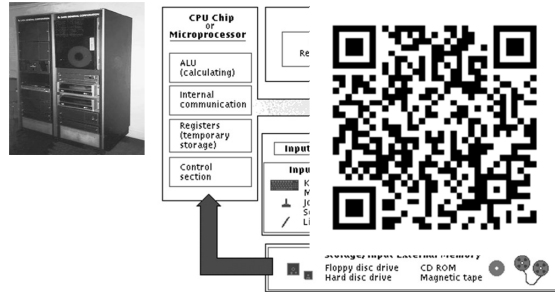


COMP25111

Operating Systems

Lectures 11

Virtual Memory (1)



© Copyright Richard Neville 2007

Dr Richard Neville
r.neville@manchester.ac.uk

Room: G12 Kilburn Building, Bottom floor

Week

7

NOTE: The up-to-date version of this lecture is kept on the associated web site -- available [on-line] @ Blackboard select: COMP25111 Introduction to Computer Systems www.manchester.ac.uk/portal

<http://www.cs.man.ac.uk/~neville/COMP25111/Lecture11full/Lecture11full.html>

1

This week's

Short Exam Questions

Q1

1. Question

What is the difference between a 'partition' and a 'program.'

ANSWER(S):

Answer(s):

© Copyright Richard Neville 2007

NOTE: In the exam approximately 2 question are taken from the topics (and program examples) covered in each lecture

Lecture 11

Virtual Memory (1)

"The equal-sized blocks concept"

Paged Virtual Memory

© Copyright Richard Neville 2007

Virtual memory – *"makes the machine appear to have more memory than it actually has."*

Normally this is not the case: virtual = 500G (secondary [hard disk]) & physical = 4G (primary [RAM]).

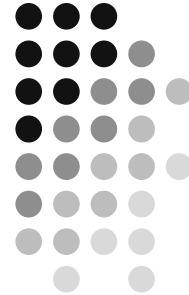
Quotation

Pages – *"paging gives the programmer the illusion of large, continuous, linear main memory, the same size as the virtual address space," [1].*

References:

[1] Quote form: Structured Computer Organization, 5/E, Andrew S. Tanenbaum, Vrije University, Amsterdam, The Netherlands, ISBN-10: 0131485210, ISBN-13: 9780131485211, Publisher: Prentice.

3



Li Learning; comprehension; & introspection Learning Outcomes

D-words

C-words

Be able to

- 1) Explain what is meant by a paged virtual memory system;
- 2) Determine the structure of an address in a paged virtual memory system; &
- 3) Establish the outcome of memory references (specified by address) for a paged virtual memory system with a specific page table.
- 4) Self-study NOTES: [Information] No. of bits in Page and Offset (At the end of the lecture notes)

© Copyright Richard Neville 2007

Footnote

- 1: Describe: aligned to paraphrase, discuss & give example [pictorial or diagrammatic].
- 2: Determine: aligned to decide upon, find out.
- 3: Establish: aligned to find out, calculate & demonstrate method.

FOOTNOTE2: D-words: direction words. C-Words: content words.
Ref.: Michael J. Wallace (1980, 2004)
Study Skills in English, ISBN 9780521537520.
D-Words also aligned to; Ref.: Taxonomy of Educational Objectives:
The Classification of Educational Goals; pp. 201–207; B. S. Bloom (Ed.)
Susan Fauer Company, Inc. 1956.

Reference: Bloom, B.S., *Taxonomy of Educational Objectives*. Handbook I: The Cognitive Domain. 1956: New York: David McKay Co Inc.

4

Virtual Memory

- Virtual Memory is provided in a computer system for two reasons: Large Add. → Smaller Add.

1) To allow a processor to address a much larger address space than is implemented by the physical memory.

- For example, a modern processor may have a 32-bit address bus, giving access to $2^{32} = 4\text{GB}$ of memory. In a practical system only 512MB of physical memory may be present.

32 Bits

29 Bits

2) To support the operating system in the management of processes.

- Especially in ensuring that processes do not interfere with one another.

These are the two reasons why Virtual Memory is provided in a computer system.

● Start

○ M

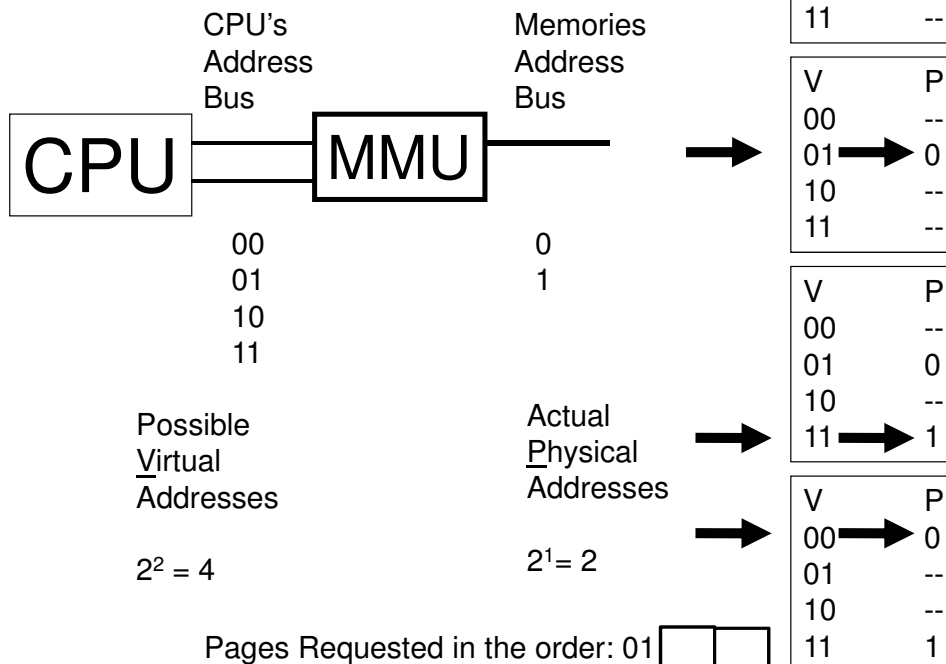
● E

5

Re. **Learning Resources 13.1**, available at the end of the lecture notes.

© Copyright Richard Neville 2007

Toy Example



6

© Copyright Richard Neville 2007

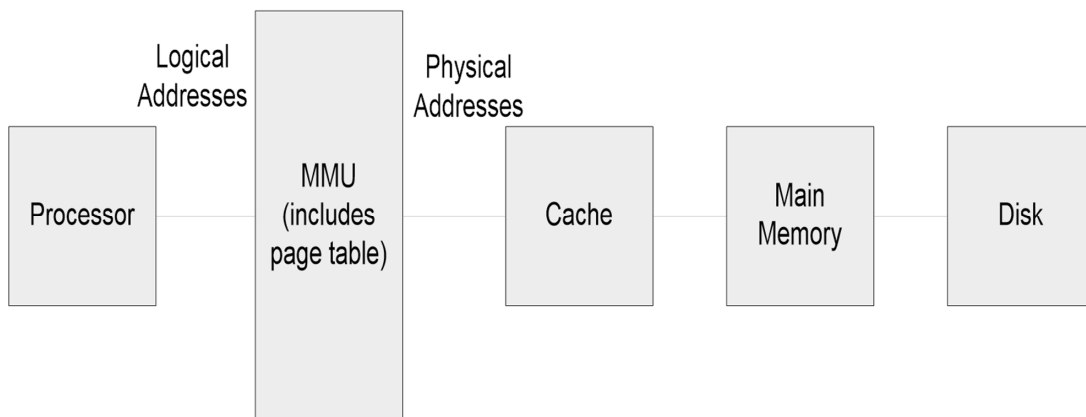
Memory Management Unit (MMU)

- The MMU is hardware that translates a logical address provided by the processor into a physical address into main memory.
- The MMU is hardware that is placed between the processor and the memory system.
- It uses a table (the page table) to translate from a logical to a physical address.
- The page table is maintained by the operating system.
- The MMU must carry out the translation very quickly because the address of all memory reads and writes are routed through it.

© Copyright Richard Neville 2007

7

Memory Management Unit (MMU)



© Copyright Richard Neville 2007

8

Virtual Memory Methods

A few additions to notes are required.

- There are two major methods for implementing virtual memory:

1) Paged Virtual Memory

- The pages are of fixed size
 - 512 B to 64 KB are common

2)

- The segments are of **variable** size
 - between 1 and 2^{16} or 2^{32} bytes

Paged Virtual Memory

- The virtual address space is divided into a number of equal-sized pages (blocks of memory).

'Page' in virtual address space

- The virtual address space is given by the number of **bits** in the address bus of a processor:
- So, if a processor has a **32-bit address**, the virtual address space is **2^{32} Bytes** or **4 GB**.

Paged Virtual Memory, cont...

- The physical address space is divided into a **smaller** number of page frames, each of which can hold a page.

'Page frame' in physical address space

- Thus a page frame is the **same size as a page**.
- In the implementation of a computer system, the physical memory will often be **smaller** than the virtual address space.
- For example, 256 MB may be available when the processor's virtual address space is 4GB.

© Copyright Richard Neville 2007

Quotation

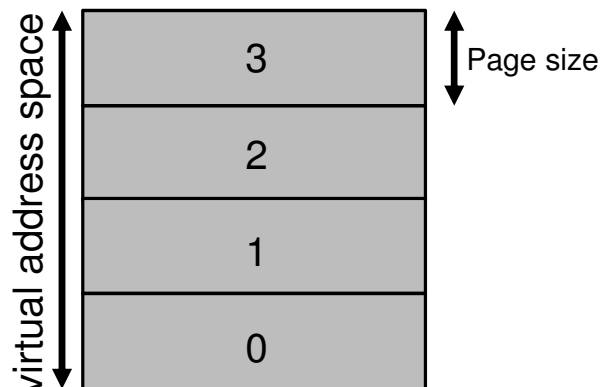
Page frames – *"pieces of main memory into which the pages go are called page frames."*

References 1: Quote from: Structured Computer Organization, 5/E, Andrew S. Tanenbaum, Vrije University, Amsterdam, The Netherlands, ISBN-10: 0131485210, ISBN-13: 9780131485211, Publisher: Prentice.

