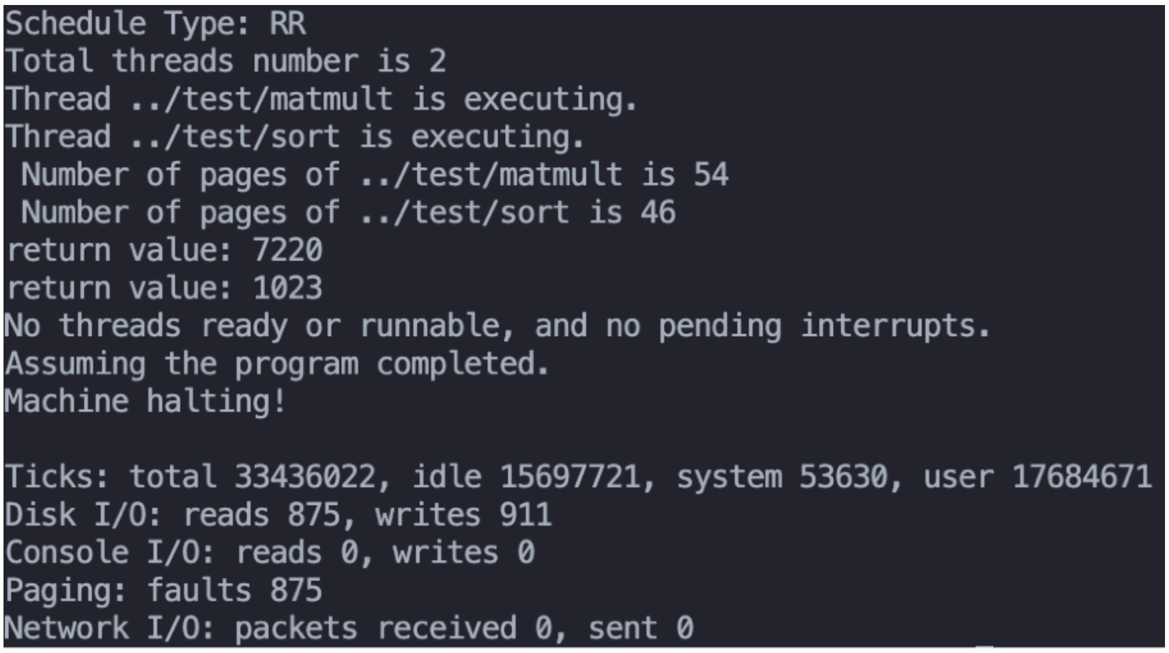
- Both algorithms need to use AddrSpace::Counter to find the victim. The difference between them is for FIFO, the code only needs to find the first victim that fulfills the condition (the number of counts is smaller than the previous one). For LSU, the code needs to find the victim set of the lowest count, and then randomly choose one to be the victim.
- For the correct LSU algorithm, the count needs to increase once the memory is used. However, such an object can only be

reached by modifying the files in the machine directory, which is forbidden in this project.

- Hence, I can only record the times of becoming the victim as the reference for implementing the LSU algorithm.

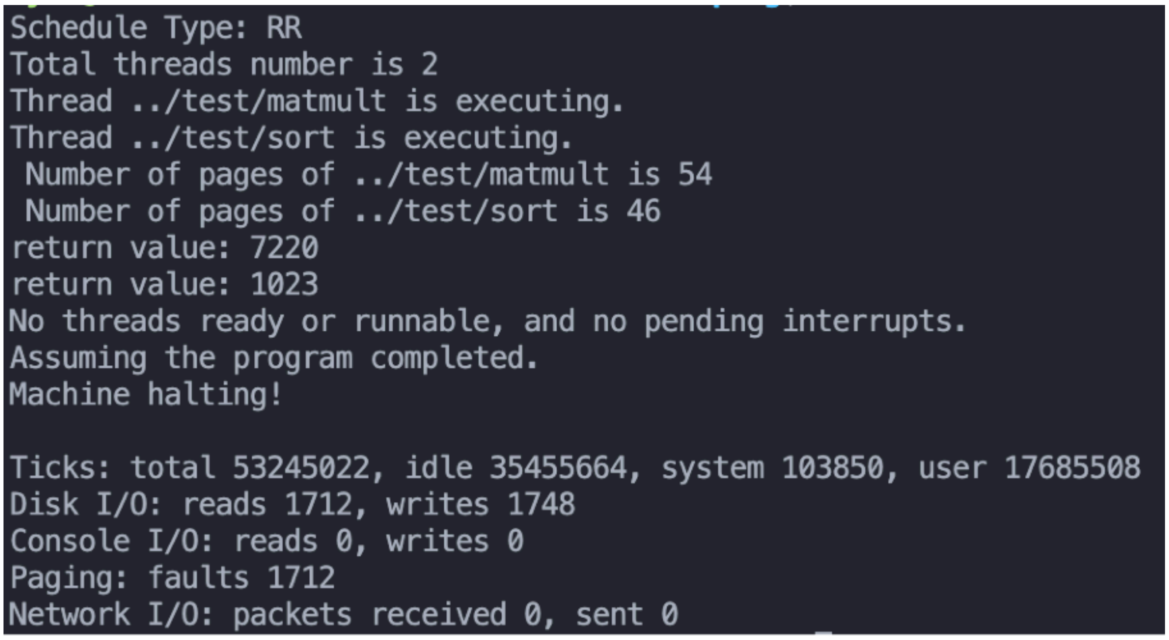
3. Experiment results with some discussion and observation.

