r06921048_project3

電機碩二　李友岐

1. Motivation and the problem analysis:

Memory management is a critical part of operating system. If the physical memory is not enough, then we need to use virtual memory to help. In this project, we have two cases, which are test/sort and test/matmult. Both of them require larger memory than the original physical memory. Our goal is to execute them concurrently and generate correct result.

2. What's your plan to deal with the problem (high-level):

When the space of main memory is not enough, we should put the pages into virtual memory。Therefore, we need a pageTable to record. Besides, we use two arrays to record which parts of physical memory and which parts of virtual memory have been used. For the page replacement, we have four algorithms, which are Random, FIFO, Second chance and LRU. In addition, there are 32 physical pages in this project.

To implement Random replacement, we randomly choose a page from 32 physical pages each time. To implement FIFO algorithm, we initialize a variable named FIFO to zero and increase it by one each time. The corresponding number of the victim page is FIFO%32. Second chance algorithm is similar to FIFO algorithm but we add a reference bit to each page. Initially, the reference bit is set to one. When finding a victim page, we follow the order of FIFO. If the reference bit of the page we find is one, then we set the bit to zero and check the next page. If the reference bit of the page we find is zero, then this page is our victim page. To implement LRU, we add a count variable to each page and we need to update this variable for each paging iteration. The count variable of the most recently used page is 31 while the count variable of the least recently used page is 0. As a result, the victim page is the page whose count variable equal to zero.

3. You can including some important code segments and comments:

File: userprog/userkernel.h
Function: class UserProgKernel

```
class UserProgKernel : public ThreadedKernel {
    public:
        UserProgKernel(int argc, char **argv);
                        *
                        *
                        *
        SynchDisk *vmswap; // used for virtual memory
```

We add a data member vmswap into class UserProgKernel. It is used to save the memory if the physical memory is not enough.

File: machine/machine.h
Function: class Machine

```
TranslationEntry *tlb;              // this pointer should be considered
                                    // "read-only" to Nachos kernel code

TranslationEntry *pageTable;
unsigned int pageTableSize;
bool ReadMem(int addr, int size, int* value);
bool isPhyPageUsed[NumPhysPages];   //record which physical page is used
bool isVirPageUsed[NumPhysPages];   //record which virtual page is used
bool ReferenceBit[NumPhysPages];    // used for Second change algorithm
int Count[NumPhysPages];            // used for LRU algorithm
int PhyPageName[NumPhysPages];      //
int PhyPageNumber[NumPhysPages];
int SectorNumber;                   // record which sector of the disk is used
int Id_;
TranslationEntry * Main[NumPhysPages];
```

We add some data members in class Machine. The funcion of these memebers is writen in comment.

File: userprog/addrspace.cc
Function: AddrSpace::Load()

```
// then, copy in the code and data segments into memory
    if (noffH.code.size > 0) {

            for(unsigned int i =0, j=0 ; i<numPages ; i++){

                while(kernel->machine->isPhyPageUsed[j]==TRUE ){//find the unused physical page
                    j++;
                    if (j>=NumPhysPages){
                        break;
                    }
                }
                //if physical memory is enough, we put data directly in without virtual memory
                if( j < NumPhysPages){
                    kernel->machine->PhyPageName[j]= Id_;// record id to physical page
                    kernel->machine->isPhyPageUsed[j] = TRUE;//record this physical page has been used
                    kernel->machine->Main[j]= &pageTable[i];//record the address
                    pageTable[i].physicalPage = j;//record the corresponding physical page
                    pageTable[i].Id_=Id_;
                    pageTable[i].valid = TRUE;
                    pageTable[i].use = FALSE;
                    pageTable[i].dirty = FALSE;
                    pageTable[i].readOnly = FALSE;

                    executable->ReadAt(&(kernel->machine->mainMemory[ PageSize*j ]), PageSize,
                                       noffH.code.inFileAddr+( PageSize*i ));

                }
```

```
    else{ // if physical memory is not enough, then we use virtual memory

        char *virMemory = new char[PageSize];
        unsigned int k=0;
        while (kernel->machine->isVirPageUsed[k]==TRUE){//find the unused virtual page
               k++;
        }
        kernel->machine->isVirPageUsed[k] = TRUE;//record this virtual page has been used
        pageTable[i].virtualPage=k;//record the corresponding virtual page
        pageTable[i].Id_=Id_;
        pageTable[i].valid=FALSE;
        pageTable[i].use=FALSE;
        pageTable[i].dirty=FALSE;
        pageTable[i].readOnly=FALSE;
        executable->ReadAt( virMemory, PageSize, noffH.code.inFileAddr+(PageSize*i ));
        kernel->vmswap->WriteSector(k, virMemory); //Write in virtual memory
    }
```

In AddrSpace::Load(), we add an if-else statement to deal with two situations. One is that the physical memory is enough, then we don't need virtual memory. The other one is that the physical memory is not enough, then we need the virtual memory. In addition, the data members of pageTable[i] need to be updated.

