OPS Memory Management

Handout Introduction

These notes were originally written using (Tanenbaum, 1992) but later editions (Tanenbaum, 2001; 2008) contain the same information.

Introduction

One of the main tasks of an operating system is to manage the computers memory. This includes many responsibilities, including

- Being aware of what parts of the memory are in use and which parts are not.
- Allocating memory to processes when they request it and de-allocating memory when a process releases its memory.
- Moving data from memory to disc, when the physical capacity becomes full, and vice versa.

In this handout we consider some ways in which these functions are achieved.

Monoprogramming

If we only allow a single process in memory at a time we can make life simple for ourselves. In addition, if we only allow one process to run at any one time then we can make life very simple.
That is, the processor does not permit multi-programming and only one process is allowed to be in memory at a time. Using this model we do not have to worry about swapping processes out to disc when we run out of memory. Nor do we have to worry about keeping processes separate in memory.
All we have to do is load a process into memory, execute it and then unload it before loading the next process.

However, even this simple scheme has its problems.

- We have not yet considered the data that the program will operate upon.
- We are also assuming that a process is self contained in that it has everything within it that allows it to function. This assumes that it has a driver for each device it needs to communicate with. This is both wasteful and unrealistic. We are also forgetting about the operating system routines. The OS can be considered another process and so we have two processes running which means we have left behind our ideal where we can consider that the memory is only being used by one process.

But, even if a monoprogramming model did not have these memory problems we would still be faced with other problems.
In this day and age monoprograming is unacceptable as multi-programming is not only expected by the users of a computer but it also allows us to make more effective use of the CPU. For example, we can allow a process to use the CPU whilst another process carries out I/O or we can allow two people to run interactive jobs and both receive reasonable response times.

We could allow only a single process in memory at one instance in time and still allow multi-programming. This means that a process, when in a running state, is loaded in memory. When a context switch occurs the process is copied from memory to disc and then another process is loaded into memory.
This method allows us to have a relatively simple memory module in the operating system but still allows multi-programming.

The drawback with the method is the amount of time a context switch takes. If we assume that a quantum is 100ms and a context switch takes 200ms then the CPU spends a disproportionate amount of time switching processes. We could increase the amount of time for a quantum but then the interactive users will receive poor response times as processes will have to wait longer to run.

Modelling Multiprogramming

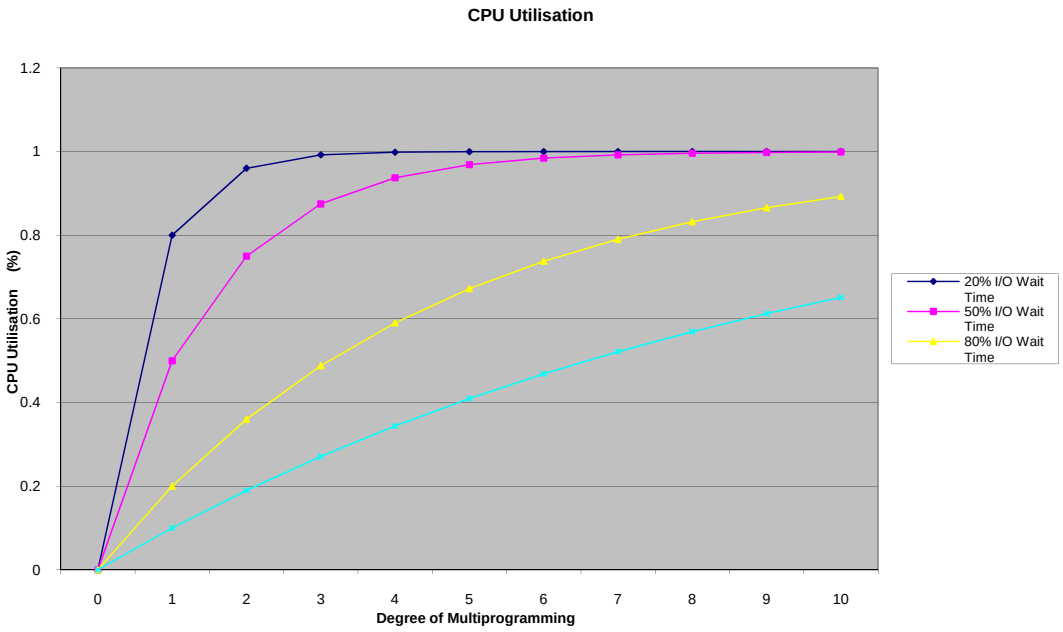
We assume (and have stated) that multiprogramming can improve the utilisation of the CPU. And, intuitively, this is the case. If we have five processes that use the processor twenty percent of the time (spending eighty percent doing I/O) then we should be able to achieve one hundred percent CPU utilisation. Of course, in reality, this will not happen as there may be times when all five processes are waiting for I/O. However, it seems reasonable that we will achieve better than twenty percent utilisation that we would achieve with monoprogramming.

But, can we model this?

We can build a model from a probabilistic viewpoint.
Assume that that a process spends p percent of its time waiting for I/O. With n processes in memory the probability that all n processes are waiting for I/O (meaning the CPU is idle) is p^n.
The CPU utilisation is then given by

CPU Utlisation = $1 - p^n$

The following graph shows this formula being used (the spreadsheet that produced this graph is available from the web site for this course).



You can see that with an I/O wait time of 20%, almost 100% CPU utilisation can be achieved with four processes.
If the I/O wait time is 90% then with ten processes, we only achieve just above 60% utilisation.
The important point is that, as we introduce more processes the CPU utilisation rises.

The model is a little contrived as it assumes that all the processes are independent in that processes could be running at the same time. This (on a single processor machine) is obviously not possible.

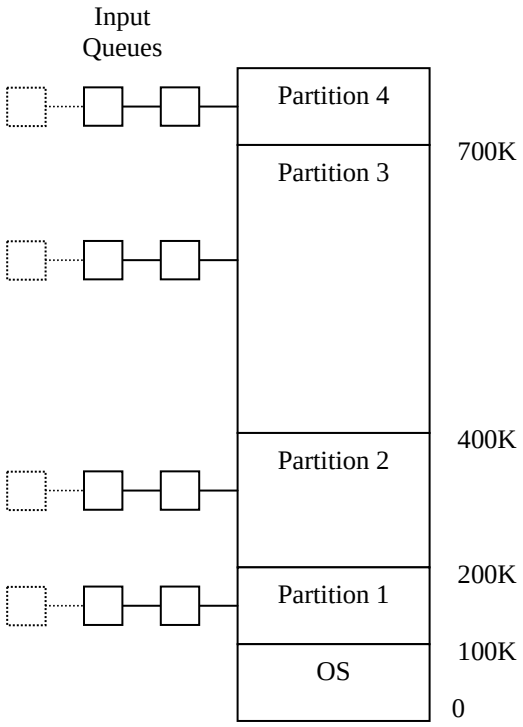
More complex models could be built using queuing theory but we can still use this simplistic model to make approximate predictions.

