Memory Management

Operating Systems

G53OPS: Operating Systems

Graham Kendall

Memory Management

Operating Systems

- Memory Management consists of many tasks, 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 it
  - Moving data from memory to disc, when the physical capacity becomes full, and vice versa

Memory Management

Operating Systems

Monoprogramming

- Only allow a single process in memory and only allow one process to run at any one time
  - Very Simple
  - No swapping of processes to disc when we run out of memory
  - No problems in having separate processes in memory

Memory Management

Operating Systems

Monoprogramming

- Even this simple scheme has its problems.
  - We have not yet considered the data that the program will operate upon
  - Process must be self contained
    - e.g. drivers in every process
  - Operating system can be seen as a process – so we have two anyway

Memory Management

Operating Systems

Monoprogramming

- Additional Problems
  - Monoprograming is unacceptable as multi-programming is expected
  - Multiprogramming makes more effective use of the CPU
- Could allow only a single process in memory at any one time but allow multi-programming
  - i.e. swap out to disc
  - Context switch would take time

Memory Management

Operating Systems

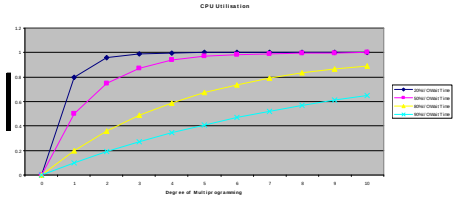
Modelling Multi-programming

- Assumption that multiprogramming can improve the utilisation of the CPU – is this true?
  - Intuitively it is
  - But, can we model it?

Modelling Multi-programming

- Probabilistic model
  - A process spends *p* percent of its time waiting for I/O
  - There are *n* processes in memory
  - The probability that all *n* processes are waiting for I/O (CPU is idle) is p^n
  - The CPU utilisation is then given by
    - CPU Utlisation = 1 - p^n

Modelling Multi-programming



- With an I/O wait time of 20%, almost 100% CPU utilisation can be achieved with four processes
- I/O wait time of 90% then with ten processes, we only achieve just above 60% utilisation
- The more processes we run, the better the CPU utilisation

Modelling Multi-programming

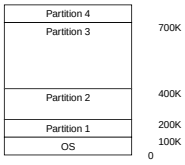
- Model assumes that all the processes are independent. This is not true
- More complex models could be built using queuing theory but we can still use this simplistic model to make approximate predictions.

Modelling Multi-programming

- Example of model use
  - 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 of memory, it allows us to run another five processes
  - We can now achieve about 86% CPU utilisation
  - If we add another megabyte of memory (fourteen processes) we will find that the CPU utilisation will increase to about 96%.

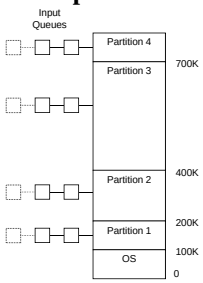
Multi-programming with fixed partitions

- Accept that mulitiprogramming is a good idea
- How do we organise the available memory?
- One method is to divide the memory into fixed sized partitions
- Partitions can be of different sizes but their size remain fixed



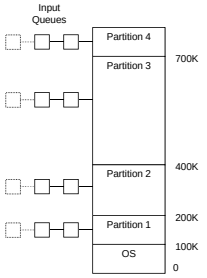
Multi-programming with fixed partitions

- Memory divided into four partitions
- When job arrives it is placed in the input queue for the smallest partition that will accommodate it



Multi-programming with fixed partitions

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



Multi-programming with fixed partitions

- Just 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

Multi-programming with fixed partitions

- Alternative search strategy
  - Search the entire input queue looking for the largest job that fits into the partition
- Do not waste a large partition on a small job but smaller jobs are discriminated against
- Have at least one small partition or ensure that small jobs only get skipped a certain number of times.

Relocation and Protection

- Introducing multiprogramming gives two problems
  - **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

Relocation and Protection

- Solution to solve both relocation and protection
  - Have two registers (called the base and limit registers)
  - The base register stores the start address of the partition
  - The limit register holds the length of the partition
- Additional benefit of this scheme is moving programs in memory

Swapping

- Fixed partitions becomes ineffective when we have more processes than we can fit into memory at one time (e.g. when timesharing)
- Solution : Hold some of the processes on disc and swap processes between disc and main memory as necessary

Swapping: Variable Partitions

- Because we are swapping processes between memory and disc does not stop us using fixed partition sizes
- But, the reason we are having swap processes out to disc is because memory is a scare resource and:
  - Fixed partitions can be wasteful of memory
  - We might want to run more processes that we can fit into memory at any given time.

