Resource Management

Fred Long

based on slides originally written by Mike Tedd
and modified by Edel Sherratt

Introduction

- A resource is something a process needs to do its job, e.g.:
  - CPU
  - memory
  - access to files
  - peripheral devices
  - semaphores, locks, etc.

Allocating Resources

- The OS has the responsibility of allocating resources to processes
- It must take account of:
  - process priority
  - fairness
  - the possibility of deadlocks

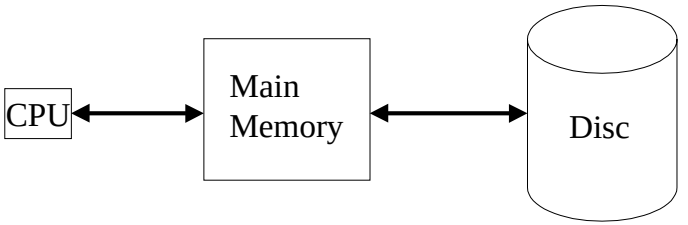
Allocating resources (2)

- Note that some resources (e.g., CPU, memory) are “pre-emptible”, i.e., can be taken away from one process temporarily
- Techniques like spooling (Simultaneous Peripheral Operation On-Line) ease resource management by using the disc as a large buffer

Memory Management

- When multi-programming, we want to achieve effects like:
  - holding multiple programs in main memory
  - protecting processes so they cannot corrupt each other's code and data
  - sharing code and data when appropriate
  - running programs larger than the main memory

Swapping



Programs, and parts of programs, move to and from disc. This is called *swapping*.

Paging

- The program is split into *fixed-size* chunks called *pages*
- The pages are not usually related to the logical structure of the program
- The store is considered to be split into pieces, each of which can hold a page.
- These are called *page frames*

Paging (2)

- The program-generated virtual address is split into two parts:

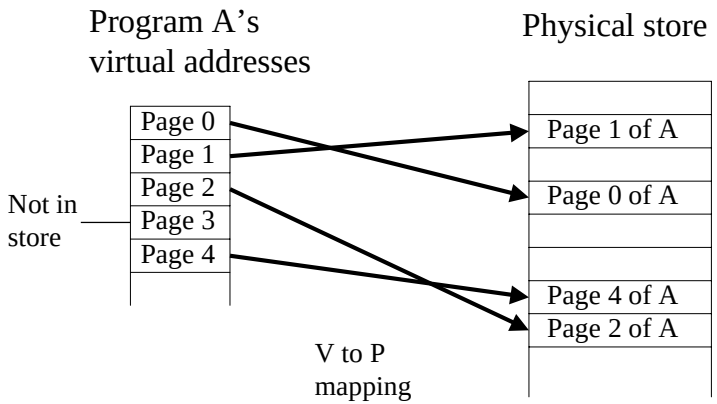
Page no.	Offset

Example: If the page size is 1024, address 2058 (10 0000001010 in binary) means word 10 of page 2.

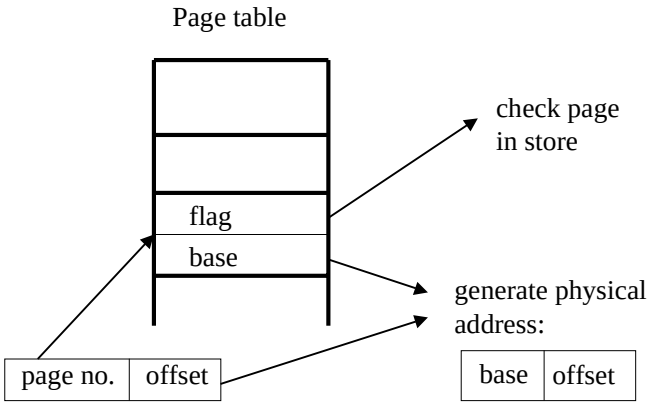
Paging (3)

- There is a *page table* with one entry for each page
- Each page table entry contains:
  - a flag to say if the page is in store
  - its start address, i.e., the page frame number
- An access consists of:
  - get page number and offset
  - get page table [page number]
  - check if in store, fetch if not
  - physical address = base + offset (by concatenation)

Page Mapping



Page Mapping Mechanism



History of Virtual Storage

- Invented in Manchester
- Used on the Atlas, c. 1960
- Burroughs used segments from early 60s
- IBM's first computer with paging was the 360/67
- In 1970, IBM added paging to the 370 series: "IBM announce tomorrow"