Assume a computer with one megabyte of memory. The operating system takes up 200K, leaving room for four 200K processes. If we have an I/O wait time of 80% then we will achieve just under 60% CPU utilisation.
If we add another megabyte, it allows us to run another five processes (nine in all). We can now achieve about 86% CPU utilisation. You might now consider adding another megabyte of memory, allowing fourteen processes to run. If we extend the above graph, we will find that the CPU utilisation will increase to about 96%.
Adding the second megabyte allowed us to go from 59% to 86%. The third megabyte only took us from 86% to 96%. It is a commercial decision if the expense of the third megabyte is worth it.

Mulitprogramming with Fixed Partitions

If we accept that mulitiprogramming is a good idea, we next need to decide how to organise the available memory in order to make effective use of the resource.

One method is to divide the memory into fixed sized partitions. These partitions can be of different sizes but once a partition has taken on a certain size then it remains at that size. There is no provision for changing its size.
The IBM OS/360 was set up in this way. The computer operator defined the sizes of the partitions in the morning (or when the machine was booted) and these partitions remained in effect until the computer was reloaded. This was called MFT (***M***ultiprogramming with ***F***ixed number of ***T***asks – or OS/MFT)



The diagram above shows how this scheme might work. The memory is divided into four partitions (we'll ignore the operating system). When a job arrives it is placed in the input queue for the smallest partition that will accommodate it.
There are a few drawbacks to this scheme.

1. As the partition sizes are fixed, any space not used by a particular job is lost.
2. It may not be easy to state how big a partition a particular job needs.
3. If a job is placed in (say) queue three it may be prevented from running by other jobs waiting (and using) that partition.

To cater for the last problem we could have a single input queue where all jobs are held. When a partition becomes free we search the queue looking for the first job that fits into the partition. An alternative search strategy is to search the entire input queue looking for the largest job that fits into the partition. This has the advantage that we do not waste a large partition on a small job but has the disadvantage that smaller jobs are discriminated against. Smaller jobs are typically interactive jobs which we normally want to service first.
To ensure small jobs do get run we could have at least one small partition or ensure that small jobs only get skipped a certain number of times.

Using fixed partitions is easy to understand and implement, although there are a number of drawbacks which we have outlined above.

Relocation and Protection

As soon as we introduce multiprogramming we have two problems that we need to address.

Relocation : When a program is run it does not know in advance what location it will be loaded at. Therefore, the program cannot simply generate static addresses (e.g. from jump instructions). Instead, they must be made relative to where the program has been loaded.

Protection : Once you can have two programs in memory at the same time there is a danger that one program can write to the address space of another program. This is obviously dangerous and should be avoided.

In order to cater for relocation we could make the loader modify all the relevant addresses as the binary file is loaded. The OS/360 worked in this way but the scheme suffers from the following problems

- The program cannot be moved, after it has been loaded without going through the same process.
- Using this scheme does not help the protection problem as the program can still generate *illegal* addresses (maybe by using absolute addressing).
- The program needs to have some sort of map that tells the loader which addresses need to be modified.

A solution, which solves both the relocation and protection problem is to equip the machine with two registers called the ***base*** and ***limit*** registers.
The base register stores the start address of the partition and the limit register holds the length of the partition. Any address that is generated by the program has the base register added to it. In addition, all addresses are checked to ensure they are within the range of the partition.

An additional benefit of this scheme is that if a program is moved within memory, only its base register needs to be amended. This is obviously a lot quicker than having to modify every address reference within the program.

The IBM PC uses a scheme similar to this, although it does not have a limit register.

Swapping

Using fixed partitions is a simple method but it becomes ineffective when we have more processes than we can fit into memory at one time. For example, in a timesharing situation where many people want to access the computer, more processes will need to be run than can be fitted into memory at the same time.
The answer is to hold some of the processes on disc and ***swap*** processes between disc and main memory as necessary.

In this section we look at how we can manage this swapping procedure.

Multiprogramming with Variable Partitions

Just because we are swapping processes between memory and disc does not stop us using fixed partition sizes. However, the reason we are having to swap processes out to disc is because memory is a scare resource and, as we have discussed, fixed partitions can be wasteful of memory. Therefore it would be a good idea to look for an alternative that makes better use of the scarce resource.

It seems an obvious development to move towards ***variable partition*** sizes. That is partitions that can change size as the need arises. Variable partitions can be summed up as follows

- The number of partition varies.
- The sizes of the partitions varies.
- The starting addresses of the partitions varies.

These make for a much more effective memory management system but it makes the process of maintaining the memory much more difficult.

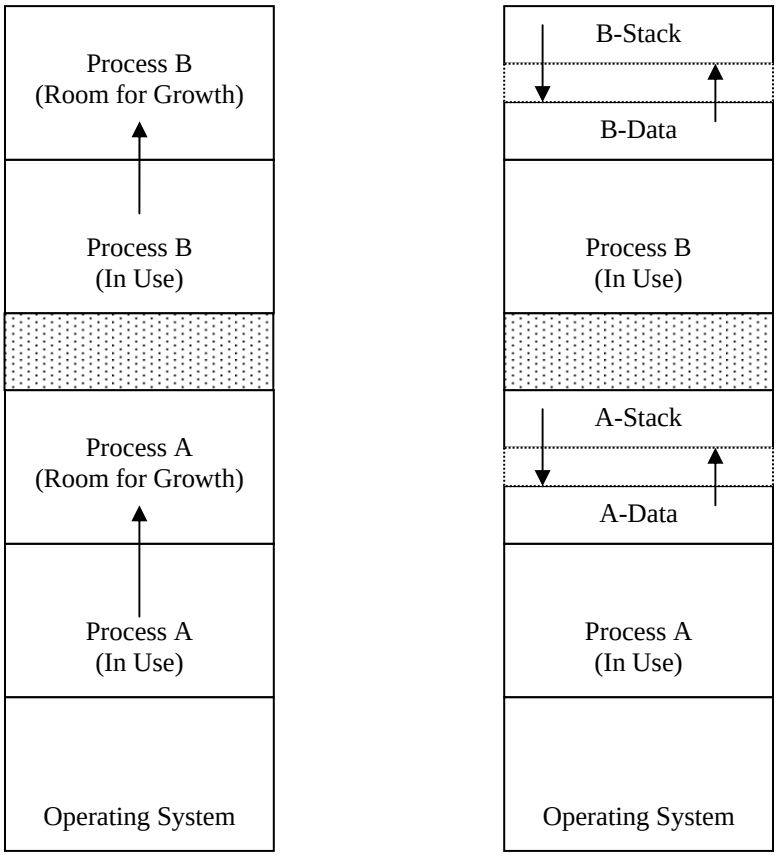
For example, as memory is allocated and deallocated *holes* will appear in the memory; it will become fragmented. Eventually, there will be *holes* that are too small to have a process allocated to it. We could simply *shuffle* all the memory being used downwards (called ***memory compaction***), thus closing up all the holes. But this could take a long time and, for this reason it is not usually done.

