MEMORY MANAGEMENT

Part 1 of 3rd Project of Operating System

Author

Name: Man-Chen Hsu

Student ID: R11942135

Due: December 28th, 2023

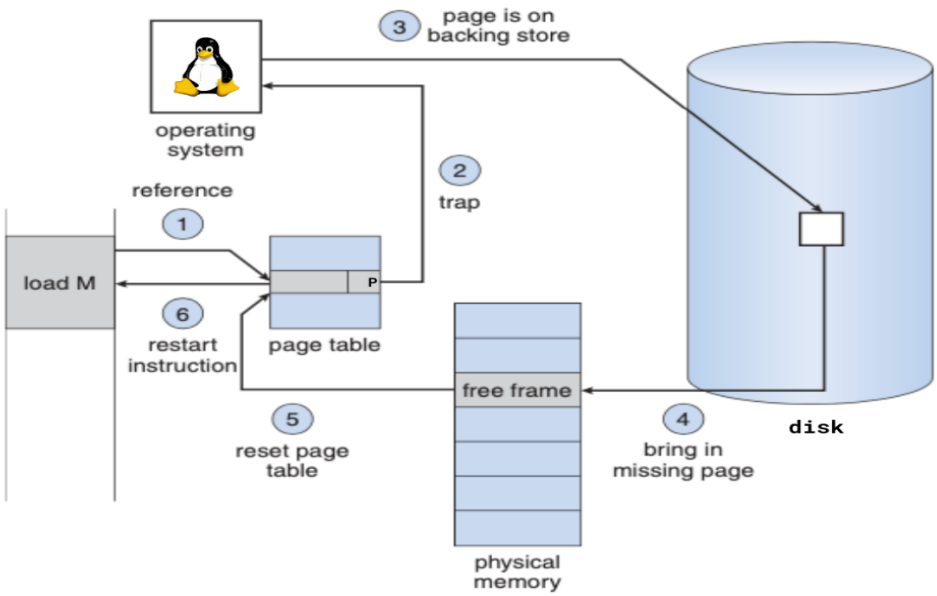
Contents

1 Plan

```
gina@gina-VirtualBox:~/repo/Nachos/nachos-4.0/code/userprog$ ./nachos -e ../test
/matmult
Total threads number is 1
Thread ../test/matmult is executing.
Assertion failed: line 138 file ../userprog/addrspace.cc
Aborted (core dumped)
```

```
gina@gina-VirtualBox:~/repo/Nachos/nachos-4.0/code/userprog$ ./nachos -e ../test
/sort
Total threads number is 1
Thread ../test/sort is executing.
Assertion failed: line 138 file ../userprog/addrspace.cc
Aborted (core dumped)
```

First, execute `../test/sort` and `../test/matmult` respectively, and then abort will happen. Since the main memory is not large enough, the two files can't be executed successfully. Therefore, we need to swap out some pages to disk and load the necessary pages. That is the technology of demand paging.



From the block diagram, `../machine/translate.cc` will first check whether the frame is valid. If it is invalid, page fault happens. From now on, our project is going to deal with the problem. When page fault happens, `../machine/translate.cc` will throw `PageFaultException`. Then we have to add a case `PageFaultException` in `../userprog/exception.cc`. This corresponds to the step 2 in the above picture. Then, we will add some

classes in `../addrspace.cc` and they includes many functions. We can implement page replacement algorithms and release the page in the file and then swap in the necessary frame, corresponding to the step 3 to step 5.

Additionally, we will create two tables: the frame table and the swap table. And the page table has existed. In all these tables, `valid` will be used to check whether the entry is in use. In the page table, we also store the virtual page and physical page. The virtual page stands for the page number in disk. And the physical page stands for the page number in memory. The total page table entry will be different between processes. In the frame table and the swap table, we will store the address space of that entry. And `vpn` stands for the entry in the page table. There are in total 32 frames in the frame table. For the swap table, there are 1024 entries.