Swapping: Variable Partitions

- Obvious step is to use variable partition sizes
  - That is a partition size can change as the need arises
- Variable partitions
  - The number of partitions vary
  - The sizes of the partitions vary
  - The starting addresses of the partitions varies

Swapping: Variable Partitions

- Makes for a more effective memory management system but it makes the process of maintaining the memory much more difficult
- As memory is allocated and deallocated holes will appear in the memory (fragmentation)
  - Eventually holes will be too small to have a process allocated to it
  - We could shuffle the memory downwards (memory compaction) but this is inefficient

Swapping: Variable Partitions

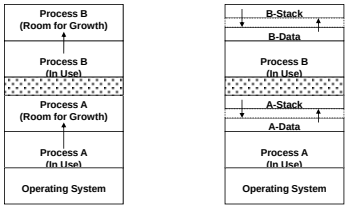
- If processes are allowed to dynamically request more memory what happens if a process requests extra memory such that increasing its partition size is impossible without it having to overwrite another partitions memory
  - Wait until memory is available that the process is able to grow into it?
  - Terminate the process?
  - Move the process to a hole in memory that is large enough to accommodate the growing process?
  - Only realistic option is the last one but very inefficient

Swapping: Variable Partitions

- None of the above proposed solutions are ideal so it would seem a good idea to allocate more memory than is initially required
- Most processes will have two growing data segments
  - Stack
  - Heap

Swapping: Variable Partitions

- 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

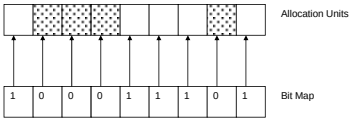


Memory Management

Operating Systems

Swapping: Bit Maps

- 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 is one, then the memory is being used



Memory Management

Operating Systems

Swapping: Bit Maps

- Main decision is the size of the allocation unit
  - The smaller the allocation unit, the larger the bit map has to be
  - But a larger allocation unit could waste memory as we may not use all the space allocated in each allocation unit

Memory Management

Operating Systems

Swapping: Bit Maps

- Another problem comes 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 (256/4 = 64)
  - Slow operation, therefore bit maps are not often used

Memory Management

Operating Systems

Swapping: Linked Lists

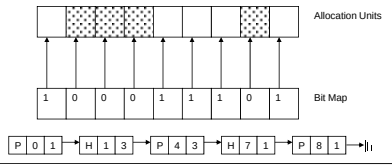
- Free and allocated memory can be represented as a linked list
- The memory shown on the bit map slide can be represented as a linked list as follows
- Each entry in the list holds the following data
  - P or H : for Process or Hole
  - Starting segment address
  - The length of the memory segment

Memory Management

Operating Systems

Swapping: Linked Lists

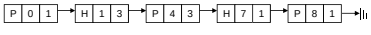
- Each entry in the list holds the following data
  - P or H : for Process or Hole
  - Starting segment address
  - The length of the memory segment



Memory Management

Operating Systems

Swapping: Linked Lists



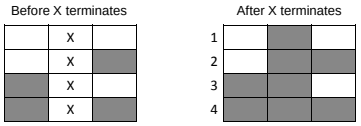
- In the list above, processes follow holes and vice versa
- Processes can be next to each other and we need to keep them as separate elements in the list
- Consecutive holes can always be merged into a single list entry

Swapping: Linked Lists

- This leads to the following observations when a process terminates
  - A terminating process can have four combinations of neighbours (ignoring the start and the end of the list)
  - Consecutive holes, on the other hand, can always be merged into a single list entry

Swapping: Linked Lists

- If X is the terminating process the four combinations are

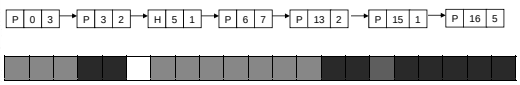


- In the first option we simply have to replace the P by an H, other than that the list remains the same
- In the second option we merge two list entries into one and make the list one entry shorter
- Option three is effectively the same as option 2
- For the last option we merge three entries into one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