Another problem is if processes are allowed to grow in size once they are running. That is, they are allowed to dynamically request more memory (e.g. the new statement in C++). What happens if a process requests extra memory such that increasing its partition size is impossible without it having to overwrite another partitions memory? We obviously cannot do that so do we wait until memory is available so that the process is able to grow into it, do we terminate the process or do we move the process to a hole in memory that is large enough to accommodate the growing process?
The only realistic option is the last one; although it is obviously wasteful to have to copy a process from one part of memory to another - maybe by swapping it out first.

None of the solutions are ideal so it would seem a good idea to allocate more memory than is initially required. This means that a process has somewhere to grow before it runs out of memory (see left hand figure below).

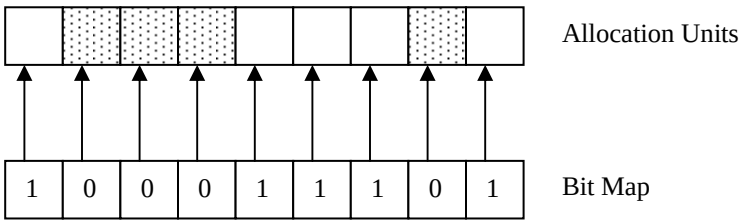
Most processes will be able to have two growing data segments, data created on the stack and data created on the heap. Instead of having the two data segments grow upwards in memory a neat arrangement has one data area growing downwards and the other data segment growing upwards.
This means that a data area is not restricted to just its own space and if a process creates more memory in the heap then it is able to use space that may have been allocated to the stack (see the right hand figure below).

In the next sections we are going to look at three ways in which the operating system can keep track of the memory usage. That is, which memory is free and which memory is being used.



Memory Usage with Bit Maps

Under this scheme the memory is divided into allocation units and each allocation unit has a corresponding bit in a bit map. If the bit is zero, the memory is free. If the bit in the bit map is one, then the memory is currently being used.
This scheme can be shown as follows.

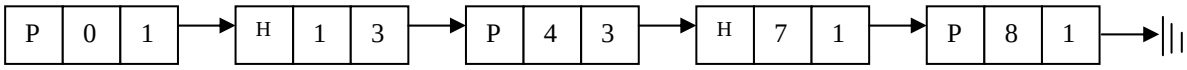


The main decision with this scheme is the size of the allocation unit. The smaller the allocation unit, the larger the bit map has to be. But, if we choose a larger allocation unit, we could waste memory as we may not use all the space allocated in each allocation unit.

The other problem with a bit map memory scheme is when we need to allocate memory to a process. Assume the allocation size is 4 bytes. If a process requests 256 bytes of memory, we must search the bit map for 64 consecutive zeroes. This is a slow operation and for this reason bit maps are not often used.

Memory Usage with Linked Lists

Free and allocated memory can be represented as a linked list. The memory shown above as a bit map can be represented as a linked list as follows.



Each entry in the list holds the following data

- P or H : for Process or Hole
- Starting segment address
- The length of the memory segment
- The next pointer is not shown but assumed to be present

In the list above, processes follow holes and vice versa (with the exception of the start and the end of the list). But, it does not have to be this way. It is possible that two processes can be next to each other and we need to keep them as separate elements in the list so that if one process ends we only return the memory for that process.
Consecutive holes, on the other hand, can always be merged into a single list entry.

This leads to the following observations when a process terminates and returns its memory.

A terminating process can have four combinations of neighbours (we'll ignore the start and the end of the list to simplify the discussion).

If X is the terminating process the four combinations are

	Before X terminates			**After X terminates**		
1		X				
2		X				
3		X				
4		X				

- In the first option we simply have to replace the P by an H, other than that the list remains the same.
- In the second option we merge two list entries into one and make the list one entry shorter.
- Option three is effectively the same as option 2.
- For the last option we merge three entries into one and the list becomes two entries shorter.

In order to implement this scheme it is normally better to have a doubly linked list so that we have access to the previous entry.

When we need to allocate memory, storing the list in segment address order allows us to implement various strategies.

First Fit : This algorithm searches along the list looking for the first segment that is large enough to accommodate the process. The segment is then split into a hole and a process. This method is fast as the first available hole that is large enough to accommodate the process is used.

Best Fit : Best fit searches the entire list and uses the smallest hole that is large enough to accommodate the process. The idea is that it is better not to split up a larger hole that might be needed later.

Best fit is slower than first fit as it must search the entire list every time. It has also be shown that best fit performs worse than first fit as it tends to leave lots of small gaps.

Worst Fit : As best fit leaves many small, useless holes it might be a good idea to always use the largest hole available. The idea is that splitting a large hole into two will leave a large enough hole to be useful.
It has been shown that this algorithm is not very good either.

These three algorithms can all be speeded up if we maintain two lists; one for processes and one for holes. This allows the allocation of memory to a process to be speeded up as we only have to search the hole list. The downside is that list maintenance is complicated. If we allocate a hole to a process we have to move the list entry from one list to another.

However, maintaining two lists allows us to introduce another optimisation. If we hold the hole list in size order (rather than segment address order) we can make the best fit algorithm stop as soon as it finds a hole that is large enough. In fact, first fit and best fit effectively become the same algorithm.

The Quick Fit algorithm takes a different approach to those we have considered so far. Separate lists are maintained for some of the common memory sizes that are requested. For example, we could have a list for holes of 4K, a list for holes of size 8K etc. One list can be kept for large holes or holes which do not fit into any of the other lists.
Quick fit allows a hole of the right size to be found very quickly, but it suffers in that there is even more list maintenance.

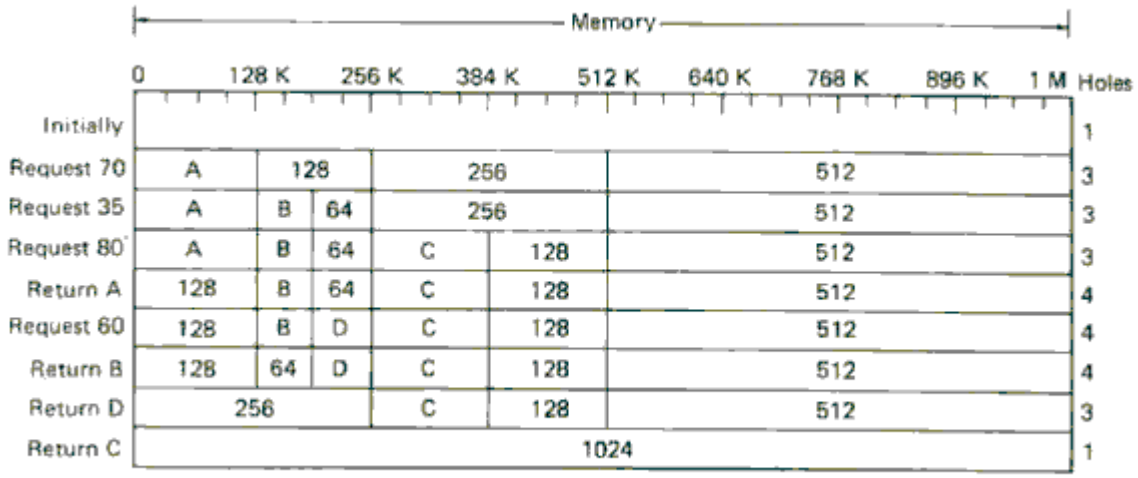
Memory Usage with the Buddy System

If we keep a list of holes sorted by their size, we can make allocation to processes very fast as we only need to search down the list until we find a hole that is big enough.
The problem is that when a process ends the maintenance of the lists is complicated. In particular, merging adjacent holes is difficult as the entire list has to be searched in order to find its neighbours.

