Comp 104: Operating Systems Concepts

Process Scheduling

Today

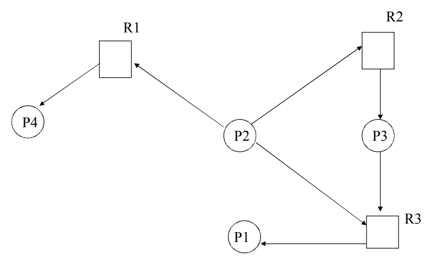
- Deadlock
  - Wait-for graphs
  - Detection and recovery
- Process scheduling
- Scheduling algorithms
  - First-come, first-served (FCFS)
  - Shortest Job First (SJF)

Wait-For Graph

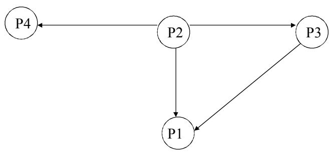
- Precise definition:
- An edge from P_i to P_j implies that process P_i is waiting for process P_j to release a resource that P_i needs
- An edge $P_i \rightarrow P_j$ exists in a wait-for graph if and only if the corresponding resource-allocation graph contains two edges $P_i \rightarrow R_q$ and $R_q \rightarrow P_j$ for some resource R_q
- Deadlock is present if there is a cycle in the wait-for graph

Exercise

- Construct the wait-for graph that corresponds to the following resource allocation graph and say whether or not there is deadlock:



Corresponding Wait-For Graph



5

Wait-For Graph

- In order to be able to effectively detect deadlock, the system must maintain the wait-for graph and run an algorithm to search for cycles, at regular intervals
- Issues:
  - How often should the algorithm be invoked?
    - Costly to do for every request
    - May be better to do at regular intervals, or when CPU utilisation deteriorates too much
  - How many processes will be affected by the occurrence of deadlock?
    - One request may result in many cycles in the graph

6

Detection and Recovery

- Once the detection algorithm has identified a deadlock, a number of alternative strategies could be employed to recover from the situation
- Recovery could involve process termination
  - All involved
    - May be huge loss of computation
  - One at a time
    - Expensive: requires re-run of detection algorithm after each termination
- Recovery could involve preemption
  - Choice of victim
  - Rollback
  - Starvation

7

Scheduling

- In any multiprogramming situation, processes must be scheduled
- The long-term scheduler (job scheduler) decides which jobs to load into memory
  - must try to obtain a good job mix: compute-bound vs. I/O-bound
- The short-term scheduler (CPU/process scheduler) selects next job from ready queue
  - Determines: which process gets the CPU, when, and for how long; when processing should be interrupted
  - Various different algorithms can be used…

8

Scheduling

- The scheduling algorithms may be preemptive or non-preemptive
  - Non-preemptive scheduling: once CPU has been allocated to a process the process keeps it until it is released upon termination of the process or by switching to the ‘waiting’ state
- The dispatcher module gives control of the CPU to the process selected by the short-term scheduler
  - Invoked during every switch: needs to be fast
- CPU–I/O Burst Cycle: process execution consists of a cycle of CPU execution and I/O wait
- So what makes a good process scheduling policy?

9

Process Scheduling Policies

- Several (sometimes conflicting) criteria could be considered:
- Maximise throughput: run as many processes as possible in a given amount of time
- Minimise response time: minimise amount of time it takes from when a request was submitted until the first response is produced
- Minimise turnaround time: move entire processes in and out of the system quickly
- Minimise waiting time: minimise amount of time a process spends waiting in the ready queue

10

Process Scheduling Policies

- Maximise CPU efficiency: keep the CPU constantly busy, e.g. run CPU-bound, not I/O bound processes
- Ensure fairness: give every process an equal amount of CPU and I/O time, e.g. by not favouring any one, regardless of its characteristics
- Examining the above list, we can see that if the system favours one particular class of processes, then it adversely affects another, or does not make efficient use of its resources
- The final decision on the policy to use is left to the system designer who will determine which criteria are most important for the particular system in question

11

Process Scheduling Algorithms

- The short-term scheduler relies on algorithms that are based on a specific policy to allocate the CPU
- Process scheduling algorithms that have been widely used are:
  - First-come, first-served (FCFS)
  - Shortest job first (SJF)
  - Shortest remaining time first (SRTF)
  - Priority scheduling
  - Round robin (RR)
  - Multilevel queues