Swapping: Linked Lists

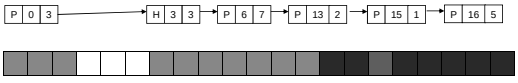
- Update the linked list if the following occurs
  - Process [P,3,2] terminates
  - Then process [P,13,2] terminates
  - Then process [P,6,7] terminates

Swapping: Linked Lists



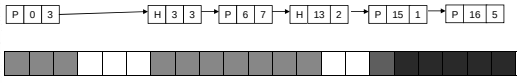
- Update the linked list if the following occurs
  - Process [P,3,2] terminates
  - Process [P,13,2] terminates
  - Process [P,6,7] terminates

Swapping: Linked Lists



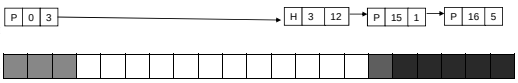
- Update the linked list if the following occurs
  - Process [P,3,2] terminates
  - Process [P,13,2] terminates
  - Process [P,6,7] terminates

Swapping: Linked Lists



- Update the linked list if the following occurs
  - Process [P,3,2] terminates
  - Process [P,13,2] terminates
  - Process [P,6,7] terminates

Swapping: Linked Lists



- Update the linked list if the following occurs
  - Process [P,3,2] terminates
  - Process [P,13,2] terminates
  - Process [P,6,7] terminates

Swapping: Linked Lists

- How could we find the best place to allocate memory?

Swapping: Linked Lists

- When we need to allocate memory, storing the list in segment address order allows us to implement various strategies
  - First Fit
  - Best Fit
  - Worst Fit
- Any other optimisations we could do?

Swapping: Linked Lists

- All three algorithms can be speeded up if we maintain two lists
  - One for processes
  - One for holes
- Allocation of memory is speeded up as we only have to search the hole list
- Any disadvantages with this method?

Swapping: Linked Lists

- Downside is that list maintenance is complicated
- Maintaining two lists allow 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

Swapping: Linked Lists

- 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 Management

Operating Systems

Swapping: Buddy System

- **Observation**: 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

Memory Management

Operating Systems

Swapping: Buddy System

- The Buddy System is a memory allocation that works on the basis of using binary numbers as these are fast for computers to manipulate

Memory Management

Operating Systems

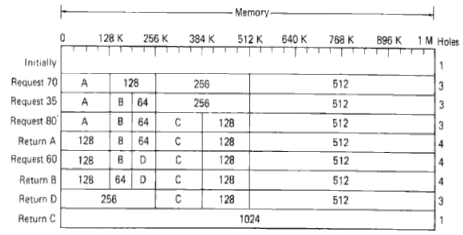
Swapping: Buddy System

- Lists are maintained which stores 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

Memory Management

Operating Systems

Swapping: Buddy System



Memory Management

Operating Systems

Swapping: Buddy System

- The buddy system is fast as 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

Memory Management

Operating Systems

Swapping: Buddy System

- Try question 5, from the 2000-2001 examination

Buddy System (Q5: 2000-2001)

a) The buddy system is a memory management scheme that uses variable sized partitions.
Explain the basic principle behind the buddy system.
(5)

b) Assume a computer with a memory size of 256K, initially empty. Requests are received for blocks of memory of 17K, 6K, 63K and 9K. Show how the buddy system would deal with each request, showing the memory layout at each stage and the status of the lists at each stage.
(8)

(c) The processes terminate in the following order; 6K, 9K, 17K and 63K. Discuss what happens as each process terminates.
(4)

d) Describe and evaluate an alternative to the buddy system
(8)

Buddy System (Q5: 2000-2001)

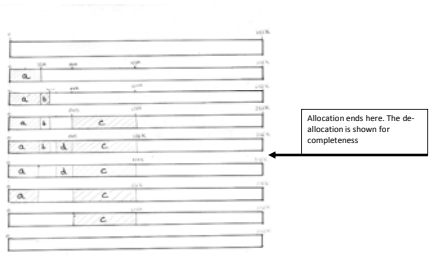
a) The buddy system is a memory management scheme that uses variable sized partitions.
Explain the basic principle behind the buddy system.
(5)

If we keep a list of holes (in memory) 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 list 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*** is a memory allocation that works on the basis of using binary numbers as these are fast for computers to manipulate.
Lists are maintained which stores 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 256K memory we require 19 lists.
If we assume we have 256K of memory and it is all unused then there will be one entry in the 256K list; and all other lists will be empty.