The ***Buddy System*** (Knuth, 1973; Knowlton, 1965) is a memory allocation technique that works on the basis of using binary numbers as these are fast for computers to manipulate.

This is how the buddy system works.

Lists are maintained which store lists of free memory blocks of sizes 1, 2, 4, 8,…, *n*, where *n* is the size of the memory (in bytes). This means that for a one megabyte memory we require 21 lists.
If we assume we have one megabyte of memory and it is all unused then there will be one entry in the 1M list; and all other lists will be empty.



Now assume that a 112K process (process A) needs to be swapped into memory. As lists are only held as powers of two we have to allocate the next highest memory that is a power of two; in this case 128K. The 128K list is currently empty. In fact, every list is empty except for the 1M list.

Therefore, we split the 1M block into two 512K blocks. One of the 512K blocks is then split into 256K blocks and one of the 256K blocks is split into two 128K blocks; one of which is allocated to the process. This situation is shown in the diagram above. At this stage we have three lists with one entry (128K, 256K and 512K) and the 1M list is empty (as are all the other lists).

The reason the scheme is known as the buddy system is because each time a block is split it is called a ***buddy***.
Next, a process (process B) requiring 35K might be swapped in. This will require a 64K block. There are no entries in the 64K list so the next size list is considered (128K). This has an entry so two buddies are created and the process is allocated to one of those blocks.
If we now request an 80K process (process C), this will have to occupy a 128K block, which will come from the 256K list.

What happens if process A ends and releases its memory? In fact, the block of memory will simply be added to the 128K list.

If another process now requests 60K of memory it will find an entry in the 64K list, so can be allocated there.

Now process B terminates and releases its memory. This will simply place its block in the 64K list.
If process D terminates we can start merging blocks. This is a fast process as we only have to check adjacent lists and check for adjoining addresses.
Finally, process C terminates and we can merge all the way back to a single list entry in the 1M list.

The reason the buddy system is fast is because when a block size of 2^k bytes is returned only the 2^k list has to be searched to see if a merge is possible.

The problem with the buddy system is that it is inefficient in terms of memory usage. All memory requests have to be rounded up to a power of two. We saw above how an 80K process has to be allocated to a 128K memory block. The extra 40K is wasted.
This type of wastage is known as ***internal fragmentation***. As the wasted memory is internal to the allocated segments. This is the opposite to ***external fragmentation*** where the wasted memory appears between allocated segments.

For the interested student (Peterson, 1977) and (Kaufman, 1984) have modified the buddy system to get around some of these problems.

Virtual Memory

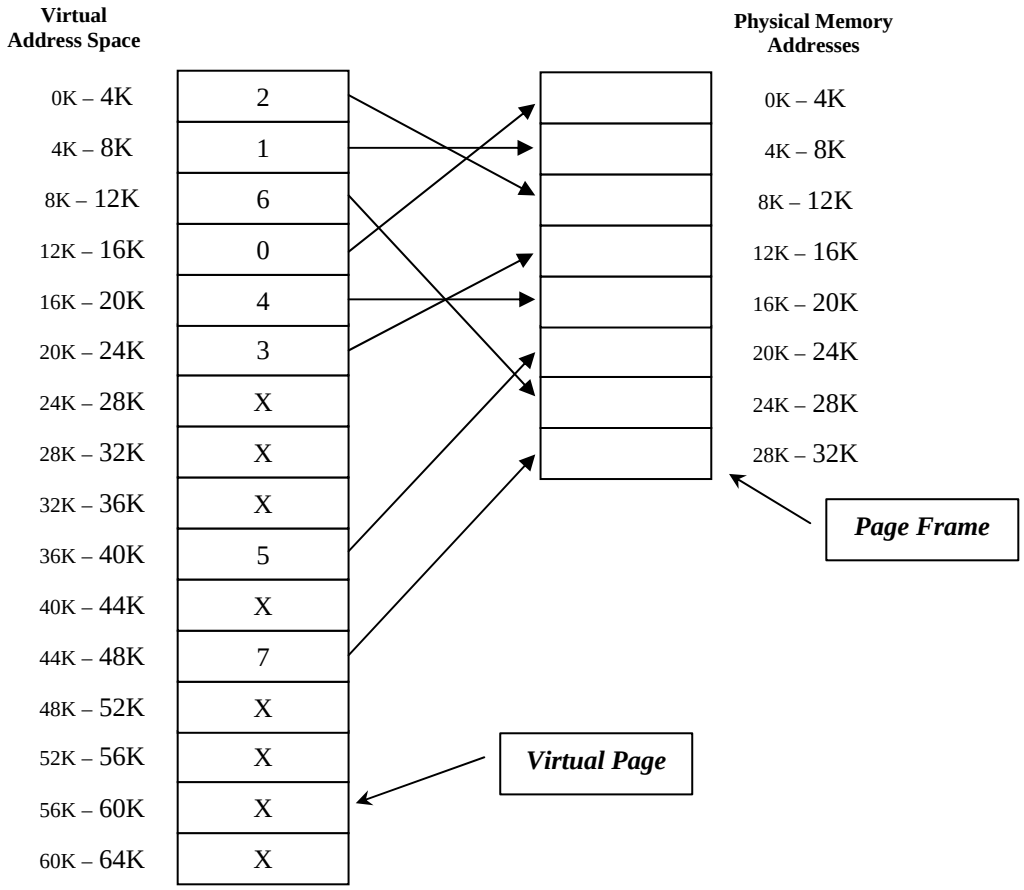
Introduction

The swapping methods we have looked at above are needed so that we can allocate memory to processes when they need it. But what happens when we do not have enough memory?
In the past, a system called ***overlays*** was used. This was a system that was the responsibility of the programmer. The program would be split into logical sections (called overlays) and only one overlay would be loaded into memory at a time. This meant that more programs could be running than would be the case if the complete program had to be in memory.
The downside of this approach is that the programmer had to take responsibility for splitting the program into logical sections. This was time consuming, boring and open to error.