File: userprog/addrspace.cc
Function: AddrSpace::SaveState()

```
if( isPageTableLoad == TRUE){
        pageTable=kernel->machine->pageTable;
        numPages=kernel->machine->pageTableSize;
    }
```

We use a variable isPageTableLoad to check whether the pageTable is loaded. If the pageTable is loaded, then we save the state. If not, we do nothing. Thus, some errors can be avoided.

File: machine/translate.cc
Function: Machine::Translate()

```
// from the virtual address
    vpn = (unsigned) virtAddr / PageSize;
    offset = (unsigned) virtAddr % PageSize;

    if (tlb == NULL) {          // => page table => vpn is index into table
        if (vpn >= pageTableSize) {
            DEBUG(dbgAddr, "Illegal virtual page # " << virtAddr);
            return AddressErrorException;
        }
        else if (!pageTable[vpn].valid) {

            printf("page fault\n");
            kernel->stats->numPageFaults++;
            j=0;
            while(kernel->machine->isPhyPageUsed[j] ==TRUE ){//find the unused physical page
                  j++;
                  if (j>= NumPhysPages){
                     break;
                  }
            }
```

In Machine::Translate(), if page fault occurs, we first find the unused physical page.

```
    //add the page into the main memory if the physical memory is not full
    if(j<NumPhysPages){

        char * buf = new char[PageSize];

        kernel->machine->PhyPageName[j]=pageTable[vpn].Id_;
        kernel->machine->Main[j]=&pageTable[vpn];
        kernel->machine->isPhyPageUsed[j]=TRUE;//record this physical page has been used

        pageTable[vpn].physicalPage = j;//record the corresponding physical page
        pageTable[vpn].valid = TRUE;

        int old=kernel->machine->Count[j];
        kernel->machine->Count[j]=32;
        for (int p=0;p<32;p++){
           // update the count variable for LRU
           if( kernel->machine->Count[p]>old){
               kernel->machine->Count[p]--;
           }
        }

        kernel->machine->ReferenceBit[j] = TRUE; //for second chance algorithm

        kernel->vmswap->ReadSector(pageTable[vpn].virtualPage, buf);
        bcopy(buf,&mainMemory[PageSize*j],PageSize);
    }
```

If the physical page is not full, we can directly put data into physical page.

```
else{// if physical memory is not enough, then we do page replacement

    char * buf_1 = new char[PageSize];
    char * buf_2 = new char[PageSize];
```

```
page replacement part
```

```
printf("page #%d swap out\n", VictimPage);

//get the page victim and save it to disk
bcopy(&mainMemory[PageSize*VictimPage],buf_1,PageSize);
kernel->vmswap->ReadSector(pageTable[vpn].virtualPage,buf_2);
bcopy(buf_2,&mainMemory[PageSize*VictimPage],PageSize);
kernel->vmswap->WriteSector(pageTable[vpn].virtualPage,buf_1);

Main[VictimPage]->virtualPage=pageTable[vpn].virtualPage;
Main[VictimPage]->valid=FALSE;

//save the page into the main memory
pageTable[vpn].valid=TRUE;// update the validity
pageTable[vpn].physicalPage=VictimPage;//record the corresponding physical page
kernel->machine->PhyPageName[VictimPage]=pageTable[vpn].Id_;
Main[VictimPage]= &pageTable[vpn];
printf("Page Replacement Done\n");
```

If the physical page is not enough, then we need to do page replacement and save the data through virtual memory.

page replacement part: (we have four page replacement algorithm to choose)

```
//Fifo

VictimPage = FIFO %32;//find the victim page by FIFO
FIFO++;  // update FIFO
cout<<"By FIFO, ";
```