Buddy System (Q5: 2000-2001)

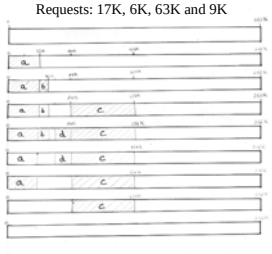
b) Assume a computer with a memory size of 256K, initially empty. Requests are received for blocks of memory of 17K, 6K, 63K and 9K. Show how the buddy system would deal with each request, showing the memory layout at each stage and the status of the lists at each stage.
(8)

Buddy System (Q5: 2000-2001)



Buddy System (Q5: 2000-2001)

Requests: 17K, 6K, 63K and 9K



List No.	Block Size	Start	17K	6K	63K	9K
1	1					
2	2					
3	4					
4	8					
5	16					
6	32					
7	64					
8	128					
9	256					
10	512					
11	1024 (1K)					
12	2048 (2K)					
13	4096 (4K)					
14	8192 (8K)			40K	40K	40K
15	16384 (16K)			48K	48K	
16	32768 (32K)		32K			
17	65536 (64K)		64K	64K		
18	131072 (128K)		128K	128K	128K	128K
19	262144 (256K)	0				

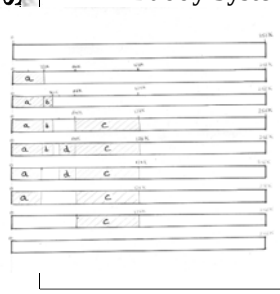
Buddy System (Q5: 2000-2001)

(c) The processes terminate in the following order; 6K, 9K, 17K and 63K. Discuss what happens as each process terminates.
(4)

The effect of each of these is described below and the changing lists are also shown.

- When the 6K process is returned (b in the above diagram), the 8K slot of memory is returned. This returns a block of memory from 32K to 40K. Checking the 8K list, it is found that there is an adjacent block of memory, which can be merged into a 16K block. Therefore the result is that the two 8K blocks are merged into a 16K block and this is added to the list.
- When the 9K block is returned (d in diagram) this releases a 16K block of memory from 48K to 64K. Checking the 16K list there is a block from 32K (to 48K). As the two 16K blocks are consecutive it allows these two blocks to be merged into a 32K block.
- The 17K block (a in diagram) returns a 32K block of memory from 0K to 32K. This can be merged with the block from 32K to 64K, giving a block in the 64K list.
- The final release of memory (64K, c in diagram) allows two 64K blocks to be merged into a 128K block and then two 128K blocks to be merged to return to a position where the memory is empty and there is only one list entry in the 256K block starting at 0K.

Buddy System (Q5: 2000-2001)



List No.	Block Size	Initial	6K	9K	17K	63K
1	1					
2	2					
3	4					
4	8					
5	16					
6	32					
7	64					
8	128					
9	256					
10	512					
11	1024 (1K)					
12	2048 (2K)					
13	4096 (4K)					
14	8192 (8K)	40K				
15	16384 (16K)		32K			
16	32768 (32K)			32K		
17	65536 (64K)				0K	
18	131072 (128K)	128K	128K	128K	128K	
19	262144 (256K)					0K

Buddy System (Q5: 2000-2001)

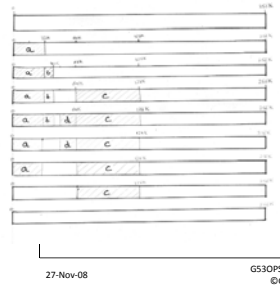
(c) The processes terminate in the following order; 6K, 9K, 17K and 63K. Discuss what happens as each process terminates.

(4)

The effect of each of these is described below and the changing lists are also shown.

- When the 6K process is returned (b in the above diagram), the 8K slot of memory is returned. This returns a block of memory from 32K to 40K. Checking the 8K list, it is found that there is an adjacent block of memory, which can be merged into a 16K block. Therefore the result is that the two 8K blocks are merged into a 16K block and this is added to the list.
- When the 9K block is returned (d in diagram) this releases a 16K block of memory from 48K to 64K. Checking the 16K list there is a block from 32K (to 48K). As the two 16K blocks are consecutive it allows these two blocks to be merged into a 32K block.
- The 17K block (a in diagram) returns a 32K block of memory from 0K to 32K. This can be merged with the block from 32K to 64K, giving a block in the 64K list.
- The final release of memory (64K, c in diagram) allows two 64K blocks to be merged into a 128K block and then two 128K blocks to be merged to return to a position where the memory is empty and there is only one list entry in the 256K block starting at 0K.