It is no surprise that somebody eventually devised a method that allowed the computer to take over the responsibility. It is (Fotheringham, 1961) who is credited with coming up with the method that is now known as ***virtual memory***.
The idea behind virtual memory is that the computer is able to run programs even if the amount of physical memory is not sufficient to allow the program and all its data to reside in memory at the same time.
At the most basic level we can run a 500K program on a 256K machine. But we can also use virtual memory in a multiprogramming environment. We can run twelve programs in a machine that could, without virtual memory, only run four.

Paging

In a computer system that does not support virtual memory, when a program generates a memory address it is placed directly on the memory bus which causes the requested memory location to be accessed.
On a computer that supports virtual memory, the address generated by a program goes via a ***memory management unit*** (MMU). This unit maps virtual addresses to physical addresses.



This diagram shows how virtual memory operates. The computer in this example can generate 16-bit addresses. That is addresses between 0 and 64K. The problem is the computer only has 32K of physical memory so although we can write programs that can access 64K of memory, we do not have the physical memory to support that.
We obviously cannot fit 64K into the physical memory available so we have to store some of it on disc.

The virtual memory is divided into ***pages***. The physical memory is divided into ***page frames***. The size of the virtual pages and the page frames are the same size (4K in the diagram above). Therefore, we have sixteen virtual pages and eight physical pages. Transfers between disc and memory are done in pages.

Now let us consider what happens when a program generates a request to access a memory location. Assume a program tries to access address 8192. This address is sent to the MMU. The MMU recognises that this address falls in virtual page 2 (assume pages start at zero). The MMU looks at its page mapping and sees that page 2 maps to physical page 6. The MMU translates 8192 to the relevant address in physical page 6 (this being 24576). This address is output by the MMU and the memory board simply sees a request for address 24576. It does not know that the MMU has intervened. The memory board simply sees a request for a particular location, which it honours.

If a virtual memory address is not on a page boundary (as in the above example) then the MMU also has to calculate an offset (in fact, there is always an offset – in the above example it was zero).
As an exercise, and using the diagram above, work out the physical page and physical address that are generated by the MMU for each of the following addresses. The answers are in a separate handout.

Virtual Address	Physical Page ??	Physical Address ???
0		
45060		
16384		
21503		
24576		

So far, all we have managed to do is map sixteen virtual pages onto eight physical pages. We have not really achieved anything yet as, in effect, we have eight virtual pages which do not map to a physical page. In the diagram above, we represented these pages with an 'X'. In reality, each virtual page will have a ***present/absent bit*** which indicates if the virtual page is mapped to a physical page.

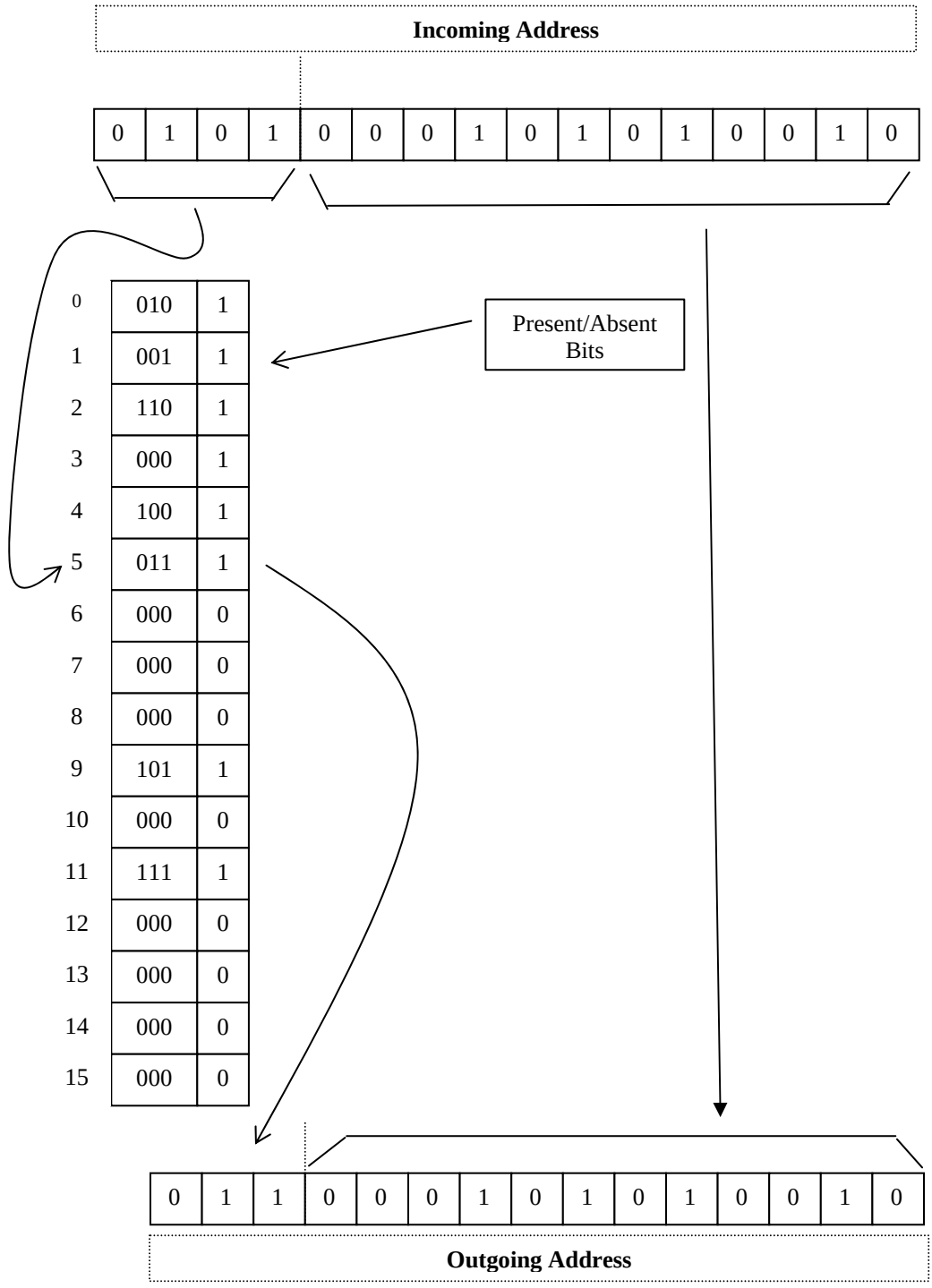
We need to look at what happens if the program tries to use an unmapped page. For example, the program tries to access address 24576 (i.e. 24K).
The MMU will notice that the page is unmapped (using the present/absent bit) and will cause a trap to the operating system. This trap is called a ***page fault***. The operating system would decide to evict one of the currently mapped pages and use that for the page that has just been referenced. The sequence of events would go like this.