12

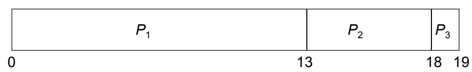
First-Come, First Served

- Simplest of the algorithms to implement and understand
  - Uses a First-In-First-Out (FIFO) queue
- Non-preemptive algorithm that deals with jobs according to their arrival time
  - The sooner a process arrives, the sooner it gets the CPU
  - Once a process has been allocated the CPU it keeps until released either by termination or I/O request
- When a new process enters the system its PCB is linked to the end of the 'ready' queue
- Process is removed from the front of the queue when the CPU becomes available, i.e. after having dealt with all the processes before it in the queue

13

Example

- Suppose we have three processes arriving in the following order:
  - P_1 with CPU burst of 13 milliseconds
  - P_2 with CPU burst of 5 milliseconds
  - P_3 with CPU burst of 1 millisecond
- Using the FCFS algorithm we can view the result as a Gantt chart:



14

First-Come, First Served

- Given the CPU burst times of the processes, we know what their individual wait times will be:
  - 0 milliseconds for P_1
  - 13 milliseconds for P_2
  - 18 milliseconds for P_3
- Thus, the average wait time will be (0 + 13 + 18)/3 = 10.3 milliseconds
- However, note that the average wait time will change if the processes arrived in the order P_2, P_3, P_1

15

Exercise

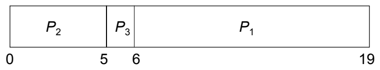
- What will the average wait time change to if the processes arrived in the order P_2, P_3, P_1?

16

Answer

- P_2 with CPU burst of 5 milliseconds
- P_3 with CPU burst of 1 millisecond
- P_1 with CPU burst of 13 milliseconds

- Represented as the following Gantt chart:



- Thus, the average wait time will be (0 + 5 + 6)/3 = 3.7 milliseconds

17

First-Come, First Served

- Advantages:
  – Very easy policy to implement

- Disadvantages:
  – The average wait time using a FCFS policy is generally not minimal and can vary substantially
  – Unacceptable for use in time-sharing systems as each user requires a share of the CPU at regular intervals
    - Processes cannot be allowed to keep the CPU for an extended length of time as this dramatically reduces system performance

18

Shortest Job First

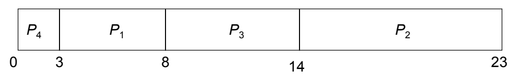
- Non-preemptive algorithm that deals with processes according to their CPU burst time
  – When the CPU becomes available it is assigned the next process that has the smallest burst time
  – If two processes have the same burst time, FCFS is used to determine which one gets the CPU

- Suppose we have four processes as follows:
  – P_1 with CPU burst of 5 milliseconds
  – P_2 with CPU burst of 9 milliseconds
  – P_3 with CPU burst of 6 milliseconds
  – P_4 with CPU burst of 3 milliseconds

- Using the SJF algorithm we can schedule the processes as viewed in the following Gantt chart…..

19

Example



- The wait times for each process are as follows:
  - 3 milliseconds for P_1
  - 14 milliseconds for P_2
  - 8 milliseconds for P_3
  - 0 milliseconds for P_4

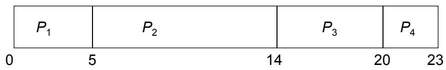
- Thus, the average wait time is (3 + 14 + 8 + 0)/4 = 6.25 milliseconds

20

Exercise

- In the previous example, what would the average wait time be if we had been using the First Come, First Served algorithm?

Answer



- The Gantt chart above shows the wait times using FCFS
- The average wait time under FCFS is:

  (0 + 5 + 14 + 20)/4 = 9.75 milliseconds

- Thus, the Shortest Job First algorithm produces a shorter average wait time than FCFS

Shortest Job First

- Advantages:
  - SJF reduces the overall average waiting time
    - Thus SJF is provably optimal in that it gives the minimal average waiting time for a given set of processes
- Disadvantages:
  - Can lead to starvation
  - Difficult to know the burst time of the next process requesting the CPU
    - May be possible to predict, but not guaranteed
    - Unacceptable for use in interactive systems

More scheduling methods

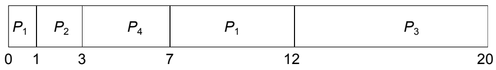
- Scheduling algorithms continued
  - Shortest remaining time first (SRTF)
  - Priority scheduling
  - Round robin (RR)

Shortest Remaining Time First

- Preemptive version of the SJF algorithm
  - CPU is allocated to the job that is closest to being completed
  - Can be preempted if there is a newer job in the ready queue that has a shorter completion time
- Suppose we have four processes arriving one CPU time cycle apart and in the following order:
  - P_1 with CPU burst of 6 milliseconds (arrives at time 0)
  - P_2 with CPU burst of 2 milliseconds (arrives at time 1)
  - P_3 with CPU burst of 8 milliseconds (arrives at time 2)
  - P_4 with CPU burst of 4 milliseconds (arrives at time 3)
- Using the SRTF algorithm we can schedule the processes as viewed in the following Gantt chart…..