11

Paged Virtual Memory (1)

- QUESTION: How many pages in the virtual address space ?
- Virtual Memory



© Copyright Richard Neville 2007

12

Paged Virtual Memory (1)

1) The virtual address space will consist of a number of pages:

$$\text{Number of pages} = \frac{\text{Address space}}{\text{Page size}}$$

Given a virtual memory 'address space' of 4GB and a 'page size' of 64KB:

$$= \frac{4 \text{ GB}}{64 \text{ KB}} = \frac{2^{32}}{2^{16}} = 2^{16}$$

e.g. if the virtual address space is 4 GB and the page size is 64 KB there are:

(64K) pages.

$$2^{32} = 4,294,967,296 \text{ or } 4\text{G}$$

$$2^{16} = 65,536 \text{ or } 64\text{K}$$

L Learning; comprehension; & introspection

Aside: on units

- When referring to units that are derived from the binary number system, the following are used:

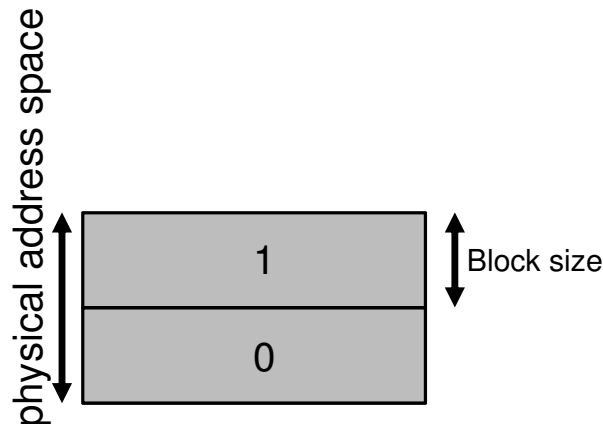
| | | | |
|-------------|---------------------------|---------------------------|---------------------------|
| $2^0 = 1$ | $2^{10} \sim 1\text{K}$ | $2^{20} \sim 1\text{M}$ | $2^{30} \sim 1\text{G}$ |
| $2^1 = 2$ | $2^{11} \sim 2\text{K}$ | $2^{21} \sim 2\text{M}$ | $2^{31} \sim 2\text{G}$ |
| $2^2 = 4$ | $2^{12} \sim 4\text{K}$ | $2^{22} \sim 4\text{M}$ | $2^{32} \sim 4\text{G}$ |
| $2^3 = 8$ | $2^{13} \sim 8\text{K}$ | $2^{23} \sim 8\text{M}$ | $2^{33} \sim 8\text{G}$ |
| $2^4 = 16$ | $2^{14} \sim 16\text{K}$ | $2^{24} \sim 16\text{M}$ | $2^{34} \sim 16\text{G}$ |
| $2^5 = 32$ | $2^{15} \sim 32\text{K}$ | $2^{25} \sim 32\text{M}$ | $2^{35} \sim 32\text{G}$ |
| $2^6 = 64$ | $2^{16} \sim 64\text{K}$ | $2^{26} \sim 64\text{M}$ | $2^{36} \sim 64\text{G}$ |
| $2^7 = 128$ | $2^{17} \sim 128\text{K}$ | $2^{27} \sim 128\text{M}$ | $2^{37} \sim 128\text{G}$ |
| $2^8 = 256$ | $2^{18} \sim 256\text{K}$ | $2^{28} \sim 256\text{M}$ | $2^{38} \sim 256\text{G}$ |
| $2^9 = 512$ | $2^{19} \sim 512\text{K}$ | $2^{29} \sim 512\text{M}$ | $2^{39} \sim 512\text{G}$ |

K = Kilo M = Mega G = Giga

| | |
|----|--------------------|
| 0 | 1.00 |
| 1 | 2.00 |
| 2 | 4.00 |
| 3 | 8.00 |
| 4 | 16.00 |
| 5 | 32.00 |
| 6 | 64.00 |
| 7 | 128.00 |
| 8 | 256.00 |
| 9 | 512.00 |
| 10 | 1,024.00 |
| 11 | 2,048.00 |
| 12 | 4,096.00 |
| 13 | 8,192.00 |
| 14 | 16,384.00 |
| 15 | 32,768.00 |
| 16 | 65,536.00 |
| 17 | 131,072.00 |
| 18 | 262,144.00 |
| 19 | 524,288.00 |
| 20 | 1,048,576.00 |
| 21 | 2,097,152.00 |
| 22 | 4,194,304.00 |
| 23 | 8,388,608.00 |
| 24 | 16,777,216.00 |
| 25 | 33,554,432.00 |
| 26 | 67,108,864.00 |
| 27 | 134,217,728.00 |
| 28 | 268,435,456.00 |
| 29 | 536,870,912.00 |
| 30 | 1,073,741,824.00 |
| 31 | 2,147,483,648.00 |
| 32 | 4,294,967,296.00 |
| 33 | 8,589,934,592.00 |
| 34 | 17,179,869,184.00 |
| 35 | 34,359,738,368.00 |
| 36 | 68,719,476,736.00 |
| 37 | 137,438,953,472.00 |
| 38 | 274,877,906,944.00 |
| 39 | 549,755,813,888.00 |

Paged Virtual Memory (1)

- QUESTION: How many page frames in the physical address space ?
- Physical Memory



© Copyright Richard Neville 2007

15

Paged Virtual Memory (2)

- 2) The physical address space will consist of a smaller number of page frames:

$$\text{Number of page frames} = \frac{\text{Address space}}{\text{Block size}}$$

Given a physical memory 'address space' of 256 MB and a 'block size' of 64KB:

$$= \frac{256 \text{ MB}}{64 \text{ KB}} = \frac{2^{28}}{2^{16}} = 2^{12}$$

e.g. if the physical address space is 256 MB and the block size is 64 KB there are:

(4K) page frames.

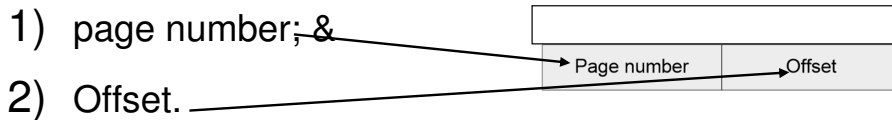
$$\begin{aligned} 2^{28} &= 268,435,456 \text{ or } 256 \text{ M} \\ 2^{16} &= 65,536 \text{ or } 64 \text{ K} \\ 2^{12} &= 4096 \text{ or } 4 \text{ K} \end{aligned}$$

© Copyright Richard Neville 2007

16

Paged Virtual Memory

- The logical address generated by the processor is split into two bit fields:



- The number of bits in the page number are given by:
- $\log_2(nv)$
- where
- nv = number of pages in the virtual address space.
- So, for our example of 4 GB of virtual address space and a page size of 64 KB, there are 2^{16} pages (previous slide) giving $\log_2(2^{16}) = 16$ bits

© Copyright Richard Neville 2007

17

$$2^{16} = 65,536 \text{ or } 64K$$

Paged Virtual Memory

- The number of bits in the offset is:

- $\log_2(bs)$
- Where bs = block size.

$$2^{16} = 65,536 \text{ or } 64K$$

- So in our example, with a 64 KB page size there are $\log_2(2^{16}) = 16$ bits in the offset.
- So address is treated as:



© Copyright Richard Neville 2007

18

Paged Virtual Memory Procedure

- This procedure can be viewed as a sequence of steps:



- The processor generates a logical address.
- The page number field is used by the MMU to look to see whether the page is in memory or not.
- If it is in memory, a physical address is computed by replacing the page number with the page frame number of where the page can be found.

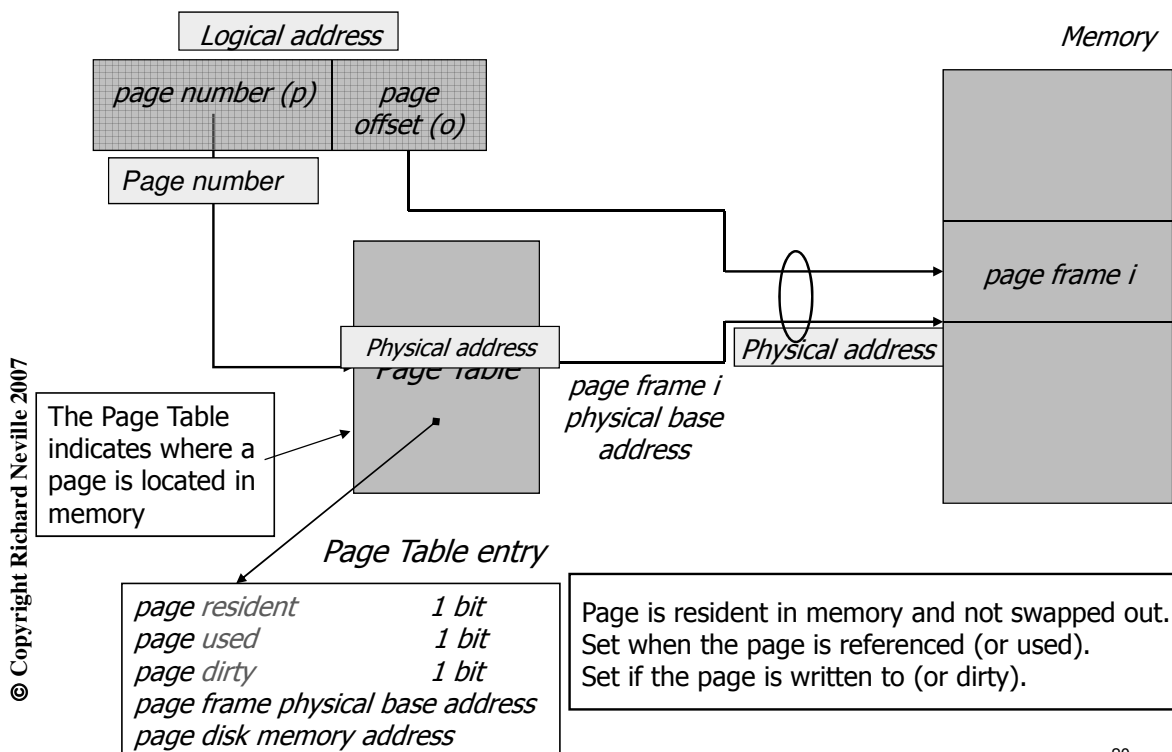
Together with the offset this is used as a physical address to memory.

