

CS 534 OS Memory Management

(Paraphrased from Modern Operating Systems, Tanenbaum, Prentice-Hall)

Memory Management without Swapping or Paging

Monoprogramming without Swapping or Paging

- The **simplest** memory management scheme is to have only one process in memory at a time and allow that process to use all of the available memory
- This approach was common until about 1960
- One problem that arises with this solution is that requires that every process must contain or have access to device drivers for each I/O device they use
- This is the strategy that was used by the IBM PC

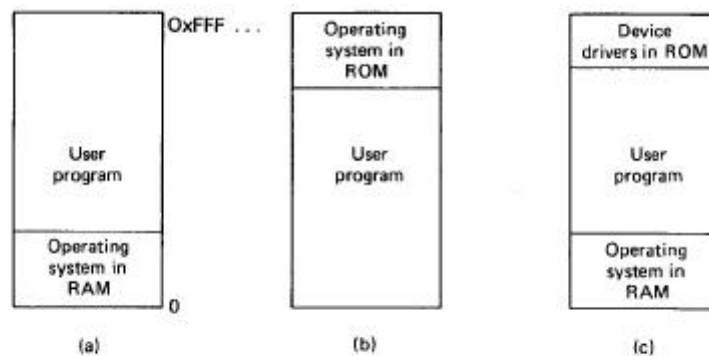


Fig. 3-1. Three ways of organizing memory with an operating system and one user process.

Multiprogramming with Fixed Partitions

- The question arises, **how do we organize memory so that more than one process can be in memory at once?**

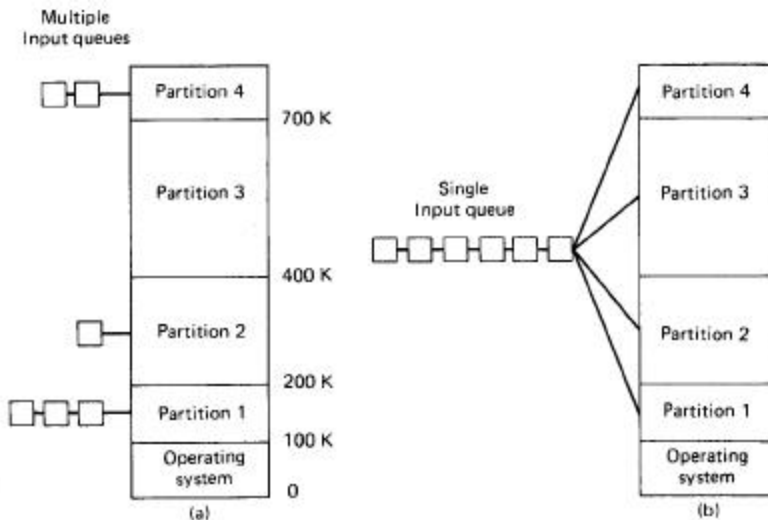


Fig. 3-4. (a) Fixed memory partitions with separate input queues for each partition.
(b) Fixed memory partitions with a single input queue.

- Memory can be divided into n (possibly unequal) partitions
- A possible solution is that each partition can have its own input queue
- If the partitions are a fixed but unequal size, a job can be placed in the input queue for the smallest partition large enough to hold that job
- Sorting the incoming jobs into separate queues has problems dealing with the smaller partitions filling before the larger ones so the larger partitions remain empty while data is waiting to get into the smaller partitions
- Another issue is that any space in a partition that is not used by a job cannot be used by another job until the current job is complete
- Yet another strategy is to have a single input queue feeding all partitions
- When a partition becomes free, the job closest to the front of the queue that will fit in the partition will be placed there
- The black cloud of wasted space still looms large
- It is possible to search the entire input queue so that when a partition becomes free, the largest job that will fit in the partition will be selected
- This algorithm tends to discriminate against smaller jobs

- Small jobs tend to be interactive jobs, and it is inappropriate to make a user wait
- A solution to this problem is to always have a small partition available for interactive jobs
- Another possible solution is to have a rule saying that a job cannot be skipped over in favor of another job (for whatever reason) more than ***k*** times
- Each time a job is skipped over, it acquires a “**point**”...when the job has acquired ***k* points**, it cannot be skipped over again