- The program tries to access a memory location in a (virtual) page that is not currently mapped.
- The MMU causes a trap to the operating system. This results in a page fault.
- A little used virtual page is chosen (how this choice is made we will look at later) and its contents are written to disc.
- The page that has just been referenced is copied (from disc) to the virtual page that has just been freed.
- The virtual page frames are updated.
- The trapped instruction is restarted.

In the example we have just given (trying to access address 24576) the following would happen.
- The MMU would cause a trap to the operating system as the virtual page is not mapped to a physical location.
- A virtual page that is mapped is elected for eviction (we'll assume that page 11 is nominated).
- Virtual page 11 is mark as unmapped (i.e. the present/absent bit is changed).
- Physical page 7 is written to disc (we'll assume for now that this needs to be done). That is the physical page that virtual page 11 maps onto.
- Virtual page 6 is loaded to physical address 28672 (28K).
- The entry for virtual page 6 is changed so that the present/absent bit is changed. Also the 'X' is replaced by a '7' so that it points to the correct physical page.
- When the trapped instruction is re-executed it will now work correctly.

You might like to work through this to see how the mapping is changed.

It is interesting to look at how the MMU works. In particular, to consider why we have chosen to use a page size that is a power of 2. Take a look at this diagram.



The incoming address (20818) consists of 16 bits. The top four bits are masked off and make an entry into the virtual page table (in this case it provides an index to entry 5 (101 in binary) and finds that this page is

mapped to physical page 011 (3 in decimal). These three bits make up the top three bits of the physical page address. The other part of the incoming address is copied directly to the outgoing address.
Thus, the page table (courtesy of the MMU) has mapped virtual address 20818 to physical address 12626. If you look at the diagram that shows how virtual memory operates you will be able to follow this conversion.

The only other point we should, perhaps, consider, is why the first twelve bits of the incoming address can be copied directly to the output address. See if you can work it out before looking at the answer, which is in a separate handout.

Page Tables

The way we have described how virtual addresses map to physical addresses is how it works but we still have a couple of problems to consider.

The page table can be large

Assume a computer uses virtual addresses of 32 bits (very common) and the page size is 4K. This results in over 1 million pages (2^{32} / 4096 = 1048576).

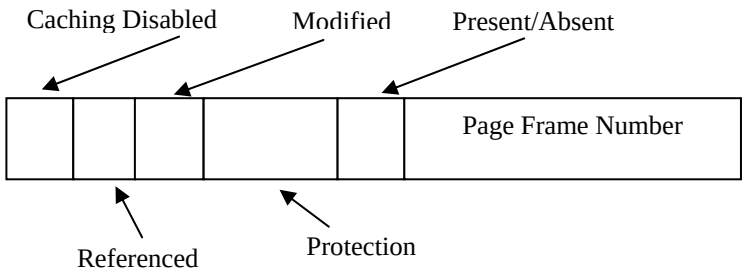
The mapping between logical and physical addresses must be fast

A typical instruction requires two memory accesses (to fetch the instruction and to fetch the data that the instruction will operate upon). Therefore, it requires two page table accesses in order to find the right physical location.

Assume an instruction takes 10ms. The mapping of the addresses must be done in a fraction of this time in order for this part of the system not to become a bottleneck.

In this course we are not going to look in detail at how these problems are overcome but the interested student might like to look at (Tanenbaum, 1992; 2001; 2008) which covers it in some detail.

However, we need to look at, in more detail, the structure of a page table so that we can use it in future material.



This diagram shows typical entries in a page table (although the exact entries are operating system dependent). The various components are described below.

Page Frame Number **:** This is the number of the physical page that this page maps to. As this is the whole point of the page, this can be considered the most important part of the page frame entry.

Present/Absent Bit **:** This indicates if the mapping is valid. A value of 1 indicates the physical page, to which this virtual page relates is in memory. A value of zero indicates the mapping is not valid and a page fault will occur if the page is accessed.

Protection	***:***	The protection bit could simply be a single bit which is set to 0 if the page can be read and written and 1 if the page can only be read. If three bits are allowed then each bit can be used to represent read, write and execute.
Modified	***:***	This bit is updated if the data in the page has been modified. This bit is used when the data in the page is evicted. If the modified bit is set, the data in the page frame needs to be written back to disc. If the modified bit is not set, then the data can simply be evicted, in the knowledge that the data on disc is already up to date.
Referenced	***:***	This bit is updated if the page has been referenced. This bit can be used when deciding which page should be evicted (we will be looking at its use later).
Caching Disabled	***:***	This bit allows caching to be disabled for the page. This is useful if a memory address maps onto a device register rather than to a memory address. In this case, the register could be changed by the device and it is important that the register is accessed, rather than using the cached value which may not be up to date.

Page Replacement Algorithms

Introduction

In the discussion above we said that when a page fault occurs we evict a page that is currently mapped and replace it with the page that we are currently trying to access. In doing this, we need to write the page we are evicting to disc, if it has been modified.
However, we have not said how we decide which page to evict. One of the easiest methods would be to choose a mapped page at random but this is likely to lead to degraded system performance. This is because the page chosen has a reasonable chance of being a page that will need to be used again in the near future. Therefore, it will be evicted (and may have to be written to disc) and then brought back into memory; maybe on the next instruction.
There has been a lot of work done in this area and in this section we describe some of the algorithms. The interested student might like to look at (Smith, 1978) which lists over 300 papers on this topic.

The Optimal Page Replacement Algorithm

You might recall, when we were discussing process scheduling, that there is an optimal scheme where we always schedule the shortest job first. This leads to the minimum average waiting time. The problem is we cannot identify the burst time of a process before it is run. Therefore, it may be optimal, but we cannot implement it.
This page replacement algorithm is similar in that it leads to an optimal solution, but we cannot implement it.
The basis of the algorithm is that we evict the page that we will not use for the longest period. For example, if we only have two pages to choose from, one of which will be used on the next instruction and the other will not be used for another one hundred instructions. It makes sense to evict the page that will be used for one hundred instructions.
The problem, of course, is that we cannot look into the future and decide which page to evict. But, if we could, we could implement an optimal algorithm.

But, if we cannot implement the algorithm then why bother discussing it? In fact, we can implement it, but only after running the program to see which pages we should evict and at what point. Once we know that we can run the optimal page replacement algorithm. We can then use this as a measure to see how other algorithms perform against this ideal.

The Not-Recently-Used Page Replacement Algorithm

In order to implement this algorithm we make use of the referenced and modified bits that were mentioned above. These bits are often implemented in hardware (as they are updated so much) but they can be simulated in software as follows.

When a process starts, all its page entries are marked as not in memory (i.e. the present/absent bit). When a page is referenced a page fault will occur. The R (reference) bit is set and the page table entry modified to point to the correct page. In addition the page is set to ***read only***. If the page is later written to, the M (modified) bit is set and the page is changed so that it is ***read/write***.