- If it is not in memory, the transfer is aborted (page fault) and the operating system will load the page from disk to memory.

© Copyright Richard Neville 2007

19

Paged Virtual Memory Address Mapping



© Copyright Richard Neville 2007

Plus other permission information (R,W,X).

20

CPU

4 G Virtual

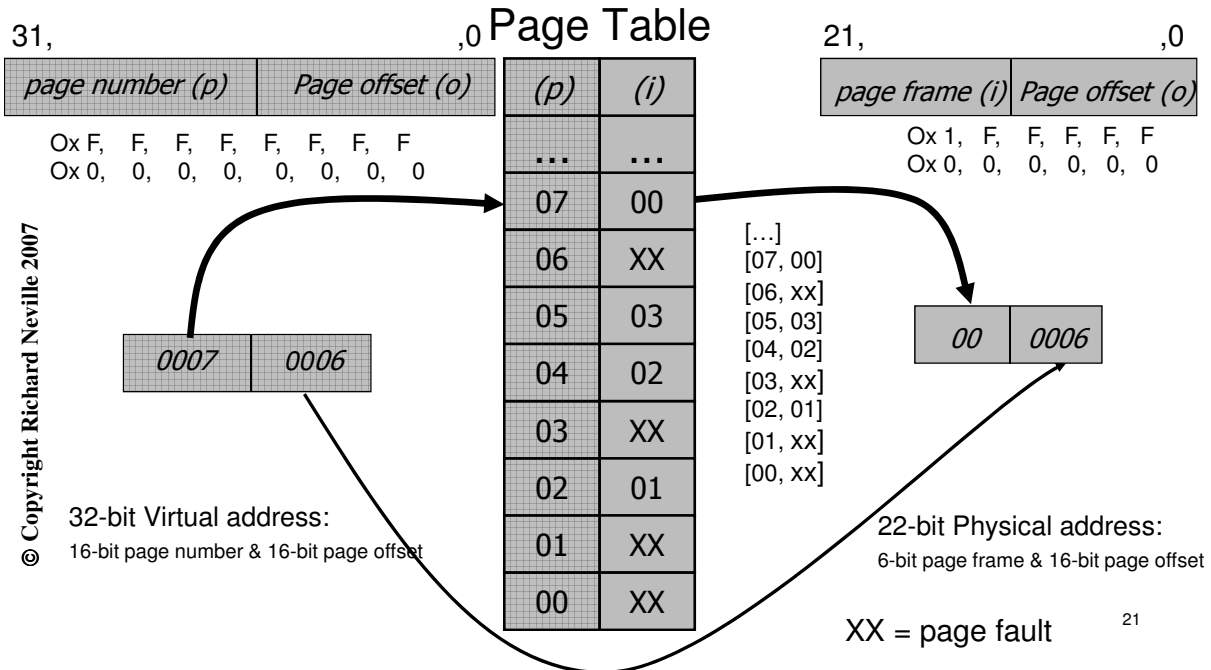
→ translates to

Simple page table

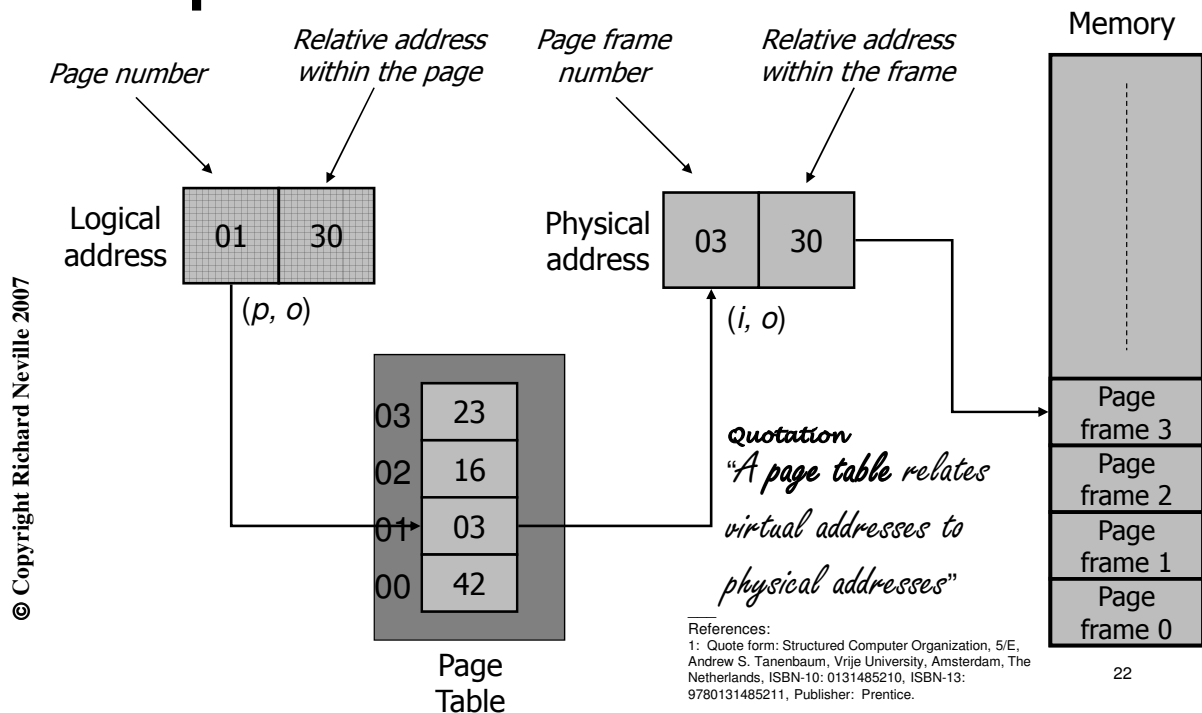
Cache

4 M Physical

32 (128 K Page frames)



Logical to Physical Address Simple View



A few additions to notes are required.

- When a reference is made to a page that is not currently in the memory, a page fault occurs.
- This is like an interrupt; initiating the following:
 - 1) The current process will be halted;
 - 2) The operating system will cause the page to be loaded from disk into a page frame;
 - Whilst waiting for the disk, other processes can run in a multitasking operating system.
 - 3) When the page has been loaded, the memory reference will be tried again.

© Copyright Richard Neville 2007

•

23

- After the computer has been operating for a while, each of the page frames in memory will be occupied by a page.
- When another page needs to be loaded, one of the pages already loaded will need to be overwritten.
- An algorithm will need to be used to determine which page is to be replaced:
 - LRU and FIFO algorithms are commonly used.

© Copyright Richard Neville 2007

24

Page Change

© Copyright Richard Neville 2007

- When the processor writes to memory, the contents of the page will change.
- We need to decide when this needs to be written back to the disk;
 - This is similar to the write strategies for caches.
- In the case of virtual memory, it makes no sense to use a write-through strategy because the time taken to write to disk is so high, so a write-back strategy is used with pages only being written to disk if they have changed and when they are unloaded
- Each page table entry has a dirty bit that indicates whether the page has been written to or not:
 - If it has been written to, the page on the disk needs to be updated when the page is unloaded from the physical memory;
 - If a page has not been written to, there is no need to update the page's data held on the disk.

25

Simple Example of Page Table

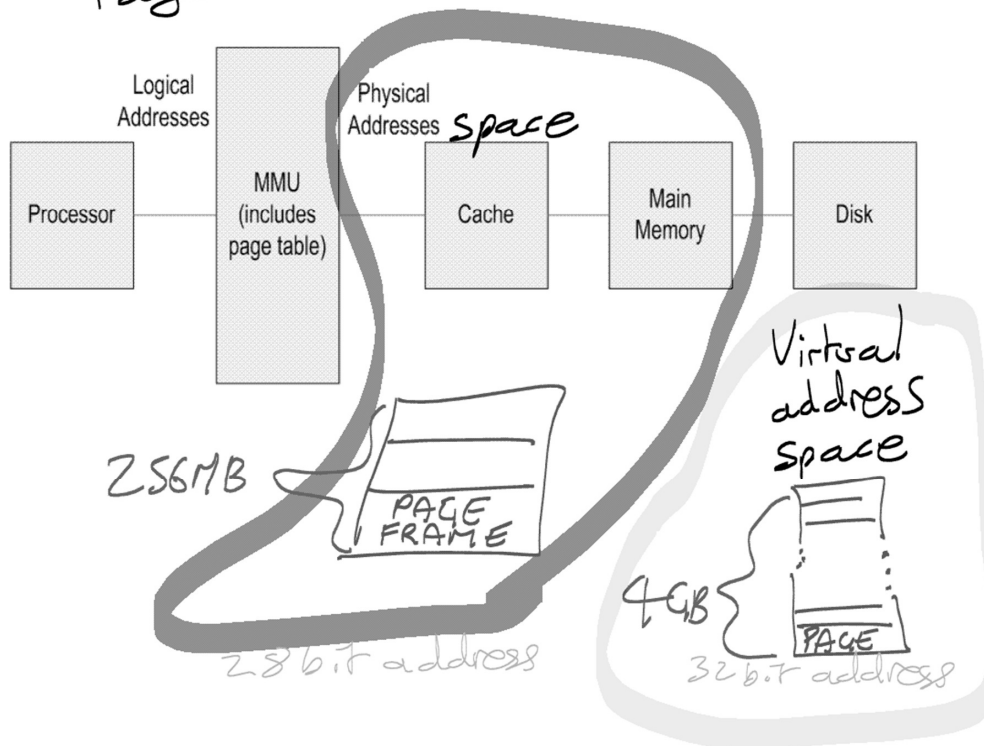
© Copyright Richard Neville 2007

| Virtual page number | Page resident | Page dirty | Page frame number |
|---------------------|---------------|------------|-------------------|
| 0 | 1 | 0 | 4 |
| 1 | 1 | 1 | 7 |
| 2 | 0 | 0 | - |
| 3 | 1 | 0 | 2 |
| 4 | 0 | 0 | - |
| 5 | 1 | 1 | 0 |