Relocation and Protection

Relocation

- It is important to note that different jobs will run at different addresses
- The job of a program called the linker is to combine the main program, user-written procedures and library procedures into a single address space
- The linker must know at what address the program will begin in memory

Example of the Relocation Problem

- Assume that first instruction is a call to a procedure at relative address 100 within the binary file produced by the linker
- If this program is loaded into **partition 1**, the instruction will jump to **absolute address 100, which is where the operating system resides**
- What is required is a call to **100K+100**
- If the program is loaded in **partition 2**, a call to **200K+100 is needed**

Protection

- A malicious program can construct a new instruction and jump to it
- When dealing with programs that use absolute addressing instead of relative addressing using CPU registers, there is no way to stop a program from building an instruction that can read or write anything in memory
- A way around this problem is to use a set of protection codes assigned to each block of memory
- Every process has a Process Status Word (PSW), so it could contain a corresponding protection code inside the PSW

- The OS can then trap any attempt by a running process to use memory where the code in its PSW did not match the code in a memory block
- A solution to both of these problems at once is to use a special set of registers called the base and limit registers
- When a process is scheduled, the base register will solve the relocation problem by being loaded with the starting address of its memory partition.
- Every memory address generated automatically will have the value in the base register added to it
- The limit register is loaded with the length of the memory partition making certain that the running process will use no addresses beyond that point

Swapping

- Swapping is moving processes from main memory to disk, or from disk back to main memory

Multiprogramming with Variable Partitions

- Fixed partitions present a problem when memory is scarce because a lot of memory is wasted by programs that are much smaller than the partition
- A system of variable partitions is more useful
- Variable partitions can change in size and number as load varies throughout the day

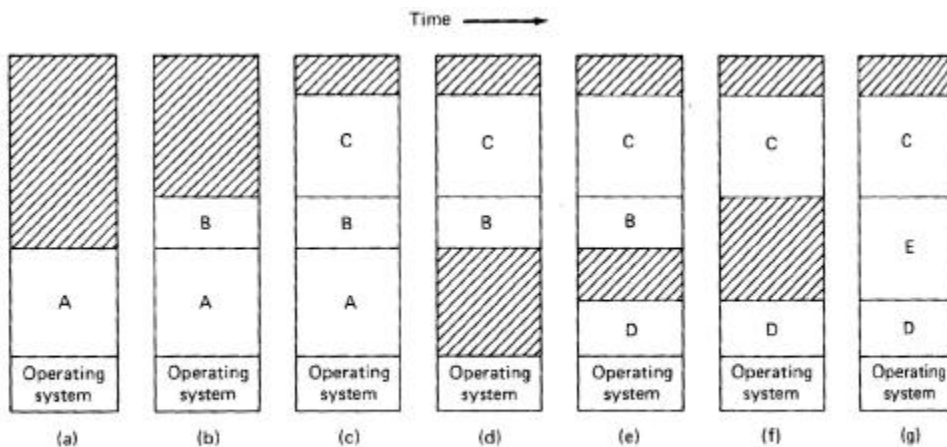


Fig. 3-5. Memory allocation changes as processes come into memory and leave it. The shaded regions are unused memory.

- **With this flexibility comes a price: complex operations dealing with tracking, allocation, and deallocation**

- Sometimes processes are allowed by programming languages to “grow” in memory (a process’s data segment for example can grow by dynamically allocating memory from a heap)
- Problems can occur when programs grow in memory
- If the process is adjacent to another process, the growing process will either have to be moved to a hole in memory large enough to accommodate it, or one or more processes will have to be swapped out to make a hole large enough for the growing process to fit
- If a process cannot grow in memory and the swap area on disk is full, the processes will have to wait or be killed
- Generally speaking, there are three ways to track memory usage:
 1. **Bit Maps**
 2. **Lists**
 3. **Buddy Systems**

Memory Management with Bit Maps

- With bit maps, **memory is divided into allocation units** (size of these units vary by the OS)
- Corresponding to each allocation unit is a **bit in the bit map**
- If the unit is **available**, the bit map contains a **zero**
- If the unit is **not available**, the bit map contains a **one**
- In systems with a **small, fixed amount of memory**, bit maps are a reasonable option
- The **main problem** with the bit map system is that when the decision has been made to bring a ***k unit*** process into memory, the memory manager has to search the map for a run of ***k consecutive zeros in the map***
- **Searching** a bit map for a run of a given length is a **slow operation**
- In real life, bit maps are **not often used**