Updating the flags in this way allows a simple paging algorithm to be built.
When a process is started up all R and M bits are cleared (set to zero). Periodically (e.g. on each clock interrupt) the R bit is cleared. This allows us to recognise which pages have been recently referenced.
When a page fault occurs (so that a page needs to be evicted), the pages are inspected and divided into four categories based on their R and M bits.

Class 0 **:** Not Referenced, Not Modified
Class 1 **:** Not Referenced, Modified
Class 2 **:** Referenced, Not Modified
Class 3 **:** Referenced, Modified

The Not-Recently-Used (NRU) algorithm removes a page at random from the lowest numbered class that has entries in it. Therefore, pages which have not been referenced or modified are removed in preference to those that have not been referenced but have been modified (which is not as impossible as it sounds due to the fact that the reference bit is periodically reset).

Although, not an optimal algorithm, NRU often provides adequate performance and is easy to understand and implement.

The First-In, First-Out (FIFO) Page Replacement Algorithm

This algorithm simply maintains the pages as a linked list, with new pages being added to the end of the list. When a page fault occurs, the page at the head of the list (the oldest page) is evicted. Whilst simple to understand and implement a simple FIFO algorithm does not lead to good performance as a heavily used page is just as likely to be evicted as a lightly used page.

The Second Chance Page Replacement Algorithm

The second chance (SC) algorithm is a modification of the FIFO algorithm. When a page fault occurs the page at the front of the linked list is inspected. If it has not been referenced (i.e. its reference bit is clear), it is evicted. If its reference bit is set, then it is placed at the end of the linked list and its reference bit reset. The next page is then inspected. In this way, a page that has been referenced will be given a second chance.

In the worst case, SC operates in the same way as FIFO. Take the situation where the linked list consists of pages which all have their reference bit set. The first page, call it *a*, is inspected and placed at the end of the list, after having its R bit cleared. The other pages all receive the same treatment. Eventually page *a* reaches the head of the list and is evicted as its reference bit is now clear.
Therefore, even when all pages in a list have their reference bit set, the algorithm will always terminate.

The Clock Page Replacement Algorithm

The clock page (CP) algorithm differs from SC only in its implementation.
Whilst SC is a reasonable algorithm it suffers in the amount of time it has to devote to the maintenance of the linked list. It is more efficient to hold the pages in a circular list and move the pointer rather than move the pages from the head of the list to the end of the list.
It is called the clock page algorithm as it can be visualised as a clock face with the hand (pointer) pointing at the pages, which represent the numbers on the clock.

The Least Recently Used (LRU) Page Replacement Algorithm

Although we cannot implement an optimal algorithm by evicting the page that will not be used for the longest time in the future, we can approximate the algorithm by keeping track of when a page was last

used. If a page has recently been used then it is likely that it will be used again in the near future. Conversely, a page that has not been used for some time is unlikely to be used again in the near future. Therefore, if we evict the page that has not been used for the longest amount of time we can implement a least recently used (LRU) algorithm.

Whilst this algorithm can be implemented (unlike the optimal algorithm) it is not cheap. Ideally, we need to maintain a linked list of pages which are sorted in the order in which they have been used. To maintain such a list is prohibitively expensive (even in hardware) as deleting and moving list elements is a time consuming process. Sorting a list is also expensive.

However, there are ways that LRU can be implemented in hardware. One way is as follows.

The hardware is equipped with a counter (typically 64 bits). After each instruction the counter is incremented.
In addition, each page table entry has a field large enough to accommodate the counter. Every time the page is referenced the value from the counter is copied to the page table field. When a page fault occurs the operating system inspects all the page table entries and selects the page with the lowest counter. This is the page that is evicted as it has not been referenced for the longest time.

Another hardware implementation of the LRU algorithm is given below.
If we have *n* page table entries a matrix of *n* x *n* bits, initially all zero, is maintained. When a page frame, *k*, is referenced then all the bits of the *k* row are set to one and all the bits of the *k* column are set to zero. At any time the row with the lowest ***binary*** value is the row that is the least recently used (where row number = page frame number). The next lowest entry is the next recently used; and so on.

If we have four page frames and access them as follows

0 1 2 3 2 1 0 3 2 3

it leads to the algorithm operating as follows.
It might be worth working through it and calculating the binary value of each row to see which page frame would be evicted.

Page

	0	1	2	3
0	0	1	1	1
1	0	0	0	0
2	0	0	0	0
3	0	0	0	0

(a)

Page

	0	1	2	3
0	0	0	1	1
1	1	0	1	1
2	0	0	0	0
3	0	0	0	0

(b)

Page

	0	1	2	3
0	0	0	0	1
1	1	0	0	1
2	1	1	0	1
3	0	0	0	0

(c)

Page

	0	1	2	3
0	0	0	0	0
1	1	0	0	0
2	1	1	0	0
3	1	1	1	0

(d)

Page

	0	1	2	3
0	0	0	0	0
1	1	0	0	0
2	1	1	0	1
3	1	1	0	0

(e)

	0	1	2	3
0	0	0	0	0
1	1	0	0	0
2	1	0	0	1
3	1	0	0	0

(f)

	0	1	2	3
0	0	1	1	1
1	0	0	1	1
2	0	0	0	1
3	0	0	0	0

(g)

	0	1	2	3
0	0	1	1	0
1	0	0	1	0
2	0	0	0	0
3	1	1	1	0

(h)

	0	1	2	3
0	0	1	0	0
1	0	0	0	0
2	1	1	0	1
3	1	1	0	0

(h)

	0	1	2	3
0	0	1	0	0
1	0	0	0	0
2	1	1	0	0
3	1	1	1	0

(i)

LRU in Software

One of the main drawbacks with implementing LRU in hardware is that if the hardware does not provide these facilities then the operating system designers, obviously, cannot make use of them. Instead we need to implement a similar algorithm in software.

One method (called the ***Not Frequently Used – NFU*** algorithm) associates a counter with each page. This counter is initially zero but at each clock interrupt the operating system scans all the pages and adds the R bit for the page to its counter. As this is either zero or one, the counter either gets incremented or it does not. When a page fault occurs the page with the lowest counter is selected for replacement.

The main problem with NFU is that it never forgets anything. For example, if a multi-pass compiler is running, at the end of the first pass the pages may not be needed anymore. However, as they will have high counts they will not be replaced but pages from the second pass, which still have low counts, will be replaced.
In fact, the situation could be even worse. If the first pass made a lot of memory references but the second pass does not make as many, then the pages from the first pass will always have higher counts than the second pass and will therefore remain in memory.

To alleviate this problem we can make a modification to NFU so that it closely simulates NRU. The modification is in two parts.
1. The counters are shifted right one bit before the R bit is added.
2. The R bit is added to the leftmost bit rather than the rightmost bit.

This implements a system of ***aging***.