26

Summary

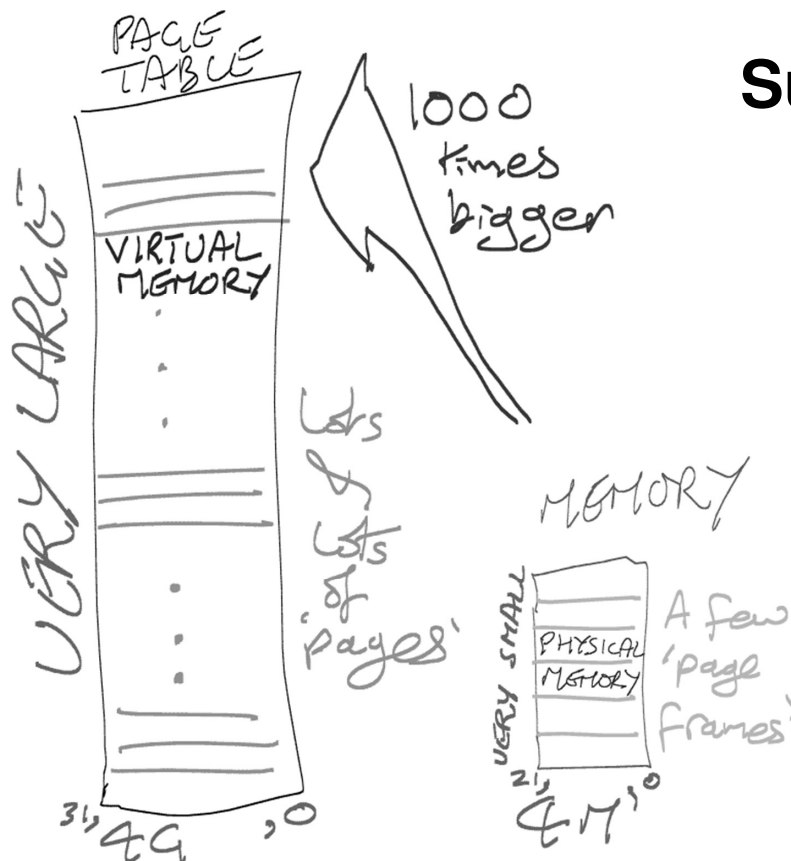
Paged VIRTUAL MEMORY



© Copyright Richard Neville 2007

27

Summary

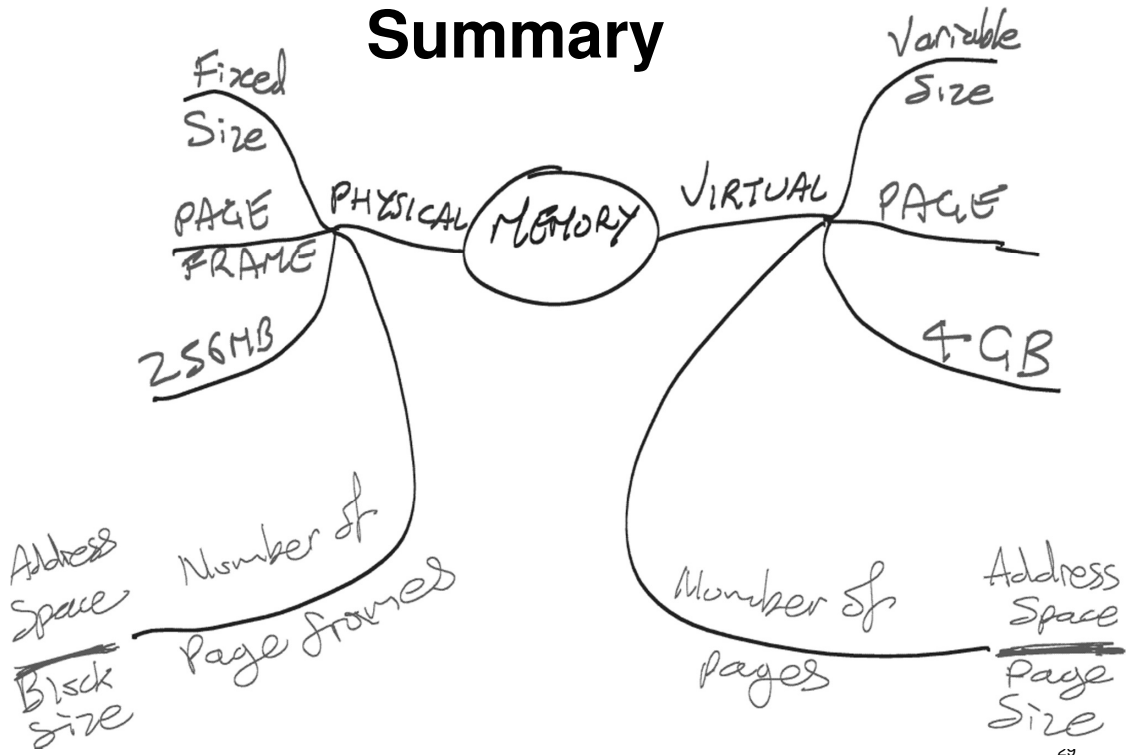


© Copyright Richard Neville 2007

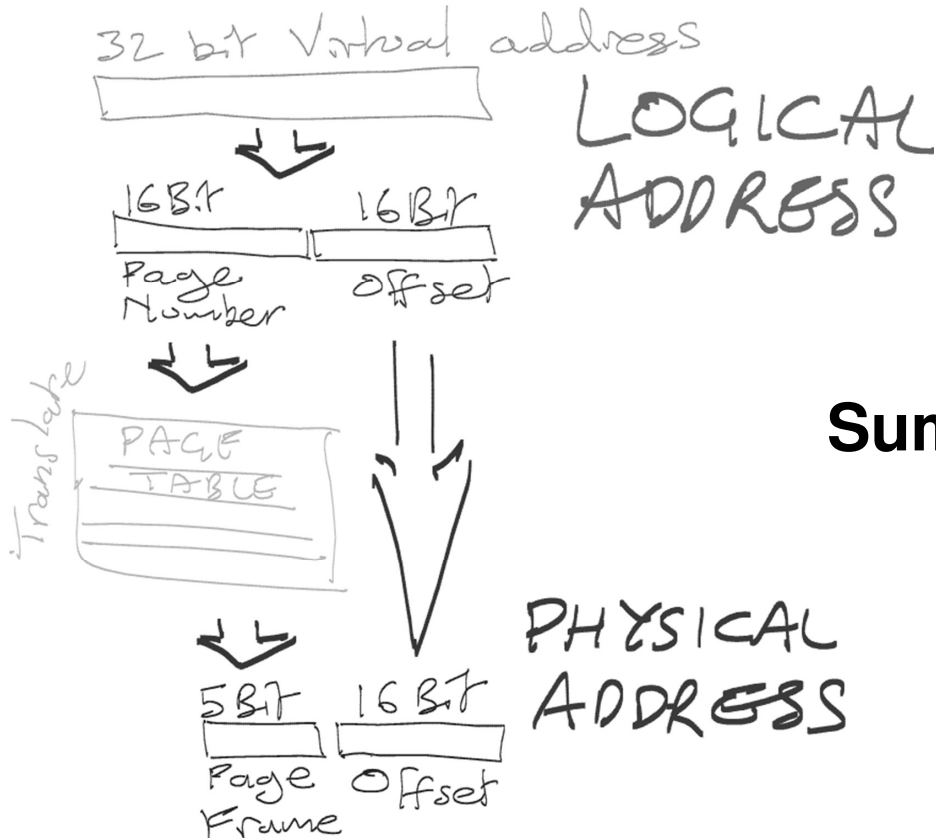
28

Summary

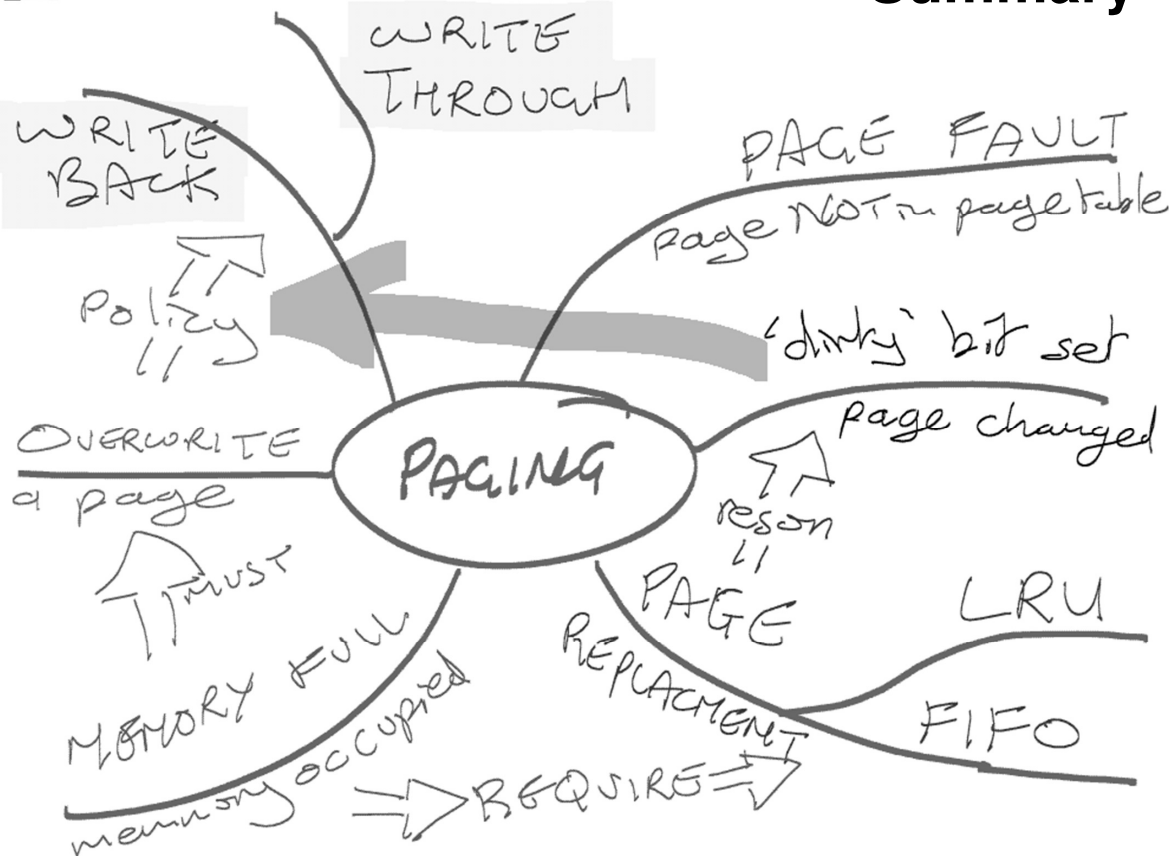
© Copyright Richard Neville 2007



© Copyright Richard Neville 2007



Summary



List of Questions to ask lecturer

- Before the 9a.m. start lecture the lecturer will be half an hour early and you can ask [any and all] questions in that half hour; before the lecture:

- 1.
- 2.
- 3.
- 4.
- 5.



Getting ready for next week

Do next week's Q3's NOW

© Copyright Richard Neville 2007

- Once you have re-read the lecture notes; and listened to the audio recording [while stepping through the PPT] of the lecture again:
- Please have a think about next week's Q3's
 - on the next page
- If you try to answer the Q3's now you will be in a much better position to recall the information.
- Once you have done this, transfer your answers to next weeks "Student [OWN answers] version" at the start of next weeks lecture.
 - YES this implies bringing the last weeks lecture notes to the next lecture ...

33



This week's

Short Exam Questions

Q3

© Copyright Richard Neville 2007

1. Question
What is the difference between a '*partition*' and a '*program*.'

Answer(s):

2. Question
What is a '*fixed partition*':

Answer(s):

3. Question
Differentiate between fragmentation and compaction:
a) Fragmentation: xxx. B) Compaction: xxx.

Answer(s):

34

NOTE: In the exam approximately 2 question are taken from the topics (and program examples) covered in each lecture



1. Question

Differentiate between the write-through strategy and the write-back strategy.

Answer(s):

2. Question

What does the dirty bit indicate. State how it is utilised.

Answer(s):

3. Question

Discuss the differences between 'page fault' and 'page demand.'

Answer(s):

© Copyright Richard Neville 2007

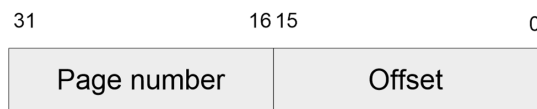
NOTE: In the exam approximately 2 question are taken from the topics (and program examples) covered in each lecture

35



Self-study NOTES: [Information]

No. of bits in Page and Offset



'Page number' No. of bits = $\log_2(nv)$,

'Offset' No. of bits = $\log_2(bs)$

© Copyright Richard Neville 2007

| Page Number | | TOTAL | Offset | |
|---------------------------|-----------------|---------------|----------------|--------------------------|
| No of Pages (<i>nv</i>) | Bits in Page No | | Bits in Offset | Block size (<i>bs</i>) |
| 16 pages | 4 | 12 {11,...,0} | 8 | 256 B |
| 256 pages | 8 | 16 {15,...,0} | 8 | 256 B |
| 16 pages | 4 | 20 {19,...,0} | 16 | 64 KB |
| 256 pages | 8 | 24 {23,...,0} | 16 | 64 KB |
| 64 K pages | 16 | 32 {31,...,0} | 16 | 64 KB |

36



Glossary

- Why build a Glossary for each course unit you undertake?
 - It is imperative that the correct terminology [keywords] are utilised in context in your exam answers; this is so important that the lecturer has added glossaries to each paper copy of your lectures. It is of such importance that in your notes [prior to the start of the glossary] the following advice is given:
 - Each module you undertake uses its own jargon.
 - This can be a problem for new students, whom are trying to comprehend the new domain knowledge attached to a particular new module.
 - One way to get to know the new jargon is to build your own GLOSSARIES for each course module.
 - The glossary on the next few pages is a starting point for this module [unit].
 - Please feel free to add to the glossaries throughout the unit...
 - The glossary is full of potential exam questions of the form "define X" or "briefly explain X."
 - Please heed the advice in the future; even if your lecturers do not supply a glossary build your own as without knowledge of the appropriate terminology [keywords] when expounding your knowledge you will not be viewed as comprehending the details of any theory.



GLOSSARY

Using the on-line resources and any other resources compile a glossary of the terms below [PIPLINES: memory management]:

- Virtual memory →
- Paging →
- Virtual address →
- Paged virtual memory →
- Logical address →
- Physical address →
- MMU →
- Page table →
- Translation →
- Segmented Virtual Memory →
- Virtual address space →
- Physical address space →
- Page frames →
- Page number →



GLOSSARY

© Copyright Richard Neville 2007

- [Page] offset →