Buddy System (Q5: 2000-2001)



List No.	Block Size	Initial	6K	9K	17K	63K
1	1					
2	2					
3	4					
4	8					
5	16					
6	32					
7	64					
8	128					
9	256					
10	512					
11	1024 (1K)					
12	2048 (2K)					
13	4096 (4K)					
14	8192 (8K)	40K				
15	16384 (16K)		32K			
16	32768 (32K)			32K		
17	65536 (64K)				0K	
18	131072 (128K)	128K	128K	128K	128K	
19	262144 (256K)					0K

Buddy System (Q5: 2000-2001)

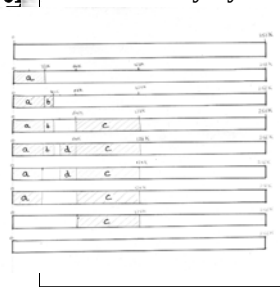
(c) The processes terminate in the following order; 6K, 9K, 17K and 63K. Discuss what happens as each process terminates.

(4)

The effect of each of these is described below and the changing lists are also shown.

- When the 6K process is returned (b in the above diagram), the 8K slot of memory is returned. This returns a block of memory from 32K to 40K. Checking the 8K list, it is found that there is an adjacent block of memory, which can be merged into a 16K block. Therefore the result is that the two 8K blocks are merged into a 16K block and this is added to the list.
- When the 9K block is returned (d in diagram) this releases a 16K block of memory from 48K to 64K. Checking the 16K list there is a block from 32K (to 48K). As the two 16K blocks are consecutive it allows these two blocks to be merged into a 32K block.
- The 17K block (a in diagram) returns a 32K block of memory from 0K to 32K. This can be merged with the block from 32K to 64K, giving a block in the 64K list.
- The final release of memory (64K, c in diagram) allows two 64K blocks to be merged into a 128K block and then two 128K blocks to be merged to return to a position where the memory is empty and there is only one list entry in the 256K block starting at 0K.

Buddy System (Q5: 2000-2001)



List No.	Block Size	Initial	6K	9K	17K	63K
1	1					
2	2					
3	4					
4	8					
5	16					
6	32					
7	64					
8	128					
9	256					
10	512					
11	1024 (1K)					
12	2048 (2K)					
13	4096 (4K)					
14	8192 (8K)	40K				
15	16384 (16K)		32K			
16	32768 (32K)			32K		
17	65536 (64K)				0K	
18	131072 (128K)	128K	128K	128K	128K	
19	262144 (256K)					0K

Buddy System (Q5: 2000-2001)

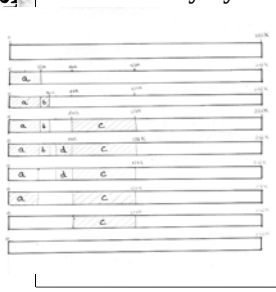
(c) The processes terminate in the following order; 6K, 9K, 17K and 63K. Discuss what happens as each process terminates.

(4)

The effect of each of these is described below and the changing lists are also shown.

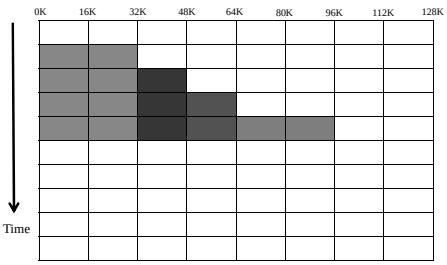
- When the 6K process is returned (b in the above diagram), the 8K slot of memory is returned. This returns a block of memory from 32K to 40K. Checking the 8K list, it is found that there is an adjacent block of memory, which can be merged into a 16K block. Therefore the result is that the two 8K blocks are merged into a 16K block and this is added to the list.
- When the 9K block is returned (d in diagram) this releases a 16K block of memory from 48K to 64K. Checking the 16K list there is a block from 32K (to 48K). As the two 16K blocks are consecutive it allows these two blocks to be merged into a 32K block.
- The 17K block (a in diagram) returns a 32K block of memory from 0K to 32K. This can be merged with the block from 32K to 64K, giving a block in the 64K list.
- The final release of memory (64K, c in diagram) allows two 64K blocks to be merged into a 128K block and then two 128K blocks to be merged to return to a position where the memory is empty and there is only one list entry in the 256K block starting at 0K.