Look at the diagram below. This represents a page table with six entries. Working from right to left we are showing the state of each of the pages (only the counter entries) at each of the six clock ticks.
Consider the (a) column. After clock tick zero the *R* flags for the six pages are set to 1, 0, 1, 0, 1 and 1. This indicates that pages 0, 2, 4 and 5 were referenced.
This results in the counters being set as shown. We assume they all started at zero so that the shift right, in effect, did nothing and the reference bit was added to the leftmost bit.
If you look at the (b) clock tick you should be able to follow the algorithm and similarly for (c) to (e) clicks.

	R bits for pages 0-5 **Clock Tick 0**	*R* bits for pages 0-5 **Clock Tick 1**	*R* bits for pages 0-5 **Clock Tick 2**	*R* bits for pages 0-5 **Clock Tick 3**	*R* bits for pages 0-5 **Clock Tick 4**
	101011	110010	110101	100010	011000
Page					
0	10000000	11000000	11100000	11110000	01111000
1	00000000	10000000	11000000	01100000	10110000
2	10000000	01000000	00100000	00010000	10001000
3	00000000	00000000	10000000	01000000	00100000
4	10000000	11000000	01100000	10110000	01011000
5	10000000	01000000	10100000	01010000	00101000
	(a)	(b)	(c)	(d)	(e)

When a page fault occurs, the counter with the lowest value is removed. It is obvious that a page that has not been referenced for, say, four clocks ticks will have four zeroes in the leftmost positions and will have a lower value that a page that has not been referenced for three clock ticks.

The NFU algorithm differs from LRU in two ways.
Using the matrix LRU implementation we update the matrix after each instruction. This, in effect, gives us more detailed information. If you look at pages 3 and 5 in the above diagram, after clock tick 4, both pages have not been referenced for the last two clock ticks. But when they were referenced we do not know how early (or late) in that clock tick the pages were referenced. All we are able to do is evict page 3 as this has not been referenced anymore, whereas page 5 has.

The aging counters of NFU have a finite size. If two counters have a zero value, all we can do is choose one at random. Assuming that the counters are 8 bits then we do not know if the page was accessed 9 clock ticks ago or 1000 clock ticks ago. In practise, 8 bits are commonly used. If a clock tick is around 20ms and a page has a zero counter this means that the page has not been referenced for 160ms. In this case, it is probably not that important.

Design Issues for Paging

In this section we are going to look at some other aspects of designing a paging system so that we can achieve good performance.

Working Set Model

The most obvious way to implement a paging system is to start a process with none of its pages in memory. When the process starts to execute it will try to get its first instruction, which will cause a page fault. Other page faults will quickly follow as the process tries to access the stack, global variables and executes other instructions. After a period of time the process should start to find that most of its pages are in memory and not so many page faults occur.
This type of paging is known as ***demand paging*** as pages are brought into memory on demand.

The reason that page faults decrease (and then stabilise) is because processes normally exhibit a ***locality of reference***. This means that at a particular execution phase of the process it only uses a small fraction of the pages available to the entire process. The set of pages that is currently being used is called its ***working set*** (Denning, 1968a; Denning 1980).
If the entire working set is in memory then no page faults will occur. Only when the process moves onto the next phase of execution (e.g. the next phase of a compiler) will page faults begin to occur as pages not part of the existing working set are brought into memory.
If the memory of the computer is not large enough to hold the entire working set, then pages will constantly be copied out to disc and subsequently retrieved. This drastically slows a process down as the time taken to execute an instruction is a lot faster than disc accesses.
A process which causes page faults every few instructions is said to be ***thrashing*** (Denning 1968b).

In a system that allows many processes to run at the same time (or at least give that illusion) it is common to move all the pages for a process to disc (i.e. swap it out). When the process is restarted we have to decide what to do. Do we simply allow demand paging, so that as the process raises page faults, it pages are gradually brought into memory? Or do we move all its working set into memory so that it can continue with minimal page faults?
It will come as no surprise that the second option is to be preferred. We would like to avoid a process, every time it is restarted, raising page faults. In order to do this the paging system has to keep track of the processes' working set so that it can be loaded into memory before it is restarted.
The approach is called the ***working set model*** (Denning, 1970). Its aim, as we have stated, is to avoid page faults being raised. This method is also known as ***prepaging***.

A problem arises when we try to implement the working set model as we need to know which pages make up the working set.
One solution is to use the aging algorithm described above. Any page that contains a 1 in *n* high order bits is deemed to be a member of the working set. The value of *n* has to be experimentally found although it has been found that the value is not that sensitive

Paging Daemons

If a page fault occurs it is better if there are plenty of free pages for the page to be copied to. If we have the situation where every page is being used we have to find a page to evict and we may have to write the page to disc before evicting it.

Many systems have a background process called a ***paging daemon***. This process sleeps most of the time but runs at periodic intervals. Its task is to inspect the state of the page frames and, if too few pages are free, it selects pages to evict using the page replacement algorithm that is being used.
A further performance improvement can be achieved by remembering which page frame a page has been evicted from. If the page frame has not been overwritten when the evicted page is needed again then the page frame is still valid and the data does not have to copied from disc again.
In addition the paging daemon, if it does not evict pages, can ensure they are ***clean***. That is they have been written to disc so that if they are later evicted they can simply be overwritten without having to write data to disc.

References

- Denning, P.J. 1968. The Working Set Model for Program Behaviour. Communications of the ACM, Vol. 11, pp 323-333.
- Denning, P.J. 1968a. Working Sets Past and Present. IEEE Trans on Software Engineering, Vol. SE-6, pp 64-84.
- Denning, P.J. 1968b. Thrashing: Its Causes and Prevention. Proceedings AFIPS National Computer Conference, pp 915-922.
- Denning, P.J. 1970. Virtual Memory. Computing Surveys, Vol. 2, pp 153-189.
- Fortheringham, J. 1961. Dynamic Storage Allocation in the Atlas Computer Including an Automatic Use of a Backing Store. Communications of the ACM, Vol. 4, pp 435-436
- Kaufman, A. 1984. Tailored-List and Recombination-Delaying Buddy Systems. ACM Trans. On Programming Languages and Systems, Vol. 6, pp 118-125
- Knowlton, K.C. 1965. A Fast Storage Allocator. Communications of the ACM, vol 8, pp 623-625.
- Knuth, D.E. 1973. The Art of Computer Programming, Volume 1 : Fundamental Algorithms, 2nd ed, Reading, MA, Addison-Wesley
- Peterson, J.L., Norman, T.A. 1977. Buddy Systems. Communications of the ACM, Vol. 20, pp 421-431
- Smith, A.J. 1978. Bibliography on Paging and Related Topics. Operating Systems Review, Vol. 12, pp 39-56
- Tanenbaum, A., S. 1992. Modern Operating Systems (1st ed.). Prentice Hall.
- Tanenbaum, A., S. 2001. Modern Operating Systems (2nd ed.). Prentice Hall.
- Tanenbaum, A., S. 2008. Modern Operating Systems (3rd ed.). Prentice Hall.