- Page fault →
- Page resident →
- Page used →
- Page dirty →
- Page frame number →
- Relative address →
- Page table →
- Demand paging →
- Overwritten →
- LRU →
- FIFO →
- Write-through →
- Write-back →
- Dirty bit →

39



GLOSSARY & AUX. DATA

© Copyright Richard Neville 2007

- Longer pipeline: (effect on the processor's performance? & effect of control hazards).
 - In theory the performance improves by n where n is pipeline length.
 - However, **control hazards** will have a greater effect in a deeper pipeline, not all of which will be mitigated by branch prediction.
There are a number of other comments that could be made about the pipeline in terms of clock speed with longer stages, the ability to fragment instruction execution into so many stages, etc.
- How does a pipelined processor works: (state what performance improvements result).
 - The technique of processing multiple parts of an instruction at the same time.
 - A sequence of instruction units [stages] which performs a task in several steps, like an assembly line in a factory. Performance increases as the units operate in parallel and this increases the process speed.

40

Learning Resources 1

- **Descriptions [Theory] (in text books)**
- Remember the key issues, highlighted in GREEN, are the concepts to look for in any book:
 - Section on virtual memory, paging, virtual address space, MMU, translation, page frame, page fault, demand paging, LRU, FIFO, write-back, & write-through in chapter 9 the operating systems in: Chalk BS, Carter AT, Hind RW (2004) Computer Organisation and Architecture: An introduction 2nd Edition, Palgrave, ISBN 1-4039-0164-3 .
 - Section on virtual memory, paging, memory management, translation, page fault, demand paging – in chapter 7 the operating system support in: Computer Organization and Architecture, Fifth Edition by William Stallings.
 - Section on virtual memory, paging, virtual address space, MMU, translation, page frame, page fault, demand paging, LRU, FIFO, write-back, & write-through – in chapter 6 the operating system machine level in: Structured Computer Organization, 5/E, Andrew S. Tanenbaum, Vrije University, Amsterdam, The Netherlands, ISBN-10: 0131485210, ISBN-13: 9780131485211, Publisher: Prentice. 41

Learning Resources 2

- **Descriptions [Theory] (on the web)**
- **Web resources:**
 - Virtual Memory ; available [on-line] @ <http://courses.cs.vt.edu/csonline/OS/Lessons/VirtualMemory/index.html>
 - **Virtual memory simulator** available [on-line] @ <http://www.ecs.umass.edu/ece/koren/architecture/Vmemory/try.html>
 - MOS Free e-book [Low resolution (Not high quality graphics or printing – but readable)]: Modern Operating Systems (MOS) 2nd Edition Andrew Tanenbaum, Available [on-line] @: http://www.freebookzone.com/fetch.php?bkcls=os_thry&bkidx=35
 - Virtual Memory II, part of course: COMP3231/9201/3891/9283 Operating Systems 2011/S1, Available [on-line] @: <http://cgi.cse.unsw.edu.au/~cs3231/11s1/lectures/lect15.pdf>



Questions

Introduction to Questions:

The set of questions are based on lecture 13.

Answer Sheet will be given later in year and will contain the answers to these questions.

© Copyright Richard Neville 2007

- Remember to find detailed and comprehensive answer you should [also] reference associated text books in the library.
- A reasonable starting place for associated book titles are:
 - 1) This units 'module guide'; given to you in RN's first lecture – or on the web [Blackboard];
 - 2) Those books mentioned in 'Background Reading;'
 - 3) Those books [and web resources] mentioned in Learning Resources.

43



Questions

1. Question:

- Given the following address spaces and associated page sizes; calculate the number of pages that result in the *virtual address space*:
 - a) Page size 32KB and virtual address space 2GB;
 - b) Virtual address space 8GB and page size 256KB.

Answer:

© Copyright Richard Neville 2007

44



1. Answer:

- Given the following address spaces and associated page sizes; calculate the number of pages that result in the *virtual address space*:
 - Page size 32KB and virtual address space 2GB;

Answer(s):



2. Question:

Given the set of specific address size and associated block size below. Calculate the number of page frames in the physical address space :

- Physical address space 1 GB and the block [page] size of 64 KB; and
- Given a block [page] size of 128 KB and a physical address space of 2 GB.