Memory Management with Linked Lists

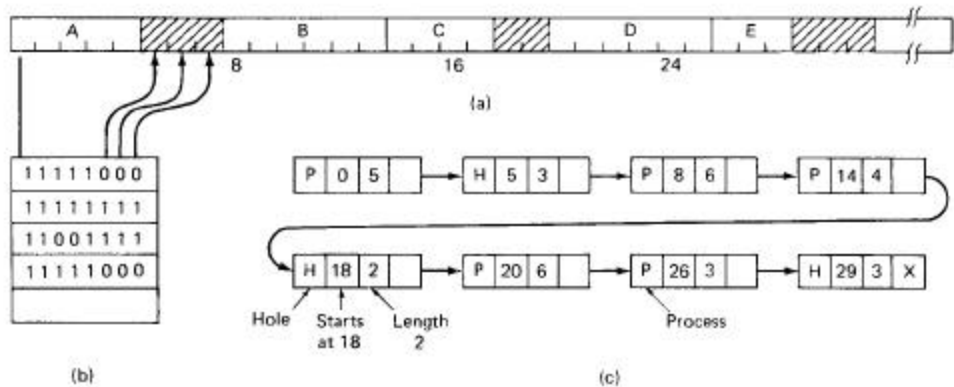


Fig. 3-7. (a) A part of memory with five processes and 3 holes. The tick marks show the memory allocation units. The shaded regions (0 in the bit map) are free. (b) The corresponding bit map. (c) The same information as a linked list.

- Memory tracking can be done by using a system of **linked lists**
- The list represents allocated and free memory segments
- Each entry represents a hole (H), a process (P), the address at which it starts, the length, and a pointer to the next entry
- Sometimes a double-linked list is used, because it can point forward and backward
- When processes and holes are kept in a list sorted by address (which they usually are), several algorithms can be used to allocate memory for newly swapped-in or newly created process
- **All of the algorithms assume that the memory manager knows how much memory to allocate**

First Fit

- The memory manager scans the list of segments until it finds a hole large enough.
- The hole is broken into two pieces (one for the process and one for unused memory) except if there is an exact fit.
- Fast algorithm...searches as little as possible. Creates larger holes on the average

Next Fit

- A variation on First Fit except that after it finds a hole, it keeps track of where it is.
- The next time a search is made, it begins where it left off instead of starting at the top.
- Slightly slower than first fit

Best Fit

- Searches the **entire list** for the hole that is closest in size to what is actually needed and takes the smallest hole needed.
- Creates tiny useless holes that fill memory.
- Slower than First Fit.

Worst Fit

- **Always take the largest possible hole** so that whatever is piece of memory broken off will be useful to another process.
- Not a very good algorithm.

Making the Four Algorithms Faster

- It is possible to make the four algorithms faster by **maintaining separate lists for processes and holes**, having them spend energy **inspecting holes** not processes
- The trade-off is the **complexity and the overhead** required to maintain two lists
- Keeping all holes in one or more lists sorted by hole size make allocation very fast
- Deallocation is very slow because because all hole lists must be searched to find the deallocated hole's neighbors so merging of space can take place

Memory Management with the Buddy System

- Developed by Knuth in 1973
- Takes advantage of the fact that computers use binary numbers for addressing in order to speed up the merging of adjacent holes when a process terminates or is swapped
- Initially, all memory is set to one block
- When a request for memory is made, the block is split into two blocks called buddies
- Further subdivisions are made until the smallest block that is a power of 2 in size larger enough to accommodate the memory request is found