Buddy System (Q5: 2000-2001)



List No.	Block Size	Initial	6K	9K	17K	63K
1	1					
2	2					
3	4					
4	8					
5	16					
6	32					
7	64					
8	128					
9	256					
10	512					
11	1024 (1K)					
12	2048 (2K)					
13	4096 (4K)					
14	8192 (8K)	40K				
15	16384 (16K)		32K			
16	32768 (32K)			32K		
17	65536 (64K)				0K	
18	131072 (128K)	128K	128K	128K	128K	
19	262144 (256K)					0K

Discussion



Buddy System (Q5: 2000-2001)

d) Describe and evaluate an alternative to the buddy system (8)

Two alternatives have been presented; managing memory with bit maps and managing memory with linked lists.

Swapping: Buddy System

- The buddy system is fast as 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

Swapping: Buddy System

- This type of wastage is known as internal fragmentation. As the wasted memory is internal to the allocated segments
- Opposite is external fragmentation where the wasted memory appears between allocated segments

Virtual Memory

- Swapping allows us to allocate memory to processes when they need it. But what happens when we do not have enough memory?

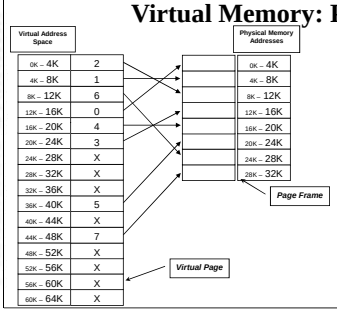
Virtual Memory

- In the past, overlays were used
  - Responsibility of the programmer
  - Program split into logical sections (called overlays)
  - Only one overlay would be loaded into memory at a time
  - Meant that more programs could be running than would be the case if the complete program had to be in memory
- Downsides
  - Programmer had to take responsibility for splitting the program into logical sections
  - Time consuming, boring and open to error

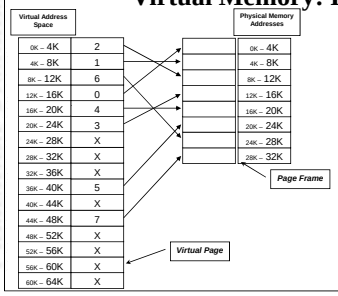
Virtual Memory: Paging

- 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

Virtual Memory: Paging



Virtual Memory: Paging



- 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.

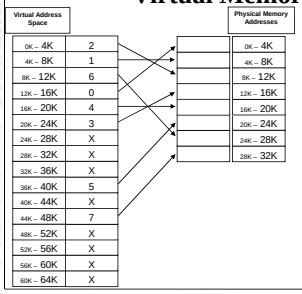
Virtual Memory: Paging

- If a virtual memory address is not on a page boundary (unlike 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)
- Exercise in Notes

Virtual Memory: Paging

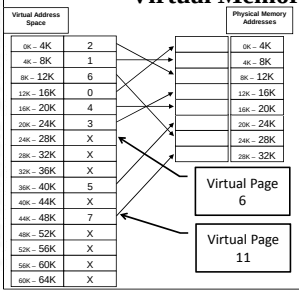
- We have not really achieved anything yet as, in effect, we have eight virtual pages which do not map to a physical page
- Each virtual page will have a present/absent bit which indicates if the virtual page is mapped to a physical page

Virtual Memory: Paging



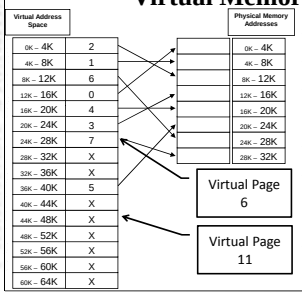
- What happens if we try to use an unmapped page? For example, the program tries to access address 24576 (i.e. 24K)
- The MMU will notice that the page is unmapped and will cause a trap (page fault) to the operating system
- The operating system will decide to evict one of the currently mapped pages and use that for the page that has just been referenced
- 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.