The above is FIFO algorithm. We have 32 pages so the index ranges from 0 to 31. We use a variable FIFO to represent index. FIFO is increased by one after each page replacement. Since FIFO would be larger than 31 if we keep increasing it by one, we use FIFO%32 to make it range from 0 to 31. Thus, the order is 0, 1, 2, ….., 30, 31, 0, 1, …….

```
//Random

VictimPage = rand()%32; //find the victim page randomly
cout<<"By Random, ";
```

The above is Random algorithm. We use rand()%32 to choose an index out of 0~31 randomly. Thus, the victim page is random.

```
//Second chance

FIFO=FIFO%32;//FIFO should be in range 0~31
//find the page with reference bit zero
while(kernel->machine->ReferenceBit[FIFO] == TRUE){
   cout<<"Page#: "<<FIFO<<"   ";
   cout<<"Reference Bit: "<<kernel->machine->ReferenceBit[FIFO];
   kernel->machine->ReferenceBit[FIFO] = FALSE;//if bit is 1, then set reference bit to 0
   FIFO++;  //try next page
   FIFO = FIFO % 32;

}
VictimPage =FIFO;//the resulted page is our victim
cout<<"Page#: "<<FIFO<<"   ";
cout<<"Reference Bit: "<<kernel->machine->ReferenceBit[FIFO];

kernel->machine->ReferenceBit[VictimPage] = TRUE;//set the reference bit back to 1
FIFO++;// update FIFO
cout<<"By Second chance, ";
```

The above is Second chance algorithm. The order of index of victim page is similar to FIFO but each page has a corresponding reference bit. Every reference bit is set to 1 initially. If the reference bit of the selected page is 1, then we set this bit to 0 and check next page. If the reference bit of the selected page is 0, then this page is the victim page and we set this bit back to 1. We implement this algorithm by a while loop. Besides, we should update the corresponding reference bit when the page fault not occur.

```
 //LRU

 VictimPage=0;
 for (int num=0;num<32;num++){
      if (kernel->machine->Count[num]>=32){
             kernel->machine->Count[num]%=32;
      }
      cout<<"Page#:"<<num<<"  Count:"<< kernel->machine->Count[num]<<endl;
      if ( kernel->machine->Count[num]==0){//find the least recently used page
            VictimPage=num; // denote this page as victim
            kernel->machine->Count[num]=31; //update this page as most recently used page

      }
      else {
            kernel->machine->Count[num]--; // update the other pages count
      }

}
cout<<"By LRU, ";
```

The above is Least Recently Used (LRU) algorithm. Each page has a count variable to record the recent usage. The count range from 0 to 31, where 0 means the least recently used and 31 means the most recently used page. To choose victim page, we use a for loop to find the page whose index equal to 0. We then update victim page's count variable to 31 since it becomes the most recently used. In addition, we update the count variable of other pages by decreasing it by 1 (e.g. 1→ 0, 2→ 1, 3→ 2). We should also update the count variables when the page fault not occur.

4. Experiment result and some discussion:

Part 1: Executing test/matmult by different page replacement algortihms

To test the performance of different page replacement algorithm, we first move to /nachos-4.0/code/, and then enter the command: ./userprog/nachos -e test/matmult
Since the output of executing test/matmult is extremely large, we only provide partial screenshot.

1. Random:

```
page fault
By Random, page #15 swap out
Page Replacement Done
page fault
By Random, page #24 swap out
Page Replacement Done
page fault
By Random, page #28 swap out
Page Replacement Done
page fault
By Random, page #10 swap out
Page Replacement Done
```

```
page fault
By Random, page #12 swap out
Page Replacement Done
page fault
By Random, page #16 swap out
Page Replacement Done
page fault
By Random, page #27 swap out
Page Replacement Done
page fault
By Random, page #27 swap out
Page Replacement Done
page fault
By Random, page #18 swap out
Page Replacement Done
return value:7220
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 7651580, idle 1325676, system 6325900, user 4
Disk I/O: reads 89, writes 111
Console I/O: reads 0, writes 0
Total Paging faults: 89
```

We can find that the number of victim page is randomly chosen from 0~31. Thus, our implementation is correct. The total paging faults are 89.

