COMP25111 Exercise 3:

Contains a copy of the "traceA" file. Note read the "Lab Rules" [on Blackboard] as you must use **SUBMIT** & **LABPRINT** for Lab 1, 2 & 3.
If you use Netbeans it is on all Lab PC :- Where to Find NetBeans on the lab machines:
Boot into Windows 7; 1) Click "Start;" 2) Click "All programs;" 3) Click the "NetBeans" folder [icon]; 4) Click the "NetBeans IDE 7.0.1" icon.

Simulation of Paging Behaviour

Duration: 2 Session

1. Learning Outcomes

On completion of this exercise, a student will:

- Have implemented the functionality of a simple Memory Management Unit (MMU) for a paged virtual memory system in Java.
- Have exercised the MMU using a trace file of memory accesses using a Java program which initialises the MMU and calls methods within it to perform read and write actions.
- Have produced statistics, which show the behaviour of the MMU with particular reference to page faults and page rejections.

2. Introduction

Virtual memory is a technique for providing a process with apparent access to the whole addressable memory space of a processor in circumstances where the real memory available may be considerably smaller. One advantages of the scheme is the process can use larger code and data sizes than would otherwise be possible. Another is the run-time layout of code and data can be considerably simplified if it does not have to fit into a confined space. This is particularly true for dynamic data whose size is unknown until run-time.

Virtual memory relies on the fact that, in the majority of programs, the use of both code and data exhibits both *spatial locality* and *temporal locality*. That is that at any point in time, the program is usually only using a small fraction of its code and data and that, over a short time period, the usage will not change significantly.

A virtual memory system therefore tries to keep currently used code and data in the (limited size) real memory of the system while the rest (if it exists) is kept on background storage such as magnetic disk. In order to relieve the programmer from the complex task of working out which data should be where at any point in time, the system arranges to move the data between real memory and disk automatically as required.

A piece of hardware know as a Memory Management Unit (MMU), together with system software within the Operating System provide all the functionality needed to do this. The purpose of this exercise is to implement the functionality of a MMU in order to gain an understanding of the principles.

3. Paged Virtual Memory

Devices such as disks usually store and retrieve data in blocks of hundreds or thousands of bytes. Accessing a block is often slow compared to CPU memory speeds

but once accessed; the contents of that block can be read or written very rapidly. In these circumstances, it makes sense to move data in units which are larger than bytes or CPU words. Virtual memory systems therefore operate at the unit of a *page* when transferring data to and from memory and disk. A page does not necessarily correspond to a disk block size but is fixed for a particular MMU typically between 4 and 64 kilobytes. This will usually correspond to several disk blocks.

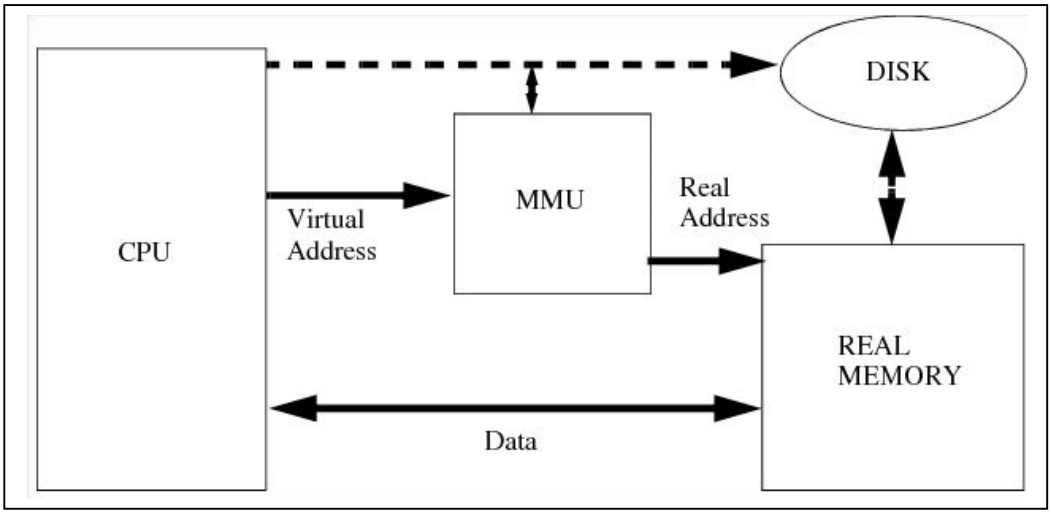

Figure 1 shows the basic structure of a paged virtual memory system.