Virtual Memory: Paging



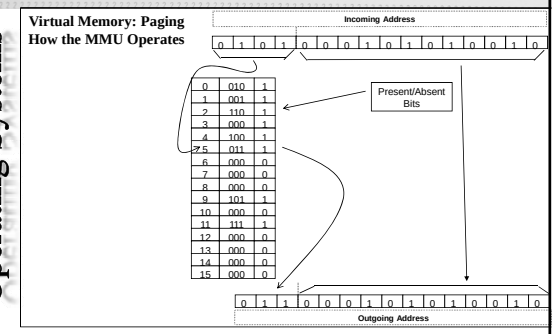
- A virtual page that is mapped is elected for eviction (let's assume 11)
- Virtual page 11 is mark as unmapped (i.e. the present/absent bit is changed)
- Physical page 7 is written to disc (we'll assume for now that this needs to be done). That is the physical page that virtual page 11 maps onto
- Virtual 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

Virtual Memory: Paging

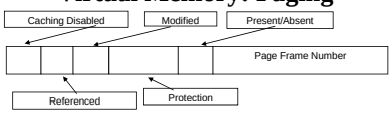


- A virtual page that is mapped is elected for eviction (let's assume 11)
- Virtual page 11 is mark as unmapped (i.e. the present/absent bit is changed)
- Physical page 7 is written to disc (we'll assume for now that this needs to be done). That is the physical page that virtual page 11 maps onto
- Virtual 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

Virtual Memory: Paging How the MMU Operates

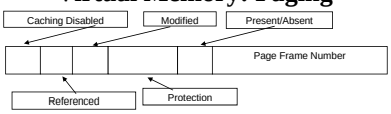


Virtual Memory: Paging



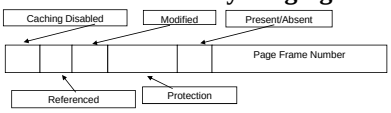
- **Page Frame Number:** This is the number of the physical page that this page maps to. As this is the whole point of the page, this can be considered the most important part of the page frame entry
- **Present/Absent Bit:** This indicates if the mapping is valid. A value of 1 indicates the physical 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

Virtual Memory: Paging



- **Protection:** The protection bit could simply be a single bit which is set to 0 if the page cane 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 is 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

Virtual Memory: Paging



- **Referenced:** This bit is updated if the page is 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

Virtual Memory: Page Replacement

- Choose a mapped page at random
  - Likely to lead to degraded system performance
  - Page chosen has a reasonable chance of being a page that will need to be used again in the near future
- The Optimal Page Replacement Algorithm
  - Evict the page that we will not use for the longest period
  - Problem is we cannot look into the future and decide which page to evict
  - But, if we could, we could implement an optimal algorithm

Virtual Memory: Page Replacement

- 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 at what point
- We can then use this as a measure to see how other algorithms perform against this ideal

Virtual Memory: Page Replacement (NRU)

- The Not-Recently-Used Page Replacement Algorithm
  - Makes use of the referenced and modified bits
  - When a process starts all its page entries are marked as not in memory
  - 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
  - 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

Virtual Memory: Page Replacement (NRU)

- 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 (allows us to recognise which pages have been recently referenced)

Virtual Memory: Page Replacement (NRU)

- 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 NRU algorithm removes a page at random from the lowest numbered class that has entries in it
- Not optimal algorithm, NRU often provides adequate performance and is easy to understand and implement

Virtual Memory: Page Replacement (FIFO)

- **The First-In, First-Out (FIFO) Page Replacement Algorithm**
  - Maintains 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
  - Simple to understand and implement but does not lead to good performance as a heavily used page is just as likely to be evicted as a lightly used page

Virtual Memory: Page Replacement (SC)

- **The Second Chance Page Replacement Algorithm**
  - Modification of the FIFO algorithm
  - When a page fault occurs if the page at the front of the linked list has not been referenced 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
- In the worst case, SC, operates the same as FIFO

Virtual Memory: Page Replacement (CP)

- **The Clock Page Replacement Algorithm**
  - The clock page (CP) algorithm differs from SC only in its implementation
  - SC suffers in the amount of time it has to devote to the maintenance of the linked list
  - 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

Virtual Memory: Page Replacement (LRU)

- **The Least Recently Used (LRU) Page Replacement Algorithm**
  - Approximate an optimal 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
  - 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 it is not cheap as we need to maintain a linked list of pages which are sorted in the order in which they have been used