2 Code

When a valid bit is set to be invalid, file `../machine/translate.cc` will throw a `PageFaultException`. Therefore, we have to add a case `PageFaultException` in `../userprog/exception.cc` to trap to kernel. In this file, define the parameter before switch. Then we have to find where the invalid virtual address is. When exception happens, the virtual address will be put into register 39. Also, we can calculate the page, where the virtual address is. Then we are going to deal with the page fault through the function PageFaultHandler, which will be explained in detail later.

```
int badVAddr;
int vpn;
```

```
97          // HW 3 added (a case for page fault to trap to OS)
98          case PageFaultException:
99                  kernel->stats->numPageFaults++; // add the page fault number
100                 badVAddr = kernel->machine->ReadRegister(39); // get the pag
    e virtual address which is not in memory
101                 vpn = badVAddr / PageSize; // calculate the page
102                 kernel->memoryManager->PageFaultHandler(vpn);
103                 return;
```

In `../userprog/addrspace.h`, we first define the types of page replacement with enum. Notice that the type must not be the same as the scheduler in Homework 2, or compile error would happen. And set the `*pageTable` to be public so that we can use it directly later. Add noffH and executable

to copy data into main memory. We add a function TranslateVir2Phys to help us translate the virtual address into physical address. Finally, create two class, `FrameInfoEntry` for creating a table and including all the parameters within a table, and MemoryManager for handling the page fault and page replacement.

```
// Page Replacement Deifinition (HW 3 added)
enum ReplacementType{
        Random,
        FIFO
        //LRU,
        //LFU
};
```

```
class AddrSpace {
  public:
    AddrSpace();                        // Create an address space.
    ~AddrSpace();                       // De-allocate an address space

    void Execute(char *fileName);       // Run the the program
                                        // stored in the file "executable"

    void SaveState();                   // Save/restore address space-specific
    void RestoreState();                // info on a context switch

    TranslationEntry *pageTable;                // Assume linear page table translation
(change to public in HW3)

    // noffH and executable will be used later to copy data into main memory (added in HW3)
    NoffHeader noffH;
    OpenFile *executable;

    static bool UsedPhysicalPageState[NumPhysPages];

  private:
    //TranslationEntry *pageTable;      // Assume linear page table translation
                                        // for now!
    unsigned int numPages;              // Number of pages in the virtual
                                        // address space

    bool Load(char *fileName);          // Load the program into memory
                                        // return false if not found

    void InitRegisters();               // Initialize user-level CPU registers,
                                        // before jumping to user code

    int TranslateVir2Phys(int virAddr);
};
```

```
// Define the parameter related to a page (added in HW3)
class FrameInfoEntry{
public:
        bool valid; // check whether a page could be used;
        bool lock;
        AddrSpace* addrspace; // the address of the process in use
        unsigned int vpn; // virtual page number

        unsigned int latestTick; // for FCFS
        unsigned int usageCount; // for LFU
};

// Define these functions to handle the page fault (later implemented) (added in HW3)
class MemoryManager{
public:
        ReplacementType VicType;
        MemoryManager(ReplacementType v);
        int TransAddr(AddrSpace* space, int virAddr);
        bool AcquirePage(AddrSpace* space, int vpn);
        bool ReleasePage(AddrSpace* space, int vpn);
        void PageFaultHandler(int faultPageNum);

        int ChooseVictim();
};
```

In Homework 1, we have created an array `UsedPhysicalPageState` to record which page has been used. If all the process has been done, we will set the entry of the array to be false and delete the page table.

```
bool AddrSpace::UsedPhysicalPageState[NumPhysPages]={0};
```

```
//----------------------------------------------------------------------
// AddrSpace::~AddrSpace
//      Dealloate an address space.
//----------------------------------------------------------------------

AddrSpace::~AddrSpace()
{

   for (unsigned int i=0;i<numPages;i++){
        AddrSpace::UsedPhysicalPageState[pageTable[i].physicalPage]=false;
}
   delete pageTable;
}
```