Answer:



2. Answer:

Given the set of specific address size and associated block size below.
Calculate the number of page frames in the physical address space :

- a) Physical address space 1 GB and the block [page] size of 64 KB;

Answer(s):



3. Question:

Given the page table, on the right, and the set of virtual addresses (p, o) below. Answer the following questions for each of the virtual address:

- Will the virtual address generate a page fault?
- If it does, generate a page fault, what must the MMU do? [If not; what does it do then?]
- What is the physical address; if 'no' to a)?

3.1) $(p, o) = \text{Ox } 00020006$; &

3.2) $(p, o) = \text{Ox } 0006000F$.

Given p is a 16-bit number and o is a 16-bit number.

Answer:

Page table

| (p) | (i) |
|-------|-------|
| ... | ... |
| 07 | 00 |
| 06 | XX |
| 05 | 03 |
| 04 | 02 |
| 03 | XX |
| 02 | 01 |
| 01 | XX |
| 00 | XX |

Note1 : the page table only displays the lower 2 Hex digits, of the page number, not all four.

Note 2: XX denotes no page frame number allocated, yet!



3. Answer:

Given the page table, on the right, and the set of virtual addresses (p, o) below. Answer the following questions for each virtual address:

3.1) (p, o) = 0x 00020006

a) Answer(s):

b)

c)

•

3.2) (p, o) = 0x 0006000F

a) Answer(s):

•

b)

•

c)

•

Page table

| (p) | (i) |
|-----|-----|
| ... | ... |
| 07 | 00 |
| 06 | XX |
| 05 | 03 |
| 04 | 02 |
| 03 | XX |
| 02 | 01 |
| 01 | XX |
| 00 | XX |

49



4. Question: exercise in conversion from page to page frame number

- A computer system operates using a paged virtual memory system (no segmentation). The processor address bus is 40 bits wide, the main memory in the system is 1GB and the page size is 128KB.

- What is the size of the virtual address space?
- How many pages are there in the virtual memory?
- How many page frames are there in the main memory?
- The 40-bit virtual address will be split into two bit fields, the *page number* and the *offset*, state how many bits there will be in each field.
- Suppose that part of the page table is given below (entries for 'page frame' number given in hexadecimal representation of a 23-bit binary value). Where it is possible to determine the physical address, state what this will be for the following addresses (given in hexadecimal):

- 0x000000ACEF
- 0x00000B0020
- 0x000004FCDA

- Now, given the Page table opposite; which of the 'page frame Number(s)' are Viable? If not why?

| Virtual Page Number | Page Resident | Page Dirty | Page Frame Number |
|---------------------|---------------|------------|-------------------|
| 0 | 1 | 0 | 3AAC00 |
| 1 | 1 | 1 | 0456BFC |
| 2 | 0 | 0 | - |
| 3 | 1 | 0 | 000200 |
| 4 | 0 | 0 | - |
| 5 | 1 | 1 | 01576F |



4. Answer

© Copyright Richard Neville 2007

a.

Answer(s):

b.

c.

51



4. Answer

© Copyright Richard Neville 2007

d. There are 2^{23} pages to be addresses, this we

Answer(s):

52



4. Answer e.

Questions

i. Firstly split the address into the frame number and offset:

1) 000000ACEF hex

2) Answer(s):

3)

4)

5)

6)

7)

8)



4. Answer e.

Questions

ii. Firstly split the address into the frame number and offset

1) 00000B0020 hex

2) Answer(s):

3)

4)

5)

6)

7)

8)



4. Answer e.

iii. Firstly split the address into the frame number and offset

- 000004FCDA hex

• Answer(s):

•

•

•



4. Answer f.

Answer(s):



5. Answer:

Explain the difference between a virtual memory address and a physical memory address.

Answer(s):



6. A 16 bit [virtual] address is divided into an 8-bit page number and an 8-bit page offset; whereas the associated physical address a 7-bit page frame with the appropriate page offset address sizing. Show, with the aid of a diagram, the table structure necessary to convert this virtual address into a real [physical] address.

Answer:



Learning; comprehension; & introspection

Long [& Short] Exam Questions

Questions

6. A 16 bit [virtual] address is divided into an 8-bit page number and an 8-bit page offset; whereas the associated physical address a 5-bit page frame with the appropriate page offset address sizing. Show, with the aid of a diagram, the table structure necessary to convert this virtual address into a real [physical] address.

Answer:

Answer(s):



Learning; comprehension; & introspection

Long [& Short] Exam Questions

Questions

7. What is a 'page fault'? Describe how a page fault is handled by the Memory Management Unit and the operating system..

Answer:

Answer(s):



Revision Exercises

- Scan read Lecture 13's Questions.
 - Answer Lecture 13's Questions
 - Particularly those questions you had difficulties with when you first tried them.



Background Reading

- [1] Chapter 9 the operating systems in: Chalk BS, Carter AT, Hind RW (2004) Computer Organisation and Architecture: An introduction 2nd Edition, Palgrave, ISBN 1-4039-0164-3 .
- [2] Chapter 7 the operating system support in: Computer Organization and Architecture, Fifth Edition by William Stallings.
- [3] Chapter 6 the operating system machine level in: Structured Computer Organization, 5/E, Andrew S. Tanenbaum, Vrije University, Amsterdam, The Netherlands, ISBN-10: 0131485210, ISBN-13: 9780131485211, Publisher: Prentice.
- [4] Operating system concepts, A. Silberschatz, P.B. Galvin, & G. Gagne, Wiley;
 - Chapter 9 virtual-memory management.
- [5] Modern Operating Systems, Andrew S. Tannenbaum, Various Editions, Prentice Hall;
 - Chapter 4 Memory management.

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.

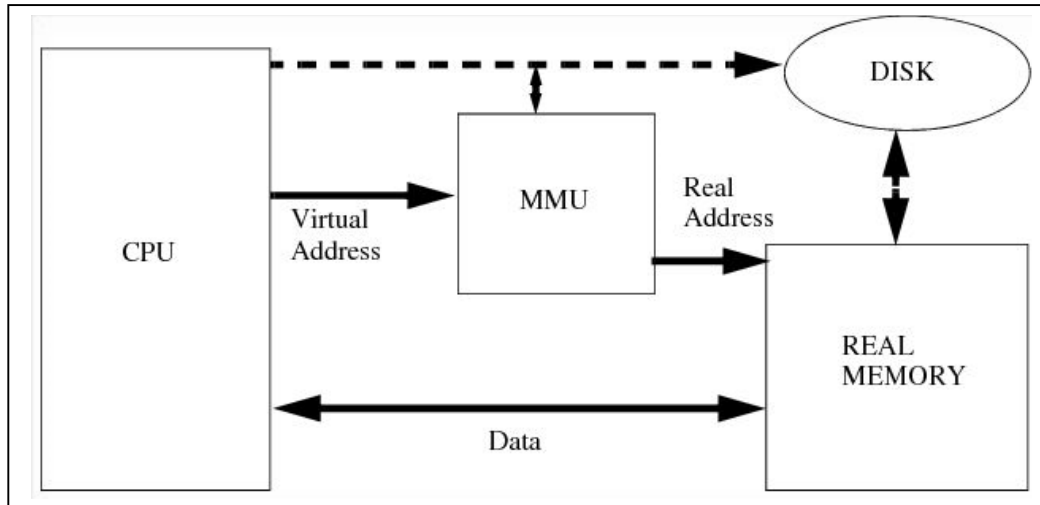


Figure 1 shows the basic structure of a paged virtual memory system.

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.

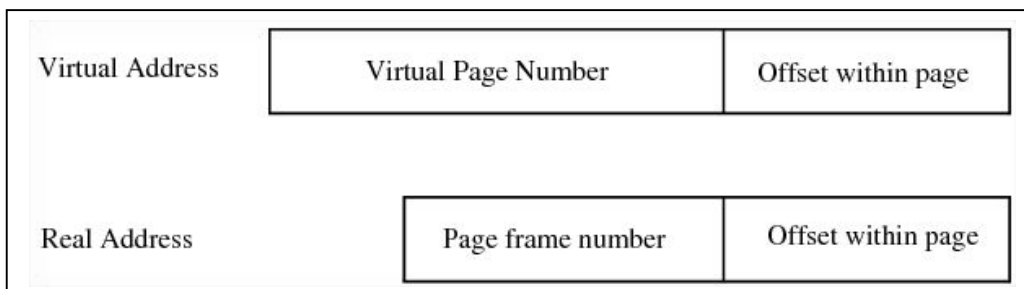


Figure 2 shows this pictorially view of *virtual page numbers* and *page frame numbers*.

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.