Advantages of Paging

- Avoids complex swapping for the OS
- Largely solves fragmentation
- Saves users the complication of overlays
- Application programs are no longer system tailored
- Programs can dynamically change size
- Sharing of code and data can be arranged

Advantages of Paging (2)

- Some types of data file can be addressed as though in main store (virtual data)
- Can run programs bigger than the main store
- Hence removes one barrier to compatibility between machines
- Virtual machine simulation, e.g., VM/370

Problems Associated with Paging

- When is a page fetched in?
  - the cost of fetching is significant:
    - backing store traffic
    - administration
    - may need to discard another page to make room
    - poor response of affected process
  - the cost of not having a page in when needed is relatively small:
    - run another process while this one waits

Problems (2)

- So, typically, we use *demand paging*:
  - the page is fetched only when the process actually tries to use it
- Where is an incoming page put?
  - if there is free space, any empty page frame will do; if not, a page must be thrown out, hence:

Problems (3)

- What is swapped out to make space?
  - note, if a page chosen for discard has not been written to, there is no need to swap it out as there is already a copy on backing store
    - the page might be code or read-only data
    - the page table entry often has a bit set by hardware whenever the page is written to, called a "dirty bit"
  - the process of choosing which page to swap out is called a *discard algorithm*

Discard Algorithms

- The problem is: given a set of pages in memory, one or more of which must be discarded, decide which
- All we have to go on is the *past* behaviour of the programs
- Form this, we are trying to predict which pages will be least valuable in the *future*

Optimum Algorithm

- Discard the page that will next be wanted at the latest time in the future
  (Ex. *Prove* this is the optimum strategy)
- This is not a practical algorithm!
- It is important in providing a yardstick for measuring practical algorithms

First In, First Out (FIFO)

- Discard the page that has been in longest
- This is the simplest algorithm to implement
- Set up a queue, add pages to the end of the queue as they are brought in, discard the one at the head of the queue
- This requires no hardware support

FIFO (2)

- It is intuitively reasonable that the page that has been in longest could well have fallen out of use
- On the other hand, there will be 'core' pages in fairly continuous use and FIFO will eventually discard them

Belady's Anomaly

- Surprisingly, with FIFO, having more page frames may *increase* the number of page faults!
- Suppose we need the following pages in the order shown: 1,2,3,4,1,2,5,1,2,3,4,5
  - if we have 3 page frames we get 9 page faults
  - if we have 4 page frames we get 10 page faults

Least Recently Used (LRU)

- Conceptually, we associate with each page the time of its last use and discard the page that has not been used for the longest time
- LRU is expensive to operate but really quite good
- The problem is: how to implement it?

LRU Implementations

- Counter: Each page in the page table has a counter that is incremented periodically; when the page is used the count is cleared; discard the page with the largest count
- Stack: Maintain a stack of page numbers; when a page is used, move its number to the top; discard the page at the bottom of the stack (actually, a doubly linked list is better)

LRU Implementations (2)

- LRU really requires hardware support for updating the counters or stack
- If the data structures were updated by software, every memory reference would be slowed by a factor of about 10
- Hence, every process would be slowed by a factor of 10

Second Chance (or Clock)

- Each page has a *used* bit, initially unset but set when the page is used
- The FIFO queue becomes a circular queue
- When a discard is required, we work through the queue looking for an unset bit, and unsetting the bits as we go
- The page discarded is the first one found with its used bit unset

Thrashing

- Paging is very useful, but not a miracle cure for insufficient memory
- Real programs have a core of highly used pages; if you try to run them in less real store then they will page fault at an intolerable rate — this is called *thrashing*

Thrashing (2)

- Note the potential feedback loop:
  - thrashing starts to happen
  - existing jobs take longer to finish
  - more jobs piles up
  - worse thrashing
- Some operating systems were total failures because of thrashing, e.g., IBM's TSS/360

Working Sets

- Denning coined the phrase *working set* of a program: the minimum main store space within which it is advisable to run the program
- This varies from program to program, and from moment to moment
- A *poor* rule of thumb is 1:3

Working Sets (2)

- The working set depends on a parameter, Δ
- The working set at a time t, WS(t), is the set of the Δ most recent page references
- For example, with $\Delta = 10$:
  - references at time t_1: ...,2,6,1,5,7,7,7,7,5,1
    WS(t_1) = {1,2,5,6,7}
  - references at time t_2: ...,3,4,4,4,3,4,3,4,4,4
    WS(t_2) = {3,4}