Virtual Memory: Page Replacement (LRU)

- We can implement the algorithm in hardware
- The hardware is equipped with a counter (typically 64 bits). After each instruction the counter is incremented
- 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

Virtual Memory: Page Replacement (LRU)

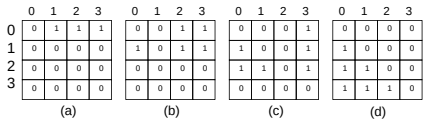
- Another hardware implementation of the LRU algorithm is given below
- If we have *n* page table entries a matrix of *n* x *n* bits, initially all zero, is maintained
- When a page frame, *k*, is referenced then all the bits of the *k* row are set to one and all the bits of the *k* column are set to 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

Virtual Memory: Page Replacement (LRU)

- If we have four page frames and access them as follows
  0 1 2 3 2 1 0 3 2 3
  the algorithm operates as follows

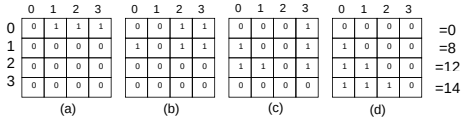
Virtual Memory: Page Replacement (LRU)

0 1 2 3 2 1 0 3 2 3



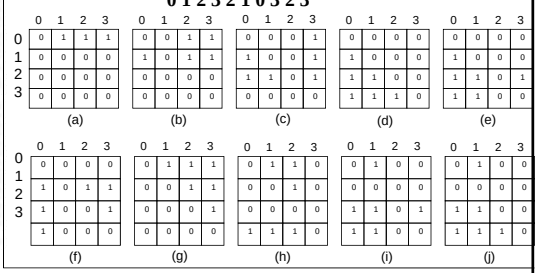
Virtual Memory: Page Replacement (LRU)

0 1 2 3 2 1 0 3 2 3



Virtual Memory: Page Replacement (LRU)

0 1 2 3 2 1 0 3 2 3



Virtual Memory: Page Replacement (LRU)

LRU in Software

- We cannot, as OS writers, implement LRU in hardware if the hardware does not provide the facilities
- We can implement a similar algorithm in software
- Not Frequently Used – NFU 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 to the counter
- When a page fault occurs the page with the lowest counter is selected for replacement.

Virtual Memory: Page Replacement (NFU)

- Problem with NFU is that it never forgets anything
- To alleviate this problem we can make a modification to NFU so that it closely simulates NRU
- The counters are shifted right one bit before the R bit is added.
- The R bit is added to the leftmost bit rather than the rightmost bit.
- This implements a system of aging.
- When a page fault occurs, the counter with the lowest value is removed

Virtual Memory: Page Replacement (NFU)

- Problem with NFU is that it never forgets anything
- To alleviate this problem we can make a modification to NFU so that it closely simulates NRU
- The counters are shifted right one bit before the R bit is added.
- The R bit is added to the leftmost bit rather than the rightmost bit.
- This implements a system of aging.
- When a page fault occurs, the counter with the lowest value is removed

Page	*R* bits for pages 0-5 **Clock Tick 0** 101011	*R* bits for pages 0-5 **Clock Tick 1** 110010	*R* bits for pages 0-5 **Clock Tick 2** 110101	*R* bits for pages 0-5 **Clock Tick 3** 100010	*R* bits for pages 0-5 **Clock Tick 4** 011000
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)

Demand Paging

- 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
- After a period of time the process should start to find that most of its pages are in memory
- Known as demand paging as pages are brought into memory on demand

Working Set

- The reason that page faults decrease (and then stabilise) is because processes normally exhibit a locality of reference
- 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
- If the entire working set is in memory then no page faults will occur
- Only when the process moves onto its next phase will page faults begin to occur again
- 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 and the process is said to be thrashing

Pre-Paging / Working Set

- In a system that allows many processes to run at the same time 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?
- Or do we move all its working set into memory so that it can continue with minimal page faults?
- The second option is to be preferred
- We would like to avoid a process, every time it is restarted, raising page faults

Pre-Paging / Working Set

- The paging system has to keep track of a processes' working set so that it can be loaded into memory before it is restarted.
- The approach is called the working set model (or pre-paging). Its aim, as we have stated, is to avoid page faults being raised
- 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 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 every page is full 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

Memory Management

Paging Daemons

- 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 can ensure pages are clean