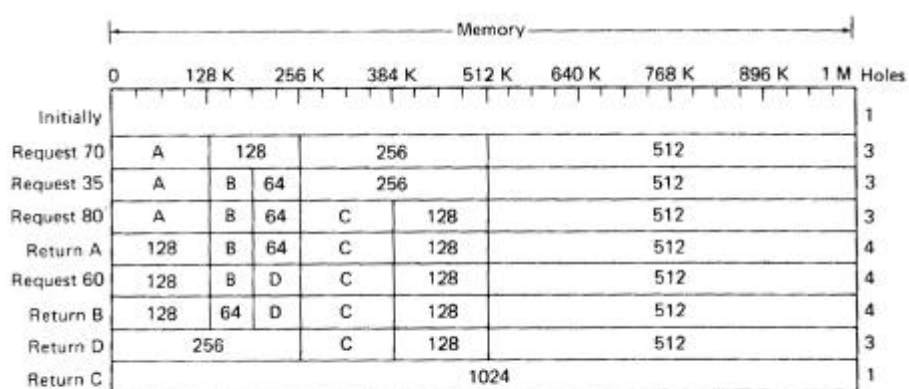


Fig. 3-9. The buddy system. The horizontal axis represents memory addresses. The numbers are the sizes of unallocated blocks of memory in K. The letters represent allocated blocks of memory.

- The advantage of the Buddy System is that when a block of size 2^k is freed, the memory manager **only has to search the list of holes of size 2^k** to see if a merge is possible
- This results in the **Buddy System** being very fast
- **Unfortunately it is very inefficient in terms of memory utilization**
- The reason is that **every** request for memory has to be **rounded UP** to the next 2^k size block that will be able to handle the request
- A **35KB process** must be allocated **64KB of space**
- The remaining 29KB **is wasted**

- This is called **internal fragmentation** because the wasted memory is internal to the allocated segments
- In other algorithms there are holes between segments, but no wasted space within the segments
- This is called **external fragmentation** or **checkerboarding**

Allocation of Swap Space

- The algorithms for managing swap space are essentially the same for managing main memory
- The primary difference is that when disk space is required for a swap, it is allocated in whatever units are used for disk blocks
- Assume a 13.5KB process, and disk blocks of size 1KB
- Then 14 blocks will have to be allocated to accommodate the process

Virtual Memory

- Years ago programs that were too large to fit into memory were broken into smaller pieces called overlays
- Overlays were typically kept on disk and were paged in and out of memory as necessary
- Overlays had to be created by programmers
- Eventually, the idea of Virtual Memory came into being, and the computer did all of the work that the programmer's did (creating overlays) automatically
- The idea is that the combined size of the program, data and stack could exceed the size of physical memory
- The OS would keep those parts of the program that were **required at the moment to be kept in main memory**, and the **rest kept on disk**
- Multiprogramming and Virtual memory work well together
- While a process is waiting for a part of itself to be swapped in, it is waiting for I/O and cannot run
- The CPU is turned over to another process

Paging

- On any computer a set of memory addresses exists that programs can produce

Given the instruction:

MOV REG,1000

- The **contents** of memory location 1000 are being moved to **REG** (or vice versa)
- Addresses can be generated using **indexing, base registers, segment registers, or other means**
- Program-generated addresses are called **virtual addresses** and constitute the **virtual address space**
- On a computer **without** virtual memory, the virtual address is placed **directly on the memory bus and the contents of that address will be read or written**
- When virtual memory is used, addresses go to the **Memory Management Unit (MMU)**
- The **MMU** is a **chip or chipset that maps the virtual memory addresses on the physical memory addresses**

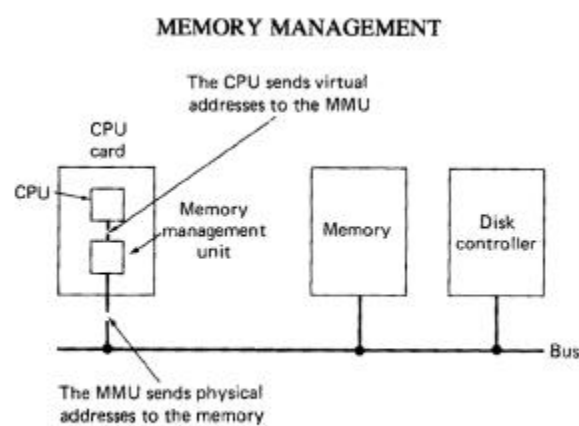


Fig. 3-10. The position and function of the MMU.