25

Example



- P_1 starts at time 0 then P_2 arrives at time 1
- As P_2 requires less time (2 milliseconds) to complete than P_1 (5 milliseconds), then P_1 is preempted and P_2 is scheduled
- The next processes are then treated in this same manner
- Thus, the average wait time is:
  ((7 - 1) + (1 - 1) + (12 - 2) + (3 - 3))/4 = 4 milliseconds

Note: the above calculation accounts for the arrival time of each processes in that it is subtracted

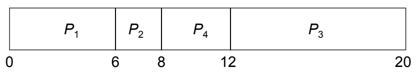
26

Exercise

- Using the same processes, arrival times and CPU burst times as in the previous example, what would the average wait time be if we were using the Shortest Job First algorithm?

27

Answer



- Thus, the average wait time is:

(0 + (6 - 1) + (12 - 2) + (8 - 3))/4 = 20/4 = 5 milliseconds

28

Priority Scheduling

- Algorithm that gives preferential treatment to important jobs
  - Each process is associated with a priority and the one with the highest priority is granted the CPU
  - Equal priority processes are scheduled in FCFS order
    - SJF is a special case of the general priority scheduling algorithm
- Priorities can be assigned to processes by a system administrator (e.g. staff processes have higher priority than student ones) or determined by the Processor Manager on characteristics such as:
  - Memory requirements
  - Peripheral devices required
  - Total CPU time
  - Amount of time already spent processing

Example

- Suppose we have five processes all arriving at time 0 in the following order and having the following CPU burst times:
  - P_1 with CPU burst of 9 milliseconds, priority 3
  - P_2 with CPU burst of 2 milliseconds, priority 4
  - P_3 with CPU burst of 1 millisecond, priority 1
  - P_4 with CPU burst of 5 milliseconds, priority 3
  - P_5 with CPU burst of 6 milliseconds, priority 2
- Assuming that 0 represents the highest priority, using the priority algorithm we can view the result as the following Gantt chart:

P_3	P_5	P_1	P_4	P_2
0 – 1	1 – 7	7 – 16	16 – 21	21 – 23

Priority Scheduling

- For the previous example, the average waiting time is:
  (7 + 21 + 0 + 16 + 1)/5 = 9 milliseconds
- Advantages:
  - Simple algorithm
  - Important jobs are dealt with quickly
  - Can have a preemptive version
- Disadvantages:
  - Process starvation can be a problem
    - Can be alleviated through the aging technique: gradually increasing the priority of processes that have been waiting a long time in the system

Round Robin

- Preemptive algorithm that gives a set CPU time to all active processes
  - Similar to FCFS, but allows for preemption by switching between processes
  - Ready queue is treated as a circular queue where CPU goes round the queue, allocating each process a pre-determined amount of time
- Time is defined by a time quantum: a small unit of time, varying anywhere between 10 and 100 milliseconds
- Ready queue treated as a First-In-First-Out (FIFO) queue
  - new processes joining the queue are added to the back of it
- CPU scheduler selects the process at the front of the queue, sets the timer to the time quantum and grants the CPU to this process

Round Robin

- Two potential outcomes ensue:

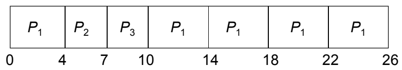
  1) If the process' CPU burst time is less than the specified time quantum it will released the CPU upon completion
  - Scheduler will then proceed to the next process at the front of the ready queue

  2) If the process' CPU burst time is more than the specified time quantum, the timer will expire and cause an interrupt (i.e. the process is preempted) and execute a context switch
  - The interrupted process is added to the end of the ready queue
  - Scheduler will then proceed to the next process at the front of the ready queue

33

Example

- Suppose we have three processes all arriving at time 0 and having CPU burst times as follows:
  - P_1 with CPU burst of 20 milliseconds
  - P_2 with CPU burst of 3 milliseconds
  - P_3 with CPU burst of 3 milliseconds
- Supposing that we use a time quantum of 4 milliseconds, using the round robin algorithm we can view the result as the following Gantt chart:



34

Round Robin

- In the previous example P_1 executed for the first four milliseconds and is then interrupted after the first time quantum has lapsed, but it requires another 16 milliseconds to complete
- P_2 is then granted the CPU, but as it only needs 3 milliseconds to complete, it quits before its time quantum expires
- The scheduler then moves to the next process in the queue, P_3, which is then granted the CPU, but that also quits before its time quantum expires

35

Round Robin

- Now each process has received one time quantum, so the CPU is returned to process P_1 for an additional time quantum
- As there are no other processes in the queue, P_1 is given further additional time quantum until it completes
  - No process is allocated the CPU for more than one time quantum in a row, unless it is the only runnable process
- The average wait time is ((10 - 4) + 4 + 7)/3 = 5.66 milliseconds

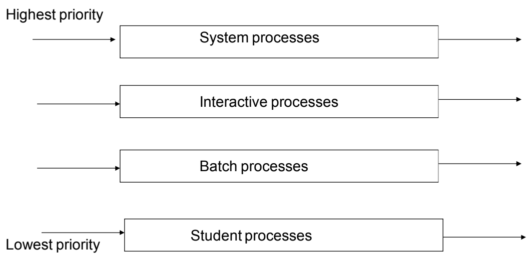
36

Round Robin

- The performance of the round robin algorithm depends heavily on the size of the time quantum
  - If time quantum is too large, RR reduces to the FCFS algorithm
  - If time quantum is too small, overhead increases due to amount of context switching needed
- Advantages:
  - Easy to implement as it is not based on characteristics of processes
    - Commonly used in interactive/time-sharing systems due to its preemptive abilities
  - Allocates CPU fairly
- Disadvantages:
  - Performance depends on selection of a good time quantum
    - Context switching overheads increase if a good time quantum is not used

37

Multilevel Queue



38

Multilevel Queue

- Each queue has its own scheduling algorithm
  - e.g. queue of foreground processes using RR and queue of batch processes using FCFS
- Scheduling must be done between the queues
  - Fixed priority scheduling: serve all from one queue then another
    - Possibility of starvation
  - Time slice: each queue gets a certain amount of CPU time which it can schedule amongst its processes
    - e.g. 80% to foreground queue, 20% to background queue

39

End of Section

- Operating systems concepts:
  - communicating sequential processes;
  - mutual exclusion, resource allocation, deadlock;
  - process management and scheduling.
- Concurrent programming in Java:
  - Java threads;
  - The Producer-Consumer problem.
- Next section: Memory Management

40

Comp 104: Operating Systems Concepts

Memory Management Systems

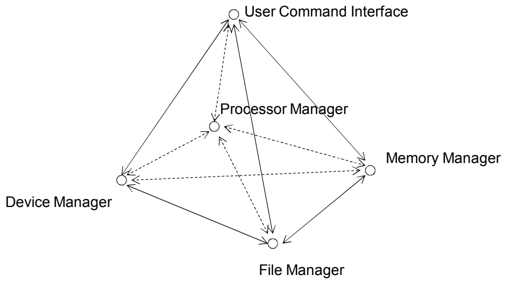
41

Today

- Memory management
  - Number systems and bit manipulation
  - Addressing
- Simple memory management systems
  - Definition
  - Issues
  - Selection policies

42

Operating System – An Abstract View



43

Recap - Binary

- Starting from right, each digit is a power of two
  - (1,2,4,8,16…)
    - 110110 = 2+4+16+32 = 54
- To convert decimal to binary, keep subtracting largest power of 2
  - e.g. 37 (sub. 32, sub. 4, sub 1) = 100101
- n bits can represent 2^n numbers
  - range 0 to $2^n – 1$
- Conversely, to represent n numbers, need $\log_2(n)$ bits

44

Recap - Octal and Hex

- Octal is base 8 (digits 0-7)
  - 134 (oct) = (1*64) + (3*8) + (1*4) = 92
- To convert between octal and binary
  - think in groups of 3 bits (since 8 = 2^3)
  - 134(oct) = 001 011 100
  - 10111010 = 272 (oct)
- Hex is base 16 (A-F = 10-15)
  - B3 (hex) = (11*16) + (3*1) = 179
- Conversion to/from binary
  - using groups of 4 bits (since 16 = 2^4)
  - B3 (hex) = 1011 0011
  - 101111 = 2F (hex)