2. FIFO:

```
page fault
By FIFO, page #26 swap out
Page Replacement Done
page fault
By FIFO, page #27 swap out
Page Replacement Done
page fault
By FIFO, page #28 swap out
Page Replacement Done
page fault
By FIFO, page #29 swap out
Page Replacement Done
page fault
By FIFO, page #30 swap out
Page Replacement Done
```

```
page fault
By FIFO, page #31 swap out
Page Replacement Done
page fault
By FIFO, page #0 swap out
Page Replacement Done
page fault
By FIFO, page #1 swap out
Page Replacement Done
page fault
By FIFO, page #2 swap out
Page Replacement Done
page fault
By FIFO, page #3 swap out
Page Replacement Done
page fault
By FIFO, page #4 swap out
Page Replacement Done
return value:7220
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 7526030, idle 1200066, system 6325960, user 4
Disk I/O: reads 90, writes 112
Console I/O: reads 0, writes 0
Total Paging faults: 90
```

We can find that the number of victim page is follow the cyclic order of 0~31. Thus, our implementation is correct. The total paging fault are 90.

3. Second chance:

```
page fault
Page#: 21  Reference Bit: 1
Page#: 22  Reference Bit: 0
By Second chance, page #22 swap out
Page Replacement Done
page fault
Page#: 23  Reference Bit: 0
By Second chance, page #23 swap out
Page Replacement Done
page fault
Page#: 24  Reference Bit: 1
Page#: 25  Reference Bit: 0
By Second chance, page #25 swap out
Page Replacement Done
page fault
Page#: 26  Reference Bit: 0
By Second chance, page #26 swap out
Page Replacement Done
```

```
page fault
Page#: 27  Reference Bit: 1
Page#: 28  Reference Bit: 1
Page#: 29  Reference Bit: 1
Page#: 30  Reference Bit: 1
Page#: 31  Reference Bit: 1
Page#: 0  Reference Bit: 1
Page#: 1  Reference Bit: 1
Page#: 2  Reference Bit: 1
Page#: 3  Reference Bit: 0
By Second chance, page #3 swap out
Page Replacement Done
page fault
Page#: 4  Reference Bit: 0
By Second chance, page #4 swap out
Page Replacement Done
return value:7220
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 7333030, idle 1008386, system 6324640, user 4
Disk I/O: reads 68, writes 90
Console I/O: reads 0, writes 0
Total Paging faults: 68
```

We can find that the number of victim page is follow the same order of FIFO but only when the reference bit of the page equal to 0, the page would be swap out. Otherwise, we try next page. Thus, our implementation is correct. The total paging faults are 68.

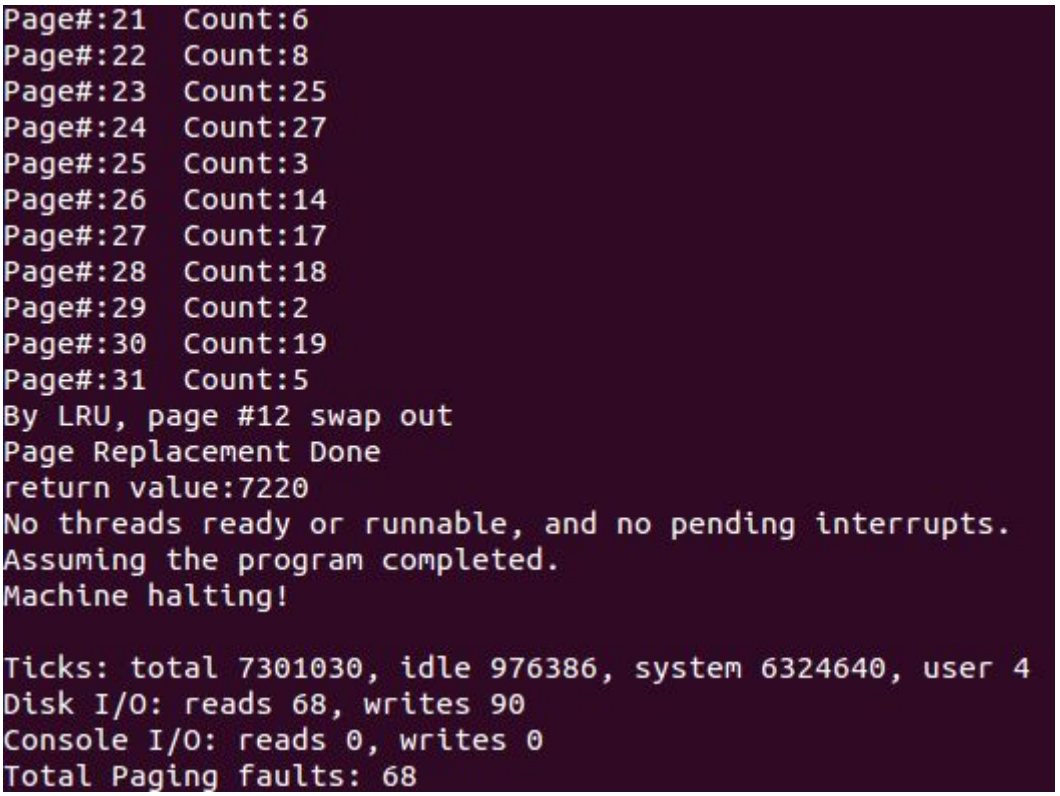
4. LRU:

```
Page#:9  Count:23
Page#:10  Count:0
Page#:11  Count:22
Page#:12  Count:1
Page#:13  Count:24
Page#:14  Count:29
Page#:15  Count:31
Page#:16  Count:11
Page#:17  Count:12
Page#:18  Count:14
Page#:19  Count:15
Page#:20  Count:5
Page#:21  Count:7
Page#:22  Count:9
Page#:23  Count:30
Page#:24  Count:27
Page#:25  Count:4
Page#:26  Count:13
Page#:27  Count:16
Page#:28  Count:17
Page#:29  Count:3
Page#:30  Count:18
Page#:31  Count:6
By LRU, page #10 swap out
```