VIRTUAL MEMORY

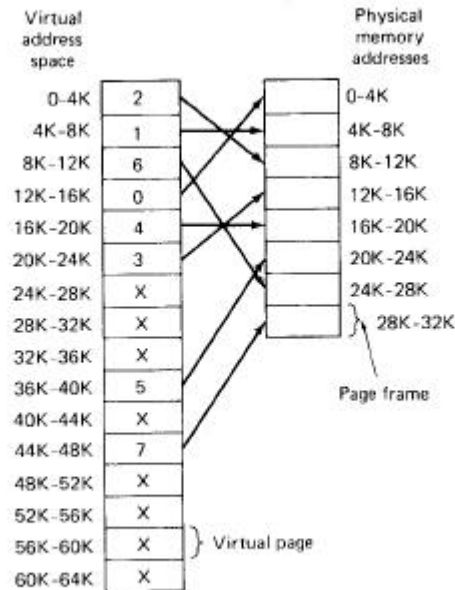


Fig. 3-11. The relation between virtual addresses and physical memory addresses is given by the page table.

- Note that this computer can **generate virtual addresses from 0-64K**
- Also note that this computer only has **32K of physical memory**
- While a 64K program can be run on this computer, **it cannot be loaded entirely in memory at one time**
- A complete 64K image of the program **must exist on disk so pieces can be brought in as required**
- The virtual address space is divided into sections called **pages**
- The corresponding units in physical memory are called **page frames**
- **The frame page and the pages are always the same size**
- Transfers between memory and disk are **always in units of a page**

VIRTUAL MEMORY

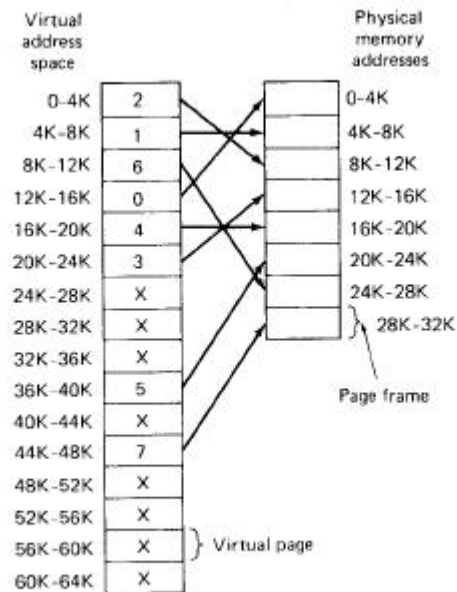


Fig. 3-11. The relation between virtual addresses and physical memory addresses is given by the page table.

Assume the instruction:

MOV REG,0

- The **virtual address 0** is sent to the MMU
- The MMU sees that this **address falls in page 0 (0-4K)**
- The MMU has **mapped** these addresses to **page frame 2 (8192-12287)**

Similarly, given the instruction:

MOV REG, 8192

- The address 8192 can be found in virtual page 2, and virtual page 2 is mapped to page frame 6 (physical addresses 24576-28761)

What happens when a program tries to use an unmapped page?

Given the instruction:

MOV REG, 32780

- According to our virtual page map, this page is **unmapped**
- The CPU causes the OS to generate an error trap that is called a **page fault**