The addresses issued by the CPU are virtual addresses which are the full width of the processor's addressing capability. In a modern 32 bit processor (for example ARM or x86) these will be a full 32 bits capable of addressing 4Gbytes of memory. The real memory will usually be smaller, typically 512 Mbytes on a laptop.

Each of these memory spaces is considered to be divided into pages. Note that this is not reflected in the internal structure of any memory, it is just an abstract division; the memory addresses are still binary values which cover a linear address space (usually at the unit of bytes). However, we can view any address as being divided into two sections, one which addresses the page and one which is an offset within the page which addresses individual units (bytes). By convention, we usually refer to the virtual address as having *virtual page numbers* (or just page numbers) and the real address as having *page frame numbers.* Figure 2 shows this pictorially.

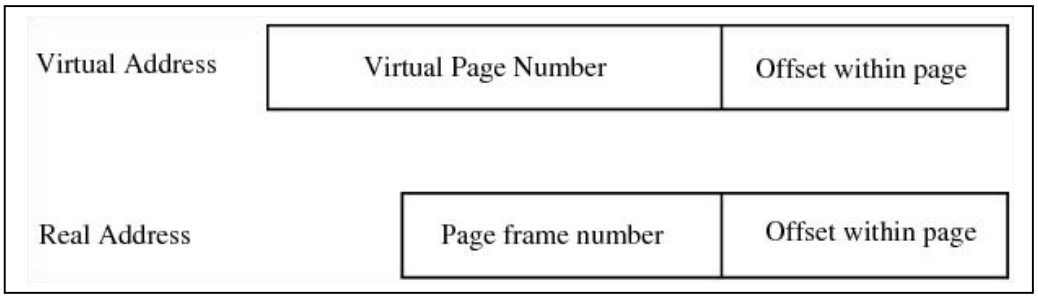

Figure 2 shows this pictorially view of *virtual page numbers* and *page frame numbers*.

The function of the MMU is to keep track of which virtual pages exist in real memory and, which on disk. It does this by keeping a *page table* of every possible virtual page number which contains the page frame numbers of any real copies. It is important to note that the offset is common to both virtual and real addresses. In order to find the data associated with a virtual address it is only necessary to take the virtual page number and replace it by the page frame number (if it exists). The page frame number is looked up in the page table to produce the real address which can then be used to access real memory. This is usually referred to as *address translation.*

If the table indicates that a virtual page does not exist in real memory this is called a *page fault.* It is necessary for the MMU to initiate a transfer of the page's data from the background storage (disk). The exact mechanism for this is not relevant here but it should be noted that this is a relatively slow process and the CPU itself can organise most of the transfer.