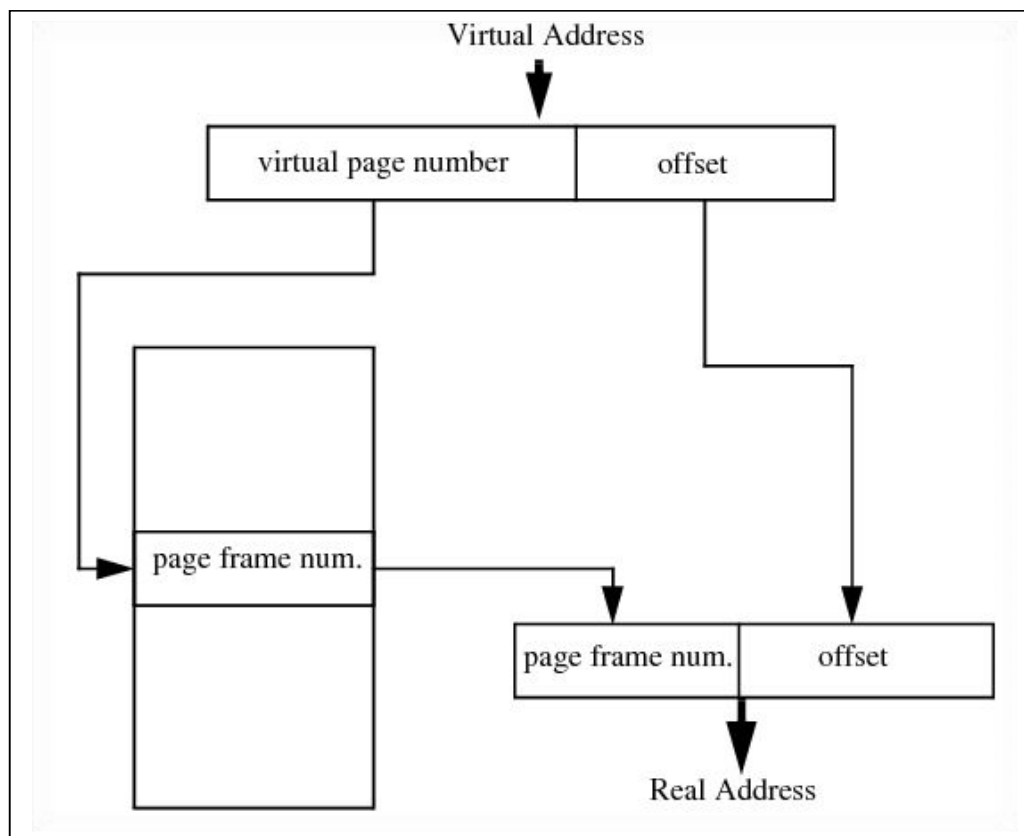


Figure 3 shows more detail of the MMU table and the address translation operation.

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 <input type="text"/> |
| Demonstrating an overall [software engineering] correct approach; of the algorithmic and combined data structure: 2 – 5 . | |
| 2. State how REQUIREMENT development was undertaken. | 1 <input type="text"/> |
| 3. State how [if any] ABSTRACT DESIGN was undertaken. | 1 <input type="text"/> |
| 4. State how IMPLEMENTATION was undertaken. | 1 <input type="text"/> |
| 5. Demonstrating a methodology for TESTING the program; and comment on: interpretation of results . | 1 <input type="text"/> |
| 6. Showing the generation of an appropriate OUTPUT trace file: traceA files [displayed in real-time on command [Prompt] screen] (using LRU). | 1 <input type="text"/> |
| 7. Showing the generation of an appropriate OUTPUT trace file: traceB files [displayed in real-time on command [Prompt] screen] (using LRU). | 2 <input type="text"/> |
| 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 <input type="text"/> |
| TOTAL mark | 10 <input type="text"/> |

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 your 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

ex2 Hint 1

The first stage of the ex2 is to get the basic [sub-set of] classes compiled and then execute and produce an output.

Prior to this it is a good idea to visualise [abstract] the classes that will be working together. One of the ways to do this is to visualise them in an UML class diagram; you will be learning about these later on in your degree – so don't worry – but if you want to find out more about UML class diagrams undertake some research; figure 1: [basic] UML class diagram of [sub-set of] classes.

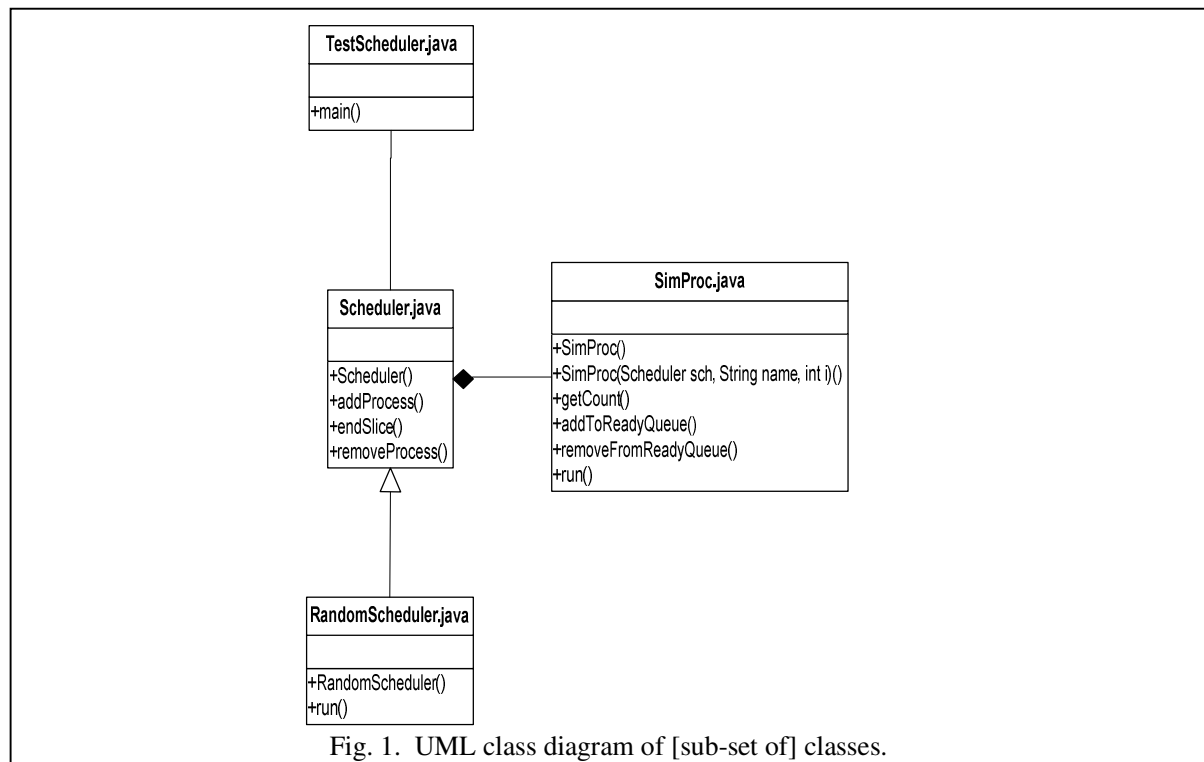
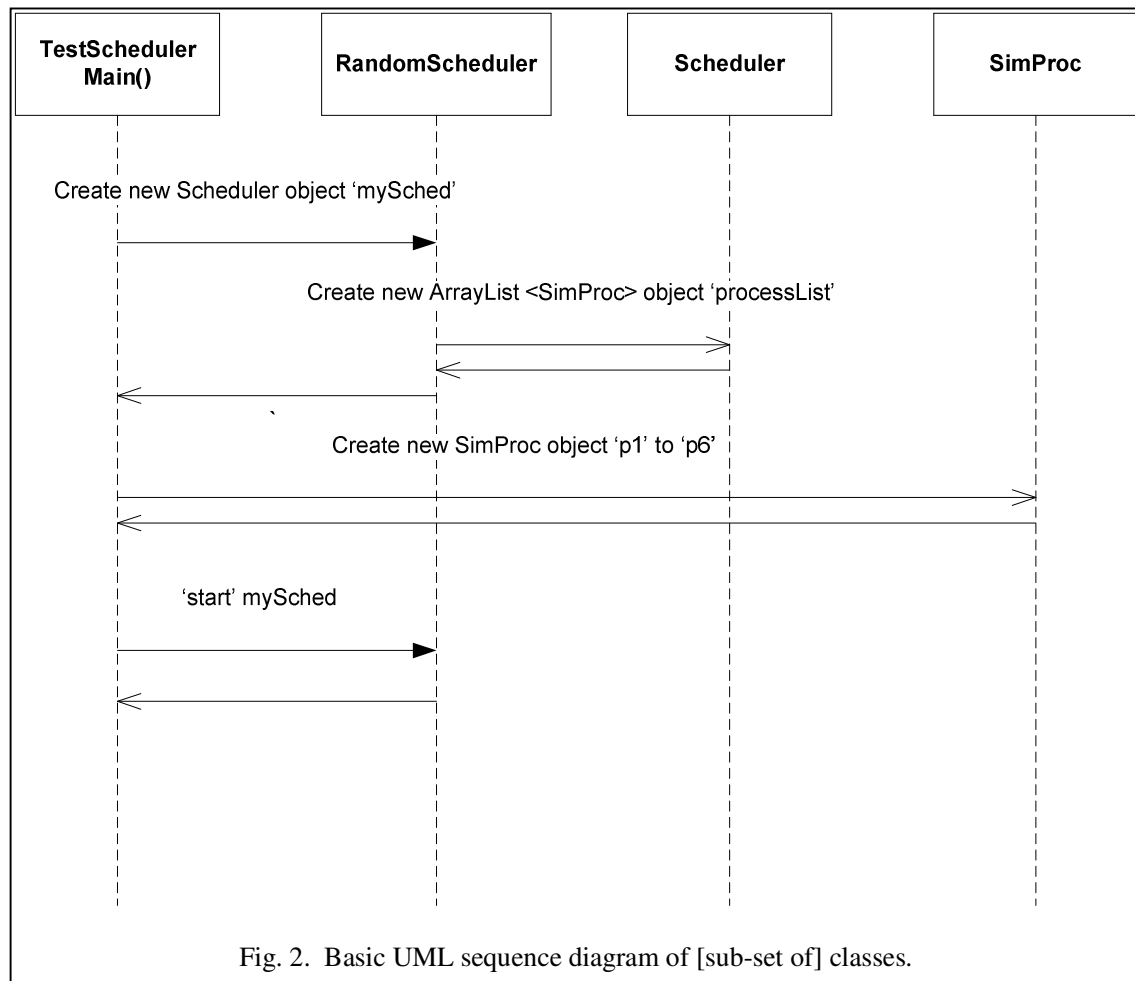


Fig. 1. UML class diagram of [sub-set of] classes.

It is also an idea to visualise the sequence of instructions; or the method call sequence; sometimes viewed as “the interactions between objects in the sequential order that those interactions occur [1];” this can be done utilising the UML sequence diagram. In the case of the sub-set of classes: `TestScheduler.java`, `Scheduler.java`, `RandomScheduler.java`, `SimProc.java` the basic outline of the sequence diagram is depicted in Figure 2; **note** this sequence diagram does NOT stick to the normal UML convention for labelling message arrows – it labels the arrows with the object method call – as specified in the java code.