The Count of page#10 is 0, so page#10 is the victim.

```
page fault
Page#:0  Count:7
Page#:1  Count:20
Page#:2  Count:28
Page#:3  Count:1
Page#:4  Count:11
Page#:5  Count:29
Page#:6  Count:21
Page#:7  Count:22
Page#:8  Count:9
Page#:9  Count:10
Page#:10  Count:30
Page#:11  Count:23
Page#:12  Count:0
Page#:13  Count:26
Page#:14  Count:24
Page#:15  Count:31
Page#:16  Count:12
Page#:17  Count:13
Page#:18  Count:15
Page#:19  Count:16
Page#:20  Count:4
```

The Count of page#12 is 0, so page#12 is the victim.



We can find that the victim page is the page that has zero count number, which means it is the least recently used. Thus, our implementation is correct. The total paging faults are 68.

Part 2: Paging faults under different commands

To further compare the total paging faults of different algorithms when executing different commands, we generate the below table. Exp1 is executing test/matmult and Exp2 is executing test/sort. Exp3 and Exp4 are executing test/matmult and test/sort concurrently but with different order. The number in the table are the total paging faults under that condition. Our goal of project 3 is to run test/matmult and test/sort concurrently and get the correct result. Since we are able to finish Exp3 and Exp4, the goal is certainly achieved.

	Exp1 -e test/matmult	Exp2 -e test/sort	Exp3 -e test/matmult -e test/sort	Exp4 -e test/sort -e test/matmult
Random	89	1262	1361	1396
FIFO	90	5702	5801	5811
Second Chance	68	4999	5119	5084
LRU	68	4999	4886	5080

Exp1 command: ./userprog/nachos -e test/matmult
Exp2 command: ./userprog/nachos -e test/sort
Exp3 command: ./userprog/nachos -e test/matmult -e test/sort
Exp4 command: ./userprog/nachos -e test/sort -e test/matmult

Based on our knowledge, the performance between FIFO, Second chance and LRU should be {1. LRU , 2. Second chance , 3. FIFO }. LRU should results in fewest paging faults out of these three algorithms and our result is consistent with this knowledge. As for Random, it should have similar performance to FIFO. In Exp1, Random and FIFO has about same paging faults, which are 89 and 90, respectively. In Exp2, Exp3 and Exp4, however, it is such a surprise that Random has way fewer paging faults than not only FIFO but also LRU and Second chance. Obviously, test/sort is a way larger program than test/matmult. We can speculate that the phenomenon of Exp2, Exp3 and Exp4 are all because of test/sort. We think test/sort might have some characteristics specially fit with Random. Therefore, Random can generate such few paging faults.