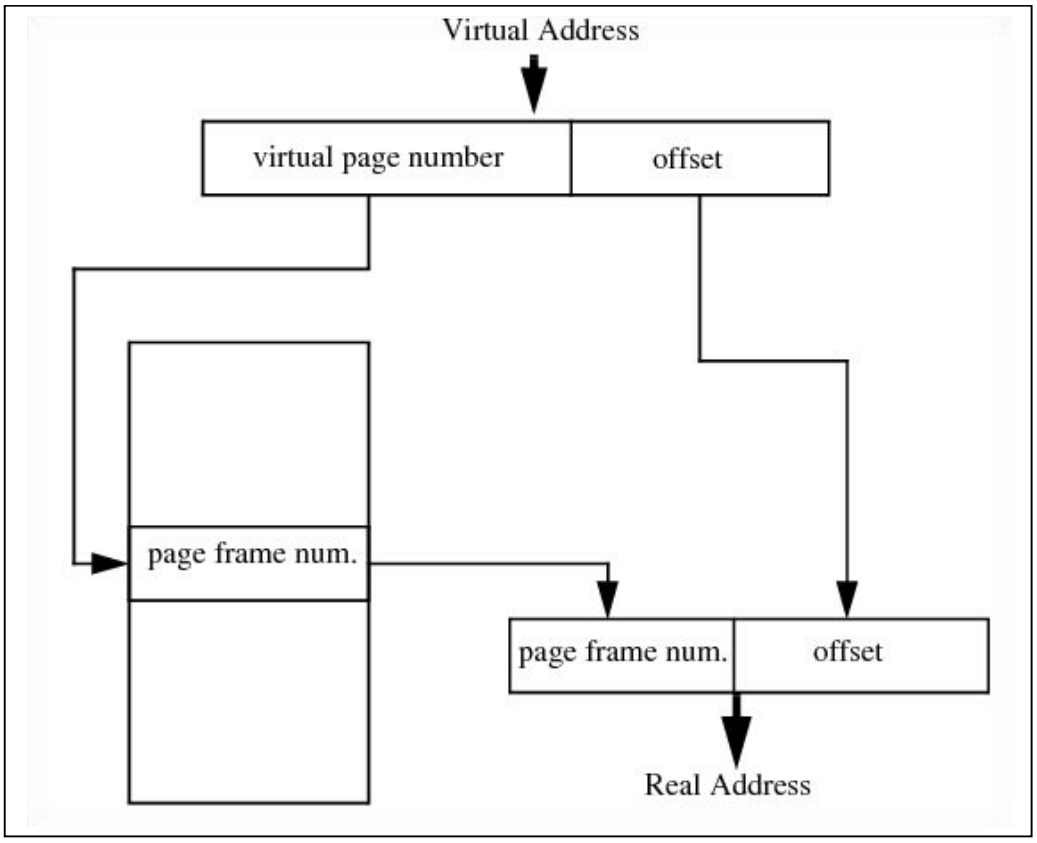

Figure 3 shows more detail of the MMU table and the address translation operation.

The page table is shown here as a single linear structure with the lookup done by the full page number. For large virtual address spaces, this may not be practical and more complex techniques may be required (this is covered in the course lectures).

4. Page Rejection

If there is a page fault, a place needs to be found in real memory for the page copy. Initially, assuming all real memory is not in use, it is possible to keep allocating page frames linearly through the address space. However, if all real memory is in use, we must find a page to reject. We must then write its contents back to disk before reallocating its page frame number to the page access, which has caused the fault.

We need to decide which page to choose using a *page rejection algorithm.* The most commonly used is an algorithm called LRU (least recently used). It works on the simple principle that the page, which has survived the longest time without being accessed, is a good candidate for replacement.
One useful optimisation is to note if a page has been written to since it was bought from disk into real memory. If not, there is no need to write it back to disk before reusing the real page frame. This is often true for pages which contain code.

5. The Exercise

The exercise is composed of two parts:

Part One: implements the majority of the code and the LRU page replacement algorithm. The majority of the (theoretical) advice below will aid you in this task; after you have designed, developed, and implemented the program utilise the two trace files: **traceA** and **traceB**; to test your program.

Part Two: implements the a second algorithm the FIFO page replacement algorithm. The advice below will not aid you in this task. Nevertheless, it is expected that once you have implemented (and tested) the LRU algorithm you will have gained sufficient knowledge to enable you to undertake the FIFO design, development and testing on your own. After you have designed, developed, and implemented the program utilise the trace file: **traceLRUandFIFO**; to test your program. The FIFO algorithm has been covered in your lecture series and it is covered extensively in your course books; also see the Learning Resources (book & Web resources) as well as the Background Reading (book & Web resources) at the end of Lecture COMP25111 Operating Systems Lectures 14: Virtual Memory (3) for further reference material.

Introduction to Exercise

The purpose of the exercise is to write Java code to implement a simple MMU, which contains a page table and code to perform the necessary actions when read, or write accesses occur to a virtual memory address. Because the exercise is using a trace of addresses rather than executing a real program, there is no need to perform any real memory accesses. Instead, it is simply necessary to access the page table to determine if there is a page fault and, if so, find a free page frame to use and place it in the table. If, initially, there are unused page frames these can be allocated linearly until there are none free. At that point it will be necessary to find a page to reject and use its page frame for the newly required page.