- Random: 875 faults.



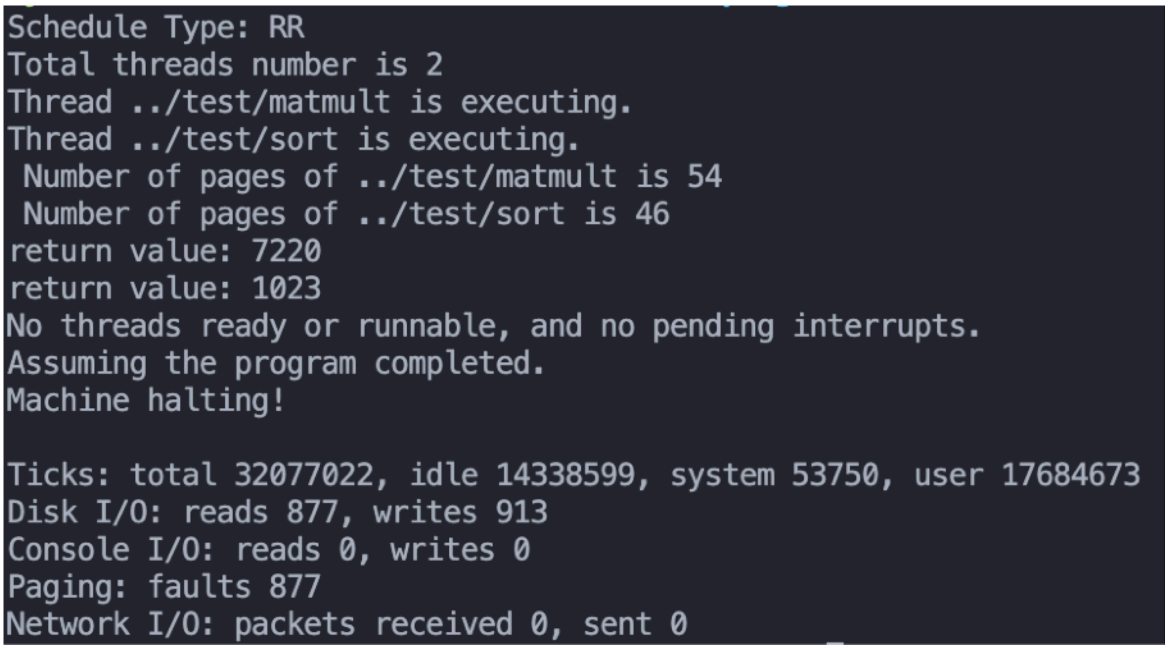
```
Schedule Type: RR
Total threads number is 2
Thread ../test/matmult is executing.
Thread ../test/sort is executing.
 Number of pages of ../test/matmult is 54
 Number of pages of ../test/sort is 46
return value: 7220
return value: 1023
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 33436022, idle 15697721, system 53630, user 17684671
Disk I/O: reads 875, writes 911
Console I/O: reads 0, writes 0
Paging: faults 875
Network I/O: packets received 0, sent 0
```

- FIFO: 1712 faults.



```
Schedule Type: RR
Total threads number is 2
Thread ../test/matmult is executing.
Thread ../test/sort is executing.
 Number of pages of ../test/matmult is 54
 Number of pages of ../test/sort is 46
return value: 7220
return value: 1023
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 53245022, idle 35455664, system 103850, user 17685508
Disk I/O: reads 1712, writes 1748
Console I/O: reads 0, writes 0
Paging: faults 1712
Network I/O: packets received 0, sent 0
```

- LSU: 877 faults.



```
Schedule Type: RR
Total threads number is 2
Thread ../test/matmult is executing.
Thread ../test/sort is executing.
 Number of pages of ../test/matmult is 54
 Number of pages of ../test/sort is 46
return value: 7220
return value: 1023
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 32077022, idle 14338599, system 53750, user 17684673
Disk I/O: reads 877, writes 913
Console I/O: reads 0, writes 0
Paging: faults 877
Network I/O: packets received 0, sent 0
```

From the above results, I noticed that the random algorithm obtains the least faults and the faster speed of execution.

However, if the number of physical pages is decreased, the number of faults under the random algorithm may increase a lot. Furthermore, the LSU algorithm in my code is not perfect. If I can modify the file in the machine directory, the number of faults may be further reduced.

4. What problem do you face and how do you tackle it?

The main problem when testing the code is the error when reading memory. If I don't modify the code carefully, the code will just pop up an unexpected exception.

If I print out the process of read/write memory, my console will be filled out with lots of messages, which is useless for debugging. This problem costs me hours to solve, and the only solution is to review the whole code.

However, the debugger flag still helps me a lot. The best way to debug is to know exactly how the code is running. If the problem is something not related to the memory/swap address, the printout message can give a hint.

For example, after I successfully executed a single program, the code always failed to move to the second execution with the memory allocation error. Track down the memory process, I finally found out that even deleting the pageTable of the kernel is not enough, I need to set the pointer of the kernel to NULL then AddrSpace::SaveState() will not be executed, hence getting the correct results. This error can only be noticed if I dig into the whole process of threadkernel.