In `../userprog/addrspace.cc`, we implement the function `TranslateVir2Phys` to calculate where the physical address is. First, we have to obtain the page by dividing virtual address by page size. And the offset is the remainder. The physical address equals to the page number multiplying page size and adding the offset.

```
// HW3 added
int AddrSpace::TranslateVir2Phys(int virAddr){
    int ret_physAddr = 0;
    int vpn = (unsigned) virAddr / PageSize;
    int offset = (unsigned) virAddr % PageSize;
    ret_physAddr = pageTable[vpn].physicalPage * PageSize + offset;
    return ret_physAddr;
}
```

In `AddrSpace::Load`, we don't need to declare noffH and executable, since we have define them in the header file. Also, we are going to implement virtual memory, so we don't need the assertion in the following picture.

```
bool
AddrSpace::Load(char *fileName)
{
   // OpenFile *executable = kernel->fileSystem->Open(fileName);
   executable = kernel->fileSystem->Open(fileName);
   // NoffHeader noffH;
    unsigned int size;
```

```
//    ASSERT(numPages <= NumPhysPages);         // check we're not trying
                                                // to run anything too big --
                                                // at least until we have
                                                // virtual memory
```

For initializing, we randomly choose 32 pages from arbitrary processes and put them into the frame table. The rest will be put into the swap table, that is, disk. After the page is put into the frame table, the virtual page in the page table will be set to be 1024. Since the size of the page table is 1024, the virtual page should be 1023. Setting virtual page to be 1024 means that the page doesn't need to exist in the page table. Then set the entry of the page table to be true, so that other process could use it. If an entry of a frame is in use, we first set the `UsedPhysicalPageState` to be true. And the frame will be not valid, since it is in use. We also update the number of using the frame for LFU algorithm and the time of a frame that has recently been used for FCFS algorithm. The usage count should be added by 1 and we use `kernel->stats->totalTicks` to get the time when a frame being put into the table.

The rest will be put into the swap table (disk). The parameter in the swap table should be set in the same way as that of the frame table being used. For the page table, valid should be set to be false, since the entry is in use. And the virtual page will equal to the sector on disk. Since it is not in the frame table, the physical page should be set as the total number of the frame table, meaning that it is not in the frame table.

```
// HW3 added
int i,j,k;
// set physical frame
for (i=0, j=0; i<numPages && j<NumPhysPages; i++, j++){
    while(kernel->frameTable[j].valid==false)j++;
    pageTable[i].virtualPage=1024; // no need for VM
    pageTable[i].physicalPage=j;
    pageTable[i].valid=true; // can be used
    pageTable[i].use=false;
    pageTable[i].dirty=false;
    pageTable[i].readOnly=false;
    // update frameTable
    AddrSpace::UsedPhysicalPageState[j]=true;
    kernel->frameTable[j].valid=false; // in use
    kernel->frameTable[j].addrspace = this;
    kernel->frameTable[j].vpn=i;
    kernel->frameTable[j].usageCount++; // for LFU
    kernel->frameTable[j].latestTick = kernel->stats->totalTicks; // for FCFS
 }
 for(k=0; i<numPages && k<1024; i++, k++){
    // find an available disk segment
    while(kernel->swapTable[k].valid==false)k++;

    // update swapTable
    kernel->swapTable[k].valid=false;
    kernel->swapTable[k].addrspace = this;
    kernel->swapTable[k].vpn=i;

    // update pageTable
    pageTable[i].valid=false; // can't be used
    pageTable[i].virtualPage=k; // in disk
    pageTable[i].physicalPage=NumPhysPages; // not in physical memory
    pageTable[i].use=false;
    pageTable[i].dirty=false;
    pageTable[i].readOnly=false;
  }
```