You should implement a LRU algorithm to determine page rejection. To make this simple, you should have a ‘timestamp’ in each page table entry, which is updated appropriately, when a page is accessed.

You should also include a ‘dirty’ indication in the page table implementation, which is set on a write. This indicates if the page needs to be written back to disk if it is rejected.

Note that a practical page table will usually also contain page access information which determines the types of access which are allowed to a particular page (for

example read only). There is no need to include such information for the purposes of this exercise.

In appropriate Blackboard subfolder, you will find a program harness which, after creating a MMU object, reads the trace of store accesses and makes calls to read and write methods within that object until the trace file ends. It then prints out some statistics:

Total Accesses
Total Instruction Fetches
Total Page Faults
Total Page Rejections (i.e. ignoring startup page faults)
Total Page Writebacks (to disk)

The first two are included in the given program; the last three need to be calculated by your MMU implementation. (A skeleton MMU.java file is provided in the appropriate Blackboard subfolder)

The trace file format is a line for each access. The first number indicates the access type (0=read, 1=write, 2=fetch) and the second is a (hex) 24 bit memory address. Within the program, the address is considered to be a 12-bit page number and a 12 bit offset representing 4k pages each of 4k bytes, i.e. 16Mbytes in total of virtual memory space. This is deliberately small so that your implementation can use a simple single level page table. Although obviously not representative of modern systems, this is typical of a virtual memory system in computers of the 1960s when virtual memory was invented (in Manchester University on the Atlas computer [1][2]).

An example of 'traceA' file is given below:

2 400000
0 600000
2 400001
1 600001
2 400002
0 700000
2 500000
0 700001

The first line "2 400000" is decoded as follows:

First number	Second number
2	400000
2=fetch	$0100\ 0000\ 0000\ 0000\ 0000\ 0000_{HEX}$
This is a fetch access	24 bit hexadecimal memory address

The following decode the first number of each of the eight lines in traceA:

First number	Type of access
2	2=fetch
0	0=read
2	2=fetch
1	1=write
2	2=fetch
0	0=read

2	2=fetch
0	0=read

The program given makes a single call to the method which performs the trace simulation using a very very small real memory (2 [real] page frames, i.e. 8k bytes [in MMUSim.java the real memory sizes (in pages) is set to "rmem_pages = 2;"). This should be run with the file traceA when initially debugging your implementation. This file contains only 8 store accesses which have been devised so that you should be able to predict what statistics your MMU should produce.

When you think this is working, you should modify the program to perform simulations using both 32 and 64 real page frames on traceB. This represents 128k and 256k bytes of real memory, again realistic in the 1960s.

6. Results

You should be prepared to demonstrate the results of the runs on traceA and traceB when your exercise is marked. To help you determine if you have got a correct implementation, the total number of page faults for 32 pages and traceB should be 227.

You should also be prepared to describe your code and draw relevant conclusions from the results.

Assessment

COMP20051 Exercise 3: Developing Simulation of Paging Behaviour

Demonstrators please fill in the student's full detail [below] before starting the assessment; may be the best way to do this is to get the students to fill this in themselves:

Students full name	Student Number	Email address	Course studied

At the deadline, at the end of the Session, the marks awarded are as given in table 1.
Demonstrators please assign marks for each of the eight questions in boxes provided.

At the deadline, at the end of Session 5, the marks awarded are as follows:

Mark awarded for:	Exercise 3
1. Show Correct page table [data] structure; plus comment on how you [the student] developed the data structure.	1
Demonstrating an overall [software engineering] correct approach; of the algorithmic and combined data structure: **2 – 5**.	
2. State how REQUIREMENT development was undertaken.	1
3. State how [if any] ABSTRACT DESIGN was undertaken.	1
4. State how IMPLEMENTATION was undertaken.	1
5. Demonstrating a methodology for TESTING the program; and comment on: **interpretation of results**.	1
6. Showing the generation of an appropriate OUTPUT trace file: **traceA** files [displayed in real-time on command [Prompt] screen] (using LRU).	1
7. Showing the generation of an appropriate OUTPUT trace file: **traceB** files [displayed in real-time on command [Prompt] screen] (using LRU).	2
8. Showing the generation of an appropriate OUTPUT trace file: **traceLRUandFIFO** files [displayed in real-time on command [Prompt] screen] (using LRU and FIFO).	2
TOTAL mark	**10**

Remember to save these exercises in a directory nominally named …/COMP20051/ex3.