Working Sets (3)

- At a given time, the size of the working set for process i is denoted by WSS(i)
- Then, the total demand for page frames is Σ WSS(i)
- The OS monitors the working set of each process and allocates enough frames to meet its working set size

Working Sets (4)

- If there are enough spare frames, a new process can be initiated
- If the total frame demand exceeds the number of frames available, a process is suspended and its frames written out and then reallocated to other processes
- This strategy prevents thrashing while keeping as much multi-programming as possible

Working Sets (5)

- The difficulty with the working set model is in keeping track of the working set
- The cost is high
- Also, the model is basing future performance on past behaviour

Page Fault Frequency

- The specific problem is how to prevent thrashing
- Thrashing is caused when there is a high page fault rate
- So, we could try to control the page fault rate more directly

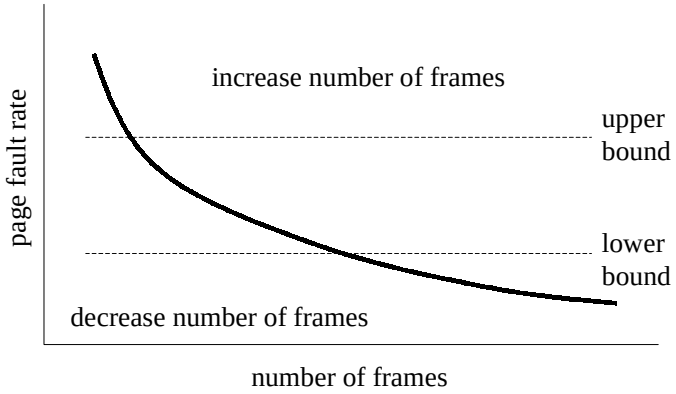
Page Fault Frequency (2)

- When the page fault rate is too high we know that the process needs more frames
- Similarly, if the page fault rate is low, then the process has more frames than it really needs
- We establish acceptable upper and lower bounds for the page fault rate

Page Fault Frequency (3)

- If the actual page fault rate exceeds the upper bound, another frame is allocated to the process
- If the actual page fault rate falls below the lower bound, a frame is removed from the process
- Thus, the page fault rate is measured and controlled directly

Page Fault Frequency (4)



Page Size

- Small pages means more backing store transfers, more administration, larger tables
- Big pages means more (internal) fragmentation
- The typical page size in use today is 4KB
- With a page table entry of 4 bytes, the page tables can get large, so the tables themselves are usually paged!

Associative Store

- We want to avoid multiple store accesses (which are potentially needed to access page tables)
- There are special forms of store that can be accessed, not by quoting addresses, but by quoting contents
- These are called *associative* store, *content addressable* store, or *cache*

Associative Store (2)

Key	Data
7	
13	X
4	
9	Y
12	
8	

- If this store is referenced with key “13”, the answer will be “X”. With key “9”, answer is “Y”. With key “5”, answer will be “Not here”
- Such store can be *fast*

Associative Store and Paging

- The key is the page number, the data is the (page base, in-store flag) pair
- A small associative store will give good performance, e.g., 16 entries may well give a 99% hit rate. Only in 1% of page table look-ups do you need to go to the page table
- Need organisation to ensure that the most valuable information is in the associative store — hardware, of course

Associative Store and Paging (2)

- Need discard algorithms to manage an associative store — usually FIFO
- The real table must always be updated if an entry changes
- One must ensure that the two copies are the same

Cache Stores for Data Access

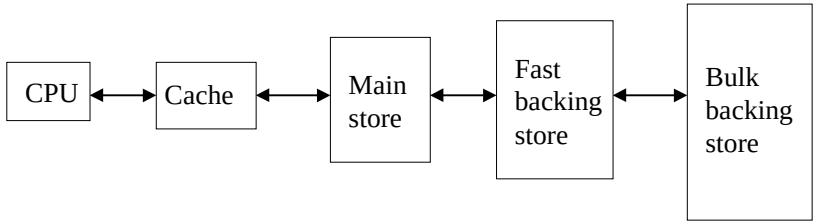
- An associative store can also be used to avoid going to the physical store for frequently used data
- The key is the store address, the data is the contents

Cache Stores for Data Access (2)