Then we are going to load the code segment and data segment. If code is now valid in frame, we could directly load it into main memory. If not, create a buffer and put code into it. Then write into the sector on the disk. And do the same thing for data.

```
    // then, copy in the code and data segments into memory
    i=0;
    if(noffH.code.size > 0){
        DEBUG(dbgAddr, "Initializing code segment.");
        DEBUG(dbgAddr, noffH.code.virtualAddr << "," << noffH.code.size);

        // load it one by one
        // since the code segment always start with viraddr=0;
        int codePages = divRoundUp(noffH.code.size, PageSize);
        for(i=0; i<codePages; i++){
              // For each page i, find frame or VM
              if (pageTable[i].valid){
                      // in physical frame
                      int physAddr_code = kernel->memoryManager->TransAddr(this, i*PageSize);
                      executable->ReadAt(&(kernel->machine->mainMemory[physAddr_code]), PageSize, noffH.code.inFileAddr+i*PageSi
ze);
              }
              else{
                      // in VM
                      int k = pageTable[i].virtualPage;
                      char* outBuffer = new char[PageSize];
                      executable->ReadAt(outBuffer, PageSize, noffH.code.inFileAddr+i*PageSize);
                      kernel->swap->WriteSector(k, outBuffer);
              }
        }
     }

     if(noffH.initData.size > 0){
      DEBUG(dbgAddr, "Initializing data segment.");
      DEBUG(dbgAddr, noffH.initData.virtualAddr << "," << noffH.initData.size
);
      for(; i<numPages;i++){
              if(pageTable[i].valid){
              // in physical frame
              int physAddr_data = kernel->memoryManager->TransAddr(this, i*PageSize);
              executable->ReadAt(&(kernel->machine->mainMemory[physAddr_data]), PageSize, noffH.initData.inFileAddr+i*PageSize);
              }
              else{
              // in VM
              int k = pageTable[i].virtualPage;
              char* outBuffer = new char[PageSize];
              executable->ReadAt(outBuffer, PageSize, noffH.initData.inFileAddr+i*PageSize);
              kernel->swap->WriteSector(k, outBuffer);
              }
      }

     int initData_physAddr = TranslateVir2Phys(noffH.initData.virtualAddr);
     executable->ReadAt(&(kernel->machine->mainMemory[initData_physAddr]), noffH.initData.size, noffH.initData.inFileAddr);
    }
```

From now on, we are going to implement the class MemoryManager. In the constructor, we will get the type of page replacement.

```
//----------------------------------------------------------------------
// Memory::MemoryManager
//      Get the type of page replacement.
//----------------------------------------------------------------------
MemoryManager::MemoryManager(ReplacementType v){
        VicType=v;
}
```

Also, we have to calculate the offset and the physical page in the class. And then return the real physical address.

```
//----------------------------------------------------------------------
// MemoryManager::TransAddr()
//      Calculate the offset and the physical address
//----------------------------------------------------------------------
int MemoryManager::TransAddr(AddrSpace* space, int virAddr){
        int physAddr = 0;
        int vpn = (unsigned) virAddr / PageSize;
        int offset = (unsigned) virAddr % PageSize;
        ASSERT(space->pageTable[vpn].valid); // should be valid before tranformation
        physAddr = space->pageTable[vpn].physicalPage * PageSize + offset;
        return physAddr;
}
```