Do this exercise [or save this exercise] in a directory named COMP251111/ex3 directory, save your downloaded template [Skeleton(s)] code:-

MMUSim.java; and
MMU.java.

as well as the trace files:

traceA;
traceB; and
traceLRUandFIFO

… in COMP25111/ex3.

Remember to save your trace files of your solution.

The three files traceA; traceB; and traceLRUandFIFO; as well as the template [Skeleton(s)] MMUSim.java & MMU.java are downloadable from Blackboard in Folder Lab3.

References

[1] One-Level Storage System, T. Kilburn, D.B.G. Edwards, M.J. Lanigan, F.H. Sumner, IRE Trans. Electronic Computers April 1962

[2] Tom Kilburn (1956), The Atlas, School of Computer Science: information on Tom Kilburn's [computing] effort known as the MUSE (microsecond) computer , available [on-line] @ http://www.computer50.org/kgill/atlas/atlas.html, [Last accessed 12/10/09, 11:09].

7. Decoding the COMP20051 Laboratory Exercise 3; and '*supporting advice*'

The previous six pages document is a typical laboratory that you are normally set in the School of Computer Science. The first task is to decode [work out] what explicitly is required. As you have read the document once now reread section 5; this time highlighting or extracting the main requirements of the exercise.

Hint 1: [#H1]

[#H1] If you require a hint, download them from Blackboard. 'Hint No 1' hints at the requirements you should have data mined from reread section 5.

After reading document 'H1.doc' to validate you have extracted the relevant requirements you next step is to start the design phase that fulfils the requirements [#R1 to #R5] of the exercise. Then once designed the MMU class can be implemented.

Hint 2: [#H2]

[#H2] If you require a hint, download them from Blackboard. After you have compiled a list of requirements, you should now think about designing the MMU class. 'Hint No 2' suggests an approach you could take.

Reading document 'H2.doc' may help you move forward in the design phase or validate your design.

You should now be able to implement, test, and run MMUSim utilising traceA. Once you have reached this stage you can compare your program's trace output with a typical trace A output [result] available on Blackboard; the file is named 'traceAres.'

One of the final requirements of the exercise is to modify the program [MMUSim] to perform simulations using both 32 and 64 real page frames on traceB. To do this you should consider altering the number of real memory pages.

Finally, you should consider the best way to describe your code; when asked. In addition, you should think about drawing relevant conclusions from the results.

The ">>" operator

Final point: what does the operator ">>" actually do?

In the code snippet:

int addr = access.addr;

int vpage_no = addr >> 24 - vmem_bits; // assumes a 24 bit virt addr
*given vmem_bits = 12. Hence the actual value of '24 - vmem_bits' is "**addr >> 12**."*

In "doSimul(…)" in class MMUSim.Java. The operator ">>" is utilised. If we look at what happend

Values:
addr = 4194304
vpage_no = 1024

So what has happened?

The java operator "**>>**"

>> shift bits right with sign extension

It implies [with two objects '**x**' & '**y**']:

x >> y

Means: Shift Right - Signed

Explicitly means: Shift **x** to the right by **y** bits. Low order bits are lost.
Same bit value as sign (0 for positive numbers, 1 for negative) fills in the left bits.

Given the example addr = 4194304
which is specified for addr an interger (int) which is base (or radix) 10.

Converting 4194304_{10}
Converting to hexadecimal:
00400000_{16}
or radix 2 it is:
$0000\ 0000\ 0100\ 0000\ 0000\ 0000\ 0000\ 0000_2$

Hence given ***addr >> 12***." After shifting right 12 places this is:

$0000\ 0000\ 0000\ 0000\ 0000\ 0100\ 0000\ 0000_2$
which is:
00000400_{16}
or radix 2 it is:
1024_{10}

QED