- Since programs reference individual data locations much more randomly than they reference larger units like pages, a much bigger associative store is needed
- E.g., 4K words of cache (80nsec access time) fronting 1M words of ordinary store (600nsec). With a 95% hit rate, the mean access time is 95%x80+5%x680 = 110nsec

Cache Stores for Data Access (3)

- All these tricks start to make big machines relatively poor at switching from one program to another
- Big computers have a *multi-level store hierarchy*

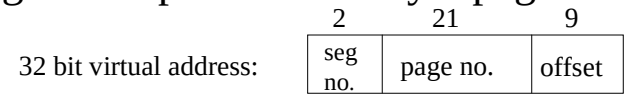


Real Systems

- Most operating systems adopt strategies more complex than the pure algorithms covered above
- Some OSs try to identify the working set size for each program, either expecting the user or operator to specify it, or learning from experience
- They run a program only if they can give it at least this much space

Paging on VAX

- VAX/VMS is basically FIFO, but pages are discarded to a pool, waiting to be actually swapped out
- If a mistake is made, the page concerned may be retrieved from the pool
- VAX hardware provides each process with 3 segments split into 512 byte pages



Paging on VAX (2)

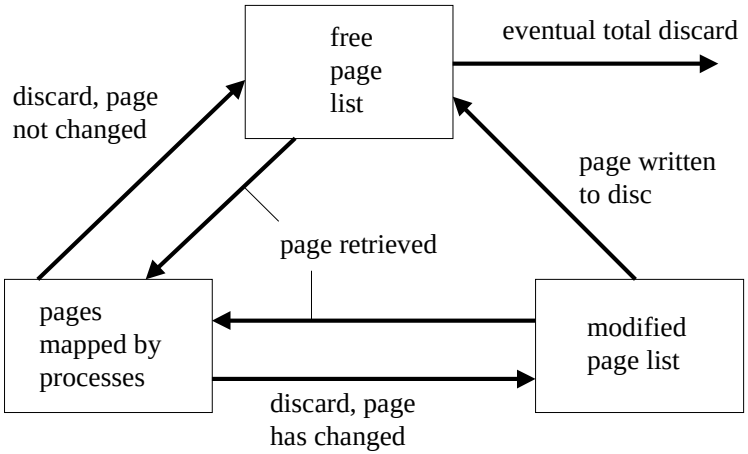
- Each page table entry contains:

bits		
1	valid	(in store *and* mapped)
4	protection	(access control)
1	modify	(set when pge changed)
5	reserved	
21	page frame number	

Paging on VAX (3)

- The page tables are large, so are themselves paged
- No “used” bit, so LRU would be too expensive
- Each process has a maximum allocation of pages
- Whole processes swapped to limit total demand
- On a page fault, if a process is at its limit, one of its own pages is discarded from the set mapped for direct use, using FIFO
- In addition, there are two pools of pages not actually mapped

Paging on VAX (4)



Paging on Berkeley Unix

- Unix has lots of processes, which are often connected by pipes. So, it is less sensible to page a process against itself
- BSD uses 1024 byte “pages”. Each “page” is actually two hardware pages

Paging on Berkeley Unix (2)

- The discard algorithm is Second Chance, and is made to free lists, like VMS
- Note that Second Chance here needs a simulation of the used bit
- There is a certain amount of *pre-paging* — several pages are fetched to the free list when the first is wanted

Paging on Linux

- Linux uses a modified version of the second chance (or clock or NRU) algorithm
- A multiple-pass clock is used and every page has an *age* that is adjusted on each pass of the clock
- Frequently used pages will attain a high age value, while the age of infrequently used pages will drop towards zero at each pass
- Discards are chosen on a LFU policy

Paging on Windows NT

- NT uses the working set model, and sets a minimum working set size for each process
- NT uses a per-process FIFO discard algorithm to take pages from processes not at their minimum working set size
- For processes at their minimum working set size, page faulting is monitored so that the working set size may be adjusted

Paging on Windows NT (2)

- Every process is initially assigned a working set size of 30 pages
- Periodically, NT tests the working set size by stealing a page from the process
- If the process continues without page faulting, its working set size is reduced by 1

Paging on Windows NT (3)

- On NT, when a page fault occurs, not only is the required page brought into memory but also a few adjacent pages
- This is because, when a page is used, it is likely that adjacent pages will also be used in the near future (the *locality* property of the process)
- This results in fewer page faults