The function `AcquirePage()` will help us search whether there is a free frame, when we need to transfer data from disk to main memory. If so, we can directly put it into the frame table, and set all the parameters in the same way as that in function `Load`.

```
//----------------------------------------------------------------------
// MemoryManager::AcquirePage()
//      First search whether there is a free frame. Assume there exists
//      already a free frame. Ask a frame for vpn to transfer data from
//      disk to memory.
//----------------------------------------------------------------------
bool MemoryManager::AcquirePage(AddrSpace* space, int vpn){
        FrameInfoEntry* frameTable = kernel->frameTable;
        FrameInfoEntry* swapTable = kernel->swapTable;
        int k = space->pageTable[vpn].virtualPage; // index for swap table
        for(int j=0; j<NumPhysPages; j++){
                // find a free frame
           if(frameTable[j].valid == true){
                // update frame table: in use
                AddrSpace::UsedPhysicalPageState[j]=true;
                frameTable[j].valid = false;
                frameTable[j].addrspace = space;
                frameTable[j].vpn = vpn;
                frameTable[j].usageCount++;
                frameTable[j].latestTick = kernel->stats->totalTicks;

                // copy data from VM to frame
                char* inBuffer = new char[PageSize];
                kernel->swap->ReadSector(k, inBuffer);
                bcopy(inBuffer, &(kernel->machine->mainMemory[j*PageSize]), PageSize);

                // update page table
                space->pageTable[vpn].virtualPage = 1024;
                space->pageTable[vpn].physicalPage = j;
                space->pageTable[vpn].valid=true;

                // update swap table
                swapTable[k].addrspace = NULL;
                swapTable[k].valid=true;

                delete[] inBuffer;

                DEBUG(dbgAddr, "Occupied Physical Frame" << j);
                return true;
           }
        }

        // Exceed NumPhysPages 32, return false to indicate
        DEBUG(dbgAddr, "Exceed NumPhysPages");
        return false;
}
```

The function `ReleasePage()` will be executed when there is no free frame, i.e., the frame table is full. We will choose a frame based on page replacement algorithms. Then the entry of swap table and page table will become false (can't be used). And update (release) the frame table. Since the entry can be used, valid should be true and the addrspace should be NULL (no one in use). Reset all other parameter associated with page replacement to be zero. Lastly, delete the buffer.

```
//----------------------------------------------------------------------
// MemoryManager::Release()
//      If the frame table is full, we have to choose a frame based on
//      page replacement algorithm. Then swap that page from memory to
//      disk through this function.
//----------------------------------------------------------------------
bool MemoryManager::ReleasePage(AddrSpace* space, int vpn){
        FrameInfoEntry* frameTable = kernel->frameTable;
        FrameInfoEntry* swapTable = kernel->swapTable;
        int j = space->pageTable[vpn].physicalPage;
        for(int k=0; k<1024; k++){
           if(swapTable[k].valid==true){
                // update swap table
                swapTable[k].valid = false;
                swapTable[k].addrspace = space;
                swapTable[k].vpn = vpn;

                // update page table
                space->pageTable[vpn].valid = false;
                space->pageTable[vpn].virtualPage = k;
                space->pageTable[vpn].physicalPage = NumPhysPages;

                // copy data from memory to disk
                char* outBuffer = new char[PageSize];
                bcopy(&(kernel->machine->mainMemory[j*PageSize]), outBuffer, PageSize);
                kernel->swap->WriteSector(k, outBuffer);

                // update frame Table
                frameTable[j].valid = true;
                frameTable[j].addrspace = NULL;
                frameTable[j].latestTick = 0;

                for(int jj=0; jj<32; jj++){
                   frameTable[jj].usageCount = 0;
                }

                delete[] outBuffer;
//              DEBUG(dbgAddr, "Realease Frame " << j << " To VM " << k);
                return true;
           }
        }
//      DEBUG(dbgAddr, "Exceed Disk Sectors");
        return false;
}
```