The next step is the compilation and execution of: `TestScheduler.java`, `Scheduler.java`, `RandomScheduler.java`, `SimProc.java`. One you have compiled the four files run them and the resultant output you should acquire is depicted in figure 3 [Remember as it is “Random” order the output threads will be in a random order (of the Threads) – hence you will not obtain exactly the same output in figure 3 – but if you inspect the sequence you will indeed see they are in random order].

```
This is Francis - Collatz of 1457712396 is 165
This is Alan - Collatz of 345300012 is 145
This is Francis - Collatz of 2054253694 is -1
This is Brian - Collatz of 1902024452 is 243
This is Edward - Collatz of 1741402412 is 137
This is Brian - Collatz of 2044483757 is 244
This is Francis - Collatz of 424086582 is -1
This is Charles - Collatz of 244589539 is 85
This is Alan - Collatz of 1967502833 is 236
This is Brian - Collatz of 784079238 is 115
This is Francis - Collatz of 115955971 is 239
This is Edward - Collatz of 1800016525 is 220
This is Charles - Collatz of 11682451 is 184
This is Alan - Collatz of 1064384834 is -1
This is Dave - Collatz of 832032679 is -1
This is Francis - Collatz of 1033704605 is -1
This is Edward - Collatz of 1125642362 is 219
This is Brian - Collatz of 902174553 is -1
This is Alan - Collatz of 2222371 is 117
This is Francis - Collatz of 351718320 is 150
This is Edward - Collatz of 1041352256 is 255
This is Charles - Collatz of 293308302 is 204
This is Brian - Collatz of 1230156273 is -1
This is Alan - Collatz of 2052532180 is 176
This is Francis - Collatz of 1458526209 is 118
This is Charles - Collatz of 382063338 is 70
This is Brian - Collatz of 1613859455 is -1
This is Dave - Collatz of 184521508 is 250
This is Brian - Collatz of 2021315824 is 181
This is Dave - Collatz of 1533157800 is 98
This is Dave - Collatz of 2054627414 is -1
This is Dave - Collatz of 1967699861 is 249
This is Dave - Collatz of 1257881609 is -1
This is Dave - Collatz of 2090545418 is -1
This is Dave - Collatz of 1560187619 is 159
Figure 3, Output of TestScheduler.java, Scheduler.java,
RandomScheduler.java, SimProc.java files...
```

Reading “the Code”

Finally, as will all exercises, we should read through the code. Hence, you should start with `TestScheduler.java` and work out the basic algorithmics it is implementing. Then take a look at `Scheduler.java`, `RandomScheduler.java`, and `SimProc.java`.

One of the questions you could ask yourself is: “Can I break the code up into functional blocks?” Where each block performs (or addresses) separate steps (or issues). If we look at the `RandomScheduler.java` code we should be able to do this.

NOTE: If you are seriously testing your comprehension of the code DO NOT read the next page till you have (yourself) worked out what `RandomScheduler.java` is doing algorithmically.

May be the best advice for sequentially separating concerns in a sequential program is to inspect the code for set of lines that are sequentially operating on the same object [or variable].

First parse of RandomScheduler.java separates the run() into three separate blocks, reference figure 4.

```
public class RandomScheduler extends Scheduler {
    private Random randGen ;

    public RandomScheduler() {
        randGen = new Random() ;
    }

    public synchronized void run() {
        while (true) {
            int noProcs = processList.size() ;
            if (noProcs == 0) System.exit(0) ;
            int next = Math.abs(randGen.nextInt()) % noProcs ;
            SimProc nextRunner = processList.get(next) ;
            synchronized(nextRunner) { nextRunner.notify() ;}
            try { wait() ;}
            catch (Exception e) {
                System.out.println("Unexpected interrupt in run " + e) ;
            }
        }
    }
}

//*****
public class RandomScheduler extends Scheduler {
    private Random randGen ;

    public RandomScheduler() {
        randGen = new Random() ;
    }

    public synchronized void run() {
        while (true) {

            //#BLOCK1:
            int noProcs = processList.size() ;
            if (noProcs == 0) System.exit(0) ;

            //#BLOCK2:
            int next = Math.abs(randGen.nextInt()) % noProcs ;
            SimProc nextRunner = processList.get(next) ;
            synchronized(nextRunner) { nextRunner.notify() ;}

            //#BLOCK3:
            try { wait() ;}
            catch (Exception e) {
                System.out.println("Unexpected interrupt in run " + e) ;
            }
        }
    }
}
```

Figure 4, Segmentation of the RandomScheduler.java class into 3 BLOCKs...

#BLOCK1: the first block retrieves the noProcs variable, which retrieves the size of threads queue (or processList). Next, the if condition, in the next instruction, checks if all the threads (processes) have terminated. Note: the basic sequence is that threads run until they have finished undertaking the task they were designated. When they end they are removed from the queue and the size of threads queue (or processList) is decremented. When all threads have terminated the size == 0, hence noProcs == 0, then (noProcs == 0) so the run() is exited; utilising the System.exit(0) method.

#BLOCK2: Delineated by the use of variables & objects: `next` & `nextRunner`. The three lines basically:

- (1) get a random number `next`;
- (2) get the `next` thread from the queue, using `next` as the index to the thread in the queue to actually retrieve a random item from the queue [`next`]. Given that `next` is a random number;
- (3) Finally, notify the next thread in the queue (or the `nextRunner`); note a lock is placed on this object, using the `synchronized(nextRunner)` statement. Actually, `notify()` wakes up a single thread which is waiting on the object's lock (object that was locked was `nextRunner`).

The basic sequence of `notify()` [in block 2] and then `wait()` [in block 3]. First the running thread calls the `notify()` method of the `nextRunner` Java object. This "wakes up" `nextRunner` thread of the threads waiting on that object. The `wait()` method will be discussed in BLOCK 3.

#BLOCK3: Puts the current thread (or process) into a `wait()` state, to enable other threads to run. Hence, BLOCK 3 calls the `wait()` method of any Java object, which suspends the current thread. The thread [queue] is said to be "waiting on" the given object. This is scoped {} in a try & catch block to deal with an unexpected interrupt.

NOTE [on Thread States] quote:

"Tasks are executed in threads. Threads can be in one of five states: New, Ready, Running, Blocked, or Finished.

When a thread is newly created, it enters the *New state*. A ready thread is started by calling its `start()` method, it [then] enters the *Ready state*. A ready thread is runnable but may not be running yet. The operating system has to allocate CPU time to it.

When a ready thread begins executing, it enters the *Running state*. A running thread may enter the *Ready* state if [the] CPU time [it was allocated] expires [or finishes] or its `yield()` method is called.

A thread can enter the *Blocked state* (i.e. become inactive) for several reasons. It may have invoked the `join()`, `sleep()`, `wait()`, or `lock()` method, or some other thread may have invoked these methods [on the running thread]. It may be waiting for an I/O operation to finish. A blocked thread may be reactivated when the action inactivating it is reversed. For example, if a thread has been put to sleep and the sleep time has expired, the thread is reactivated and enters the *Ready* state. Finally, a thread is finished if it completes execution of its `run()` method," [3].

Hence, in `RandomScheduler.java` class when instruction `try { wait() ; }` is executed the thread "enter the *Blocked state* (i.e. become inactive)," [3].

IMPORTANT: hence for the other two schedulers there is probably the same number of blocks. It is BLOCK 2 in both `RoundRobinScheduler.java` and `PriorityScheduler.java` that you have to implement.

Best advice is copy `RandomScheduler.java` to `RoundRobinScheduler.java` and `PriorityScheduler.java`, then modify them to perform sequential round robin scheduling and priority based scheduling.

Appendix

Definitions of terminology utilised in ex2.

Pre-emptive Algorithms

“Pre-emptive algorithms are the scheduling algorithms that take a process from the queue and assign the algorithm a set time slot, which it is allowed to run for. If the process does not finish running by then it is put back into the queue at the end and another process is selected to be run. However if during that time it does finish then the process is removed from the queue. The way it determines the algorithm to run next is via the scheduling algorithm that is in place. The animation system animates two types of pre-emptive algorithms, Round Robin Scheduler and Priority Scheduler [2].”

Round Robin

“This scheduling algorithm takes a process from the head of the queue. In other words, it takes the first process that entered the queue. It then will run the process for a specified period of time. This timeslot can be specified by the user when the applet is run. If the process is not finished after its allocated time then the process is put back in the queue at the end and the scheduler takes the next one at the head of the queue and runs it. It continues to do this till the processes have finished running [2].”

Priority Scheduler

“This algorithm has a further property defined with the processes. This property is the priority of the algorithm. Each process can have a priority defined with 1 being the highest. The algorithm will look through the processes in the queue and find the one with the highest priority and run that first. However when the process has been ran for the allocated timeslot, the algorithm deducts the priority by one and inserts it back in the queue. This gives the other algorithms an opportunity to run too [2].”

Reference

- [1] UML basics: The sequence diagram, available on line @ <http://www.ibm.com/developerworks/rational/library/3101.html>, last accessed 6/10/2011.
- [2] R. Begum, (May, 2010) Algorithm Animation, Final year Project, The University of Manchester School of Computer Science.
- [3] Y. D. Liang, Introduction to Java Programming, ISBN 10: 0132130807 / 0-13-213080-7, ISBN 13: 9780132130806, Publisher: Prentice Hall, Publication Date: 2010, Binding: Softcover, also [some information] is available on-line @ <http://cs.armstrong.edu/liang/intro6e/> , <http://cs.armstrong.edu/liang/intro7e/>, http://books.google.com/books/about/Introduction_to_JAVA_programming.html?id=fI6dl1flmk8C , and <http://www.filestube.com/i/introduction+to+java+programming+liang> , last accesses 13-10-2011.