Page Tables

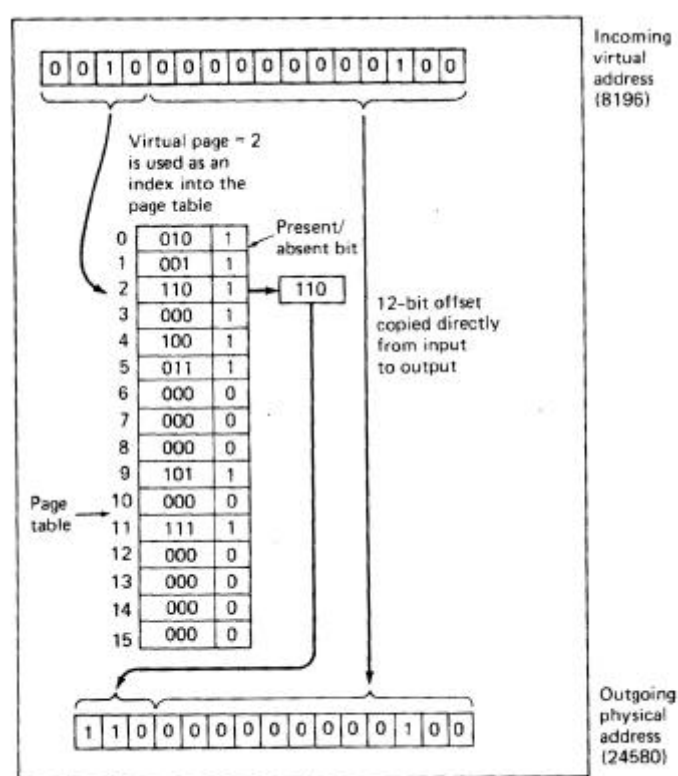


Fig. 3-12. The internal operation of the MMU with 16 4K pages.

- If the absent/present bit is **zero**, a page fault will be generated
- the **virtual page number** is split into a **virtual page number** (high-order bits) and an **offset** (low-order bits)
- The virtual page number is **used as an index into the page table** to locate the entry for that virtual page
- The **page frame entry** (if any) is found in the page table

Page Tables (continued)

- The page frame number is attached to the high-order end of the offset
- This replaces the virtual page number, forming the physical address
- **There are two major issues:**
 1. **Page tables can be extremely large**
 - Consider a virtual address size of 32-bits
 - With a 4KB page size, a 32-bit address space has > 1 million pages
 - With 1 million pages in the virtual address space, the page table must have 1 million entries
 - REMEMBER: each process needs its own page table!
 2. **The mapping must be very fast**
 - Virtual-to-physical mapping must be done on **each memory reference**
 - A typical instruction has an instruction word and often a memory reference
 - Therefore, it is possible that one, two, or more references to the page table will be made for **each instruction**

Hardware Solutions to Page Tables

- The simplest conceptual design is to have a **single page** consisting of an **array of fast hardware registers**, with **one entry for each virtual page, indexed by virtual page number**
- The **advantage** of this solution is that it is **straight forward** and requires **no memory references during mapping**
- **Disadvantages** are that it is **potentially expensive** (performance-wise) if the page table is large...**having to load the page table with each context switch can be time consuming**

Page Tables (continued)

- At the other extreme, a page table can be **kept entirely in memory**
- The **advantage** is that the hardware needs just a **single register that points to the start of the page table**
- This means that when a context switch is made, the **memory map can be changed by reloading one register**
- A **MAJOR disadvantage** is that **one or more memory references are required to read page table entries during the execution of each instruction**
- For the above reason, this approach is **rarely used in pure form**
- There are, as a result, **two, three, and four-level solutions** to the hardware implementation of page tables

Page Replacement Algorithms

- When a page fault occurs, the OS must choose a page to remove from memory so that space can be made for the page to be brought in
- If the page to be removed has been **modified** while in memory, it must be **written back to disk to bring the disk copy current**
- If the page has **NOT been modified** (maybe a page of program text), the disk copy is already up to date so a **rewrite to disk is not required**
- In this case, the **page to be read in merely overwrites the existing page**
- While possible to randomly select a page to evict at each page fault, **system performance is improved if a page is chosen that is not frequently used**
- There has been much theoretical work done on the subject of page replacement algorithms

The Optimal Page Replacement Algorithm

- This is the **best possible algorithm**
- This algorithm is **impossible to implement in the real world**
- This algorithm says that the page with the HIGHEST label will be removed
- If one page will not be used for 8 million instructions, and another page will not be used for 4 million instructions, removing the page that is 8 million instructions away is optimal
- Like people, computers like to put off unpleasant tasks for as long as they can
- The problem is that at the time of the page fault, the OS has no way of knowing when each of pages will be referenced next
- A simulation of a program run can be made and logs of all page references kept
- Then, on a SECOND RUN of the same exact program, it might be possible to implement optimal page replacement based on information gained from the first run
- This situation makes this algorithm unfeasible