The function `ChooseVictim` is the implementation of page replacement algorithm. For `Random`, we will randomly choose a frame from 32 frames. And for FIFO, we have define `latestTick` in table to get the time that an page is put into the frame table. Compare the time of all entries and get the minimum. Then the entry will be replaced.

```
//----------------------------------------------------------------------
// MemoryManager::ChooseVictim()
//      If page fault happens and there is no free frame, we have to
//      choose a frame to swap out with some algorithms.
//
//      random - swap out a frame from 32 frames randomly
//      LRU - swap out a frame that is least recently used
//      LFU - swap out a frame that is least frequently used
//      FIFO - swap out a frame that is first put in
//----------------------------------------------------------------------
int MemoryManager::ChooseVictim(){
        FrameInfoEntry* frameTable = kernel->frameTable;
        FrameInfoEntry* swapTable = kernel->swapTable;

        int vic_j=-1; // index for victim

        // random
        if(kernel->memoryManager->VicType==Random){
                vic_j = rand() % 32;
//              DEBUG(dbgPage, "Random Swapout " << vic_j);
        }
        // FCFS
        else if(kernel->memoryManager->VicType==FIFO){
                int minTick=0;
                int min_j;
                for (int j=0;j<NumPhysPages;j++){
//                DEBUG(dbgPage, "Frame " << j << " latestTick "<< frameTable[j].latestTick);
                  if(j==0){
                        min_j=0;
                        minTick=frameTable[j].latestTick;
                        }
                  else{
                        int curTick = frameTable[j].latestTick;
                        if(curTick<minTick){
                                minTick=curTick;
                                min_j=j;
                        }
                  }
                }
                vic_j = min_j;
//              DEBUG(dbgPage, "LRU Swapout " << vic_j);
        }

        return vic_j;
}
```

`PageFaultHandler()` links all other function in the class `MemoryManger`. When page fault happens, it will first check whether there is free frame through `AcquirePage()`. If not, choose a frame with `ChooseVictim` and release the frame with `ReleasePage`. And do `AcquirePage()` again.

```
//----------------------------------------------------------
// void MemoryManager::PageFaultHandler
//      When PageFaultException happens, this function
//      will help us to deal with page fault.
//----------------------------------------------------------
void MemoryManager::PageFaultHandler(int faultPageNum){
        DEBUG(dbgAddr, "Handling");
        TranslationEntry* pageTable = kernel->currentThread->space->pageTable;
        FrameInfoEntry* frameTable = kernel->frameTable;
        FrameInfoEntry* swapTable = kernel->swapTable;

        int k = pageTable[faultPageNum].virtualPage;

        // No free frame
        // release a frame
        while(AcquirePage(kernel->currentThread->space, faultPageNum)==false){
                int j_vic = ChooseVictim();
                AddrSpace* addr_vic = frameTable[j_vic].addrspace;
                int vpn_vic = frameTable[j_vic].vpn;
                bool releaseSuccess = ReleasePage(addr_vic, vpn_vic);
                ASSERT(releaseSuccess);
        }
        AcquirePage(kernel->currentThread->space, faultPageNum);
}
```

`../userprog/userkernel.cc` is the initialization and cleanup routines for the the version of the Nachos kernel that supports running user program. And the constructor will interpret command line arguments in order to determine flags for the initialization. So we have to add some codes so that kernel could obtain which page replacement algorithm is. Also, we should initialize the frame table and swap table in `UserProgKrenel::Initialize()`. Set all the tables to be free. Then initialize the `memoryManager` with constructor.

```
// HW3 added
else if (strcmp(argv[i], "-vic") == 0){
   ASSERT(i + 1 < argc);
   char* vicArg = argv[i + 1];
   if (strcmp(vicArg, "random") == 0) VicType = Random;
   else if (strcmp(vicArg, "fifo") == 0) VicType = FIFO;
   else if (strcmp(vicArg, "lfu") == 0) VicType = LFU;
   else VicType = Random;
   i++;
}
```

```
// HW3 added
swap = new SynchDisk("SWAPSPACE");
frameTable = new FrameInfoEntry[32];
for(int i=0; i<32; i++){
    frameTable[i].valid = true;
    frameTable[i].lock = false;
    frameTable[i].addrspace = NULL;
    frameTable[i].vpn = 0;
    frameTable[i].latestTick = 0;
    frameTable[i].usageCount = 0;
}

swapTable = new FrameInfoEntry[1024];
for(int i=0; i<1024; i++){
    swapTable[i].valid = true;
    swapTable[i].lock = false;
    swapTable[i].addrspace = NULL;
    swapTable[i].vpn = 0;
}

memoryManager = new MemoryManager(VicType);
```