5. How to switch to another page replacement algorithm:

The code of page replacement is in the file machine/translate.cc and the function Machine::Translate(). We have four page replacement algorithms, which are FIFO, Random, Second Chance and LRU. If we want to apply this algorithm, then we uncomment the code of this algorithm and comment the code of other three algorithms. Right now, we use Second chance algorithm so the code of FIFO, Random and LRU are commented just like below.

```
        else{// if physical memory is not enough, then we do page replacement

            char * buf_1 = new char[PageSize];
            char * buf_2 = new char[PageSize];

            //Fifo
/*
                        // start
            VictimPage = FIFO %32;//find the victim page by FIFO
            FIFO++;  // update FIFO
            cout<<"By FIFO, ";
                        // end
*/
            //Random
 /*
                        // start
            VictimPage = rand()%32; //find the victim page randomly
            cout<<"By Random, ";
                        // end
 */
```

```
            //LRU
/*
                        // start
            VictimPage=0;
            for (int num=0;num<32;num++){
                 if (kernel->machine->Count[num]>=32){
                        kernel->machine->Count[num]%=32;
                 }
                 cout<<"Page#:"<<num<<"  Count:"<< kernel->machine->Count[num]<<endl;
                 if ( kernel->machine->Count[num]==0){//find the least recently used page
                       VictimPage=num; // denote this page as victim
                       kernel->machine->Count[num]=31; //update this page as most recently used page

                 }
                 else {
                       kernel->machine->Count[num]--; // update the other pages count
                 }

            }
            cout<<"By LRU, ";
                        // end
 */
```

```
            //Second chance

                // start
            FIFO=FIFO%32;//FIFO should be in range 0~31
            //find the page with reference bit zero
            while(kernel->machine->ReferenceBit[FIFO] == TRUE){
               cout<<"Page#: "<<FIFO<<"  ";
               cout<<"Reference Bit: "<<kernel->machine->ReferenceBit[FIFO]<<endl;
               kernel->machine->ReferenceBit[FIFO] = FALSE;//if bit is 1, then set reference bit to 0
               FIFO++;  //try next page
               FIFO = FIFO % 32;

            }
            VictimPage =FIFO;//the resulted page is our victim
            cout<<"Page#: "<<FIFO<<"  ";
            cout<<"Reference Bit: "<<kernel->machine->ReferenceBit[FIFO]<<endl;

            kernel->machine->ReferenceBit[VictimPage] = TRUE;//set the reference bit back to 1
            FIFO++;// update FIFO
            cout<<"By Second chance, ";
                  // end
```

If we want to change the algorithm to FIFO, then we can simply uncomment the code of FIFO and comment the code of Second chance, Random and LRU just like below.

```
            //Fifo

                        // start
            VictimPage = FIFO %32;//find the victim page by FIFO
            FIFO++;  // update FIFO
            cout<<"By FIFO, ";
                        // end
```

```
            //Random
/*
                        // start
            VictimPage = rand()%32; //find the victim page randomly
            cout<<"By Random, ";
                        // end
*/
```

```
//LRU
/*
                // start
VictimPage=0;
for (int num=0;num<32;num++){
     if (kernel->machine->Count[num]>=32){
          kernel->machine->Count[num]%=32;
     }
     cout<<"Page#:"<<num<<"  Count:"<< kernel->machine->Count[num]<<endl;
     if ( kernel->machine->Count[num]==0){//find the least recently used page
          VictimPage=num; // denote this page as victim
          kernel->machine->Count[num]=31; //update this page as most recently used page

     }
     else {
          kernel->machine->Count[num]--; // update the other pages count
     }

}
cout<<"By LRU, ";
              // end
 */
```

```
//Second chance
/*
     // start
FIFO=FIFO%32;//FIFO should be in range 0~31
//find the page with reference bit zero
while(kernel->machine->ReferenceBit[FIFO] == TRUE){
   cout<<"Page#: "<<FIFO<<"  ";
   cout<<"Reference Bit: "<<kernel->machine->ReferenceBit[FIFO]<<endl;
   kernel->machine->ReferenceBit[FIFO] = FALSE;//if bit is 1, then set reference bit to 0
   FIFO++;  //try next page
   FIFO = FIFO % 32;

}
VictimPage =FIFO;//the resulted page is our victim
cout<<"Page#: "<<FIFO<<"  ";
cout<<"Reference Bit: "<<kernel->machine->ReferenceBit[FIFO]<<endl;

kernel->machine->ReferenceBit[VictimPage] = TRUE;//set the reference bit back to 1
FIFO++;// update FIFO
cout<<"By Second chance, ";
      // end
*/
```

If we want to change the algorithm to Random or LRU, the approach is similar. Just uncomment the code of Random or LRU and make the unused three algorithm commented.