The Not-Recently-Used Page Replacement Algorithm

- A real world algorithm
- Statistics are kept by the OS as to which pages are being used, and which are not
- Computers with virtual memory **have two status bits associated with each page**
- ***R*** is set when a page is **referenced** (read or written)
- ***M*** is set when the page is **modified**
- These status bits are kept in the page table and are **updated with every memory reference by hardware**
- Once a bit is set to one, it stays one until the OS changes it back to zero

Page Replacement Algorithms (continued)

The algorithm is as follows:

- When a process is started, **R** and **M** are set to zero by the OS
- On each clock interrupt the **R** bit is cleared
- This makes it possible to distinguish between pages that have been **referenced recently**, and those that have not
- When a page fault occurs, the OS inspects all pages and divides them into four categories based on the status of the **R** and **M** bits:
 1. **Class 0: not referenced, not modified**
 2. **Class 1: not referenced, modified**
 3. **Class 2: referenced, not modified**
 4. **Class 3: referenced, modified**
- Class 1 pages seem impossible at first glance...
- Class 1 pages occur when a **Class 3 page has had its R bit cleared** by a clock interrupt
- Clock interrupts **do not change the M bit**...this info is needed to determine whether a page has to be written to disk
- The NRU removes a page **at random** from the **lowest numbered non-empty class**

The First-In-First-Out Page Replacement Algorithm

- **FIFO** is implemented as a list
- The OS maintains a list of all pages
- The **oldest** is the page at the **head**
- The **newest** is the page at the **tail**
- On a page fault, the **page at the head of the list** **kicked off the list** and a **new page is added to the end of the list**
- Because of the **inability to discriminate between the priority of heavily used types of pages**, this algorithm is rarely used in pure form

The Second Chance Page Replacement Algorithm

- A **modified** version of **FIFO**
- Inspects the **R** bit of the **oldest page**
- If **zero**, the page is **old and unused** so it is **replaced immediately**
- If **one**, the **bit is cleared** and the **page is moved to the end of the list**
- The **load time of the page** is **updated** as though it had **just arrived in memory**
- **Ineffecient** because pages are **constantly being shuffled around the list**

The Clock Page Replacement Algorithm

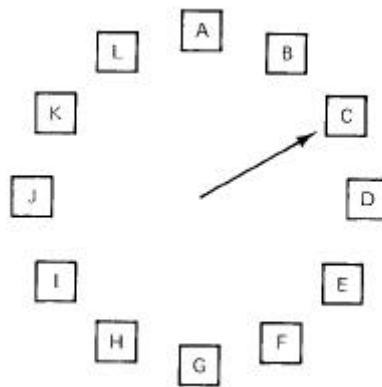


Fig. 3-25. The clock page replacement algorithm

- All pages are placed in a circular queue like a clock
- A pointer points to the pages
- When a page fault occurs, the **R** bit of the page **currently being pointed to** is inspected
- If it is **zero**, the **page is replaced (in its place in the queue) by the new page and the pointer is advanced**
- If it is **one**, the bit is cleared, and the pointer **skips to the next position and repeats the previous procedure until a page that can be replaced is found (R = 0)**

The Least-Recently-Used (LRU) Page Replacement Algorithm

- It can be shown that pages that have been heavily used in the last few instructions will probably be heavily used again in the next few instructions
- The converse can also be shown
- Given these observations, the LRU algorithm just throws out the page that has gone unused for the longest time when a page fault occurs
- This algorithm can be implemented as either a linked list or an array
- The linked list contains all pages in memory
- The most recently used page is at the head

The First-In-First-Out Page Replacement Algorithm (continued)

- The least recently used page is at the tail
- Problems with the implementation are:
 1. **The list has to be updated with every memory reference**
 2. **Finding a page and moving it to the front of the list is very time consuming**
- These problems exist even when implemented in hardware

Implementation Issues

Paging Daemons

- Paging works best when there are always plenty of free page frames that can be claimed as page faults occur
- Remember that if all page frames are full or if pages have been modified, the pages must be written to disk before they can be evicted and a new page brought in
- A background process called a **Paging Daemon** makes sure that **free page frames** are plentiful at all times
- The Paging Daemon sleeps most of the time, but awakens periodically to inspect the state of memory
- If there are too few free page frames, the daemon selects pages to evict based on the page replacement algorithm
- If a page has been modified since being loaded, it is written back to disk