We also have to define the frame table and the swap table in the header file. To initialize the page replacement type, we should declare some variables `VicType` and `memoryManager`.

```
20 #include "addrspace.h" // HW3 added


// HW3 added
SynchDisk *swap;
FrameInfoEntry* frameTable;
FrameInfoEntry* swapTable;
MemoryManager* memoryManager;
ReplacementType VicType;
```

We have define `noffH` in `../userprog/addrspace.h`. To prevent compile error for redeclaration (compile error), we should add if statement in `../bin/noff.h`. If `noffH` has not been defined, it will be defined here.

```
 9 #ifndef NOFF_H
10 #define NOFF_H
11
12 #define NOFFMAGIC       0xbadfad        /* magic number denoting Nachos
13                                          * object code file
14                                          */
15
16 typedef struct segment {
17   int virtualAddr;              /* location of segment in virt addr space */
18   int inFileAddr;               /* location of segment in this file */
19   int size;                     /* size of segment */
20 } Segment;
21
22 typedef struct noffHeader {
23    int noffMagic;               /* should be NOFFMAGIC */
24    Segment code;                /* executable code segment */
25    Segment initData;            /* initialized data segment */
26    Segment uninitData;          /* uninitialized data segment --
27                                  * should be zero'ed before use
28                                  */
29 } NoffHeader;
30
31 #endif
```

3 Result

- Random:

```
gina@gina-VirtualBox:~/repo/Nachos/nachos-4.0/code/userprog$ ./nachos -e ../test
/sort -e ../test/matmult -vic random
Total threads number is 2
Thread ../test/sort is executing.
Thread ../test/matmult is executing.
return value:1023
return value:7220
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 25874052, idle 8139608, system 49820, user 17684624
Disk I/O: reads 828, writes 831
Console I/O: reads 0, writes 0
Paging: faults 828
Network I/O: packets received 0, sent 0
```

- FCFS (FIFO):

```
gina@gina-VirtualBox:~/repo/Nachos/nachos-4.0/code/userprog$ ./nachos -e ../test
/sort -e ../test/matmult -vic fifo
Total threads number is 2
Thread ../test/sort is executing.
Thread ../test/matmult is executing.
return value:1023
return value:7220
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 62338052, idle 44455714, system 195290, user 17687048
Disk I/O: reads 3252, writes 3256
Console I/O: reads 0, writes 0
Paging: faults 3252
Network I/O: packets received 0, sent 0
```

- Without assigning the type:

```
gina@gina-VirtualBox:~/repo/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:1023
return value:7220
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 25874052, idle 8139608, system 49820, user 17684624
Disk I/O: reads 828, writes 831
Console I/O: reads 0, writes 0
Paging: faults 828
Network I/O: packets received 0, sent 0
```

`../test/sort` will return 1023 and `../test/matmult` will return 7220. It seems that our result is correct. When we don't assign the type, the result will be same as that of the type random.

4 Problem

1. Why would the error in the following picture occurs?

```
../userprog/exception.cc:102:2: error: jump to case label [-fpermissive]
  default:
  ^
```

   A: Declaration of new variables in case statements is what causing problems. There are two solutions. One is to add a bracket for each case.

The other is to declare the variable before switch statement. Here, I choose the second solution.

2. Why don't we implement LRU and LFU algorithm in this project? A: If we want to implement the algorithms mentioned in this question, we have to obtain when every page reference happens. To achieve this goal, we have to add some code in `Machine::Translate()` in `../machine/translate.cc`. Nevertheless, doing so is illegal. So we just implement other algorithms that can be done in `../userprog/addrspace.cc`.

5 References

- co-work with R11942140 Yu-Wei, Chang
- Work from Predecessor