Random:

```
//Fifo
/*
                // start
VictimPage = FIFO %32;//find the victim page by FIFO
FIFO++;  // update FIFO
cout<<"By FIFO, ";
                // end
*/
```

```
//Random

                // start
VictimPage = rand()%32; //find the victim page randomly
cout<<"By Random, ";
                // end
```

```
                //LRU
 /*
                        // start
                VictimPage=0;
                for (int num=0;num<32;num++){
                     if (kernel->machine->Count[num]>=32){
                            kernel->machine->Count[num]%=32;
                     }
                     cout<<"Page#:"<<num<<"  Count:"<< kernel->machine->Count[num]<<endl;
                     if ( kernel->machine->Count[num]==0){//find the least recently used page
                           VictimPage=num; // denote this page as victim
                           kernel->machine->Count[num]=31; //update this page as most recently used page

                     }
                     else {
                           kernel->machine->Count[num]--; // update the other pages count
                     }

               }
               cout<<"By LRU, ";
                            // end
  */
```

```
                //Second chance
/*
                     // start
                FIFO=FIFO%32;//FIFO should be in range 0~31
                //find the page with reference bit zero
                while(kernel->machine->ReferenceBit[FIFO] == TRUE){
                   cout<<"Page#: "<<FIFO<<"  ";
                   cout<<"Reference Bit: "<<kernel->machine->ReferenceBit[FIFO]<<endl;
                   kernel->machine->ReferenceBit[FIFO] = FALSE;//if bit is 1, then set reference bit to 0
                   FIFO++;  //try next page
                   FIFO = FIFO % 32;

                }
                VictimPage =FIFO;//the resulted page is our victim
                cout<<"Page#: "<<FIFO<<"  ";
                cout<<"Reference Bit: "<<kernel->machine->ReferenceBit[FIFO]<<endl;

                kernel->machine->ReferenceBit[VictimPage] = TRUE;//set the reference bit back to 1
                FIFO++;// update FIFO
                cout<<"By Second chance, ";
                     // end
*/
```

LRU:

```
           else{// if physical memory is not enough, then we do page replacement

               char * buf_1 = new char[PageSize];
               char * buf_2 = new char[PageSize];

               //Fifo
 /*
                              // start
               VictimPage = FIFO %32;//find the victim page by FIFO
               FIFO++;  // update FIFO
               cout<<"By FIFO, ";
                              // end
 */
               //Random
  /*
                             // start
               VictimPage = rand()%32; //find the victim page randomly
               cout<<"By Random, ";
                             // end
  */
```

```
//LRU

            // start
VictimPage=0;
for (int num=0;num<32;num++){
     if (kernel->machine->Count[num]>=32){
            kernel->machine->Count[num]%=32;
     }
     cout<<"Page#:"<<num<<"  Count:"<< kernel->machine->Count[num]<<endl;
     if ( kernel->machine->Count[num]==0){//find the least recently used page
           VictimPage=num; // denote this page as victim
           kernel->machine->Count[num]=31; //update this page as most recently used page

     }
     else {
           kernel->machine->Count[num]--; // update the other pages count
     }

}
cout<<"By LRU, ";
            // end
```

```
//Second chance
/*
     // start
FIFO=FIFO%32;//FIFO should be in range 0~31
//find the page with reference bit zero
while(kernel->machine->ReferenceBit[FIFO] == TRUE){
   cout<<"Page#: "<<FIFO<<"  ";
   cout<<"Reference Bit: "<<kernel->machine->ReferenceBit[FIFO]<<endl;
   kernel->machine->ReferenceBit[FIFO] = FALSE;//if bit is 1, then set reference bit to 0
   FIFO++;  //try next page
   FIFO = FIFO % 32;

}
VictimPage =FIFO;//the resulted page is our victim
cout<<"Page#: "<<FIFO<<"  ";
cout<<"Reference Bit: "<<kernel->machine->ReferenceBit[FIFO]<<endl;

kernel->machine->ReferenceBit[VictimPage] = TRUE;//set the reference bit back to 1
FIFO++;// update FIFO
cout<<"By Second chance, ";
      // end
*/
```