45

Recap - Bit Manipulation

- Use AND (&) to mask off certain bits
  - x = y & 0x7 // put low 3 bits of y into x
- Use left and right shifts as necessary
  - x = (y & 0xF0) >> 4 // put bits 4-7 of y into bits 0-3 of x
- Can also test if a bit is set
  - if (x & 0x80)… // if bit 7 of x is set…

('0x' states that a number is in hexadecimal)

46

Recap - Bit Manipulation

- Can switch a bit off
  - x = x & 0x7F // unset bit 7 of x (assume x is only 8 bits)
- Use OR (|) to set a bit
  - x = x | 0x80 // set bit 7 of x
- A right shift is divide by 2; left shift is multiply by 2
  - 6 << 1 = 0110 << 1 = 1100 = 12
  - 6 >> 1 = 0110 >> 1 = 0011 = 3

47

Memory Management

- A large-scale, multi-user system may be represented as a set of sequential processes proceeding in parallel
- To execute, a process must be present in the computer's memory
- So, memory has to be shared among processes in a sensible way

48

Memory

- Memory: a large array of words or bytes, each with its own address
- The value of the program counter (PC) determines which instructions from memory are fetched by the CPU
  - The instructions fetched may cause additional loading/storage access from/to specific memory addresses
- Programs usually reside on a disk as a binary executable file
- In order for the program to be executed it must be brought into memory and placed within a process
- When the process is executed it accesses data and instructions from memory, then upon termination its memory space is declared available

49

Address Binding – Compile Time

- Programs often generate addresses for instructions or data, e.g.

```
START:    CALL FUN1
               .
               .
             LOAD NUM
             JUMP START
```

- Suppose assemble above to run at address 1000, then jump instruction equates to JUMP #1000
- Consider what happens if we move program to another place in memory
- Obvious disadvantage for multiprogrammed systems
- Fixed address translation like this is referred to as compile-time binding

50

Load-Time Binding

- Ideally, would like programs to run anywhere in memory
- May be able to generate position-independent code (PIC)
  - aided by various addressing modes, e.g.
    PC-relative: JUMP +5
    Register-indexed: LOAD (R1) #3
- If not, binary program file can include a list of instruction addresses that need to be initialised by loader

51

Dynamic (Run-Time) Binding

- Used in modern systems
- All programs are compiled to run at address zero
- For program with address space of size N, all addresses are in range 0 to N-1
- These are logical (virtual) addresses
- Mapping to physical addresses handled at run-time by CPU's memory management unit (MMU)
- MMU has relocation register (base register) holding start address of process
  - Contents of registers are added to each virtual address

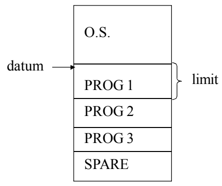
52

Logical and Physical Addresses

- Addresses generated by the CPU are known as logical (virtual) addresses
  - The set of all logical addresses generated by a program is known as the logical address space
- The addresses seen by the MMU are known as physical addresses
  - The set of all physical addresses corresponding to the logical addresses is known as the physical address space
- The addresses generated by compile-time binding and load-time binding result in the logical and the physical addresses being the same
- Run-time binding results in logical and physical addresses that are different

53

Simple System Store Management

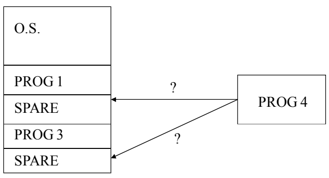


- Store is allocated to programs in contiguous partitions from one end of the store to the other
- Each process has a base, or datum (where it starts)
- Each process also has a limit (length)

54

Problem

- What if one program terminates and we want to replace it by another?

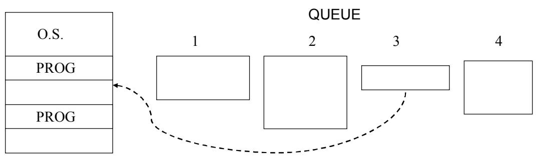


New program may not fit any available partition

55

First Possibility

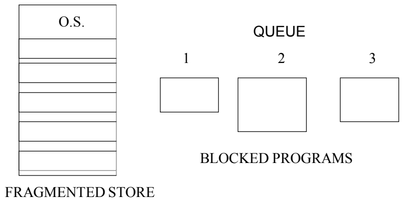
- In a multiprogramming system where we have a queue of programs waiting, select a program of right size to fit partition



56

Problems

- Large programs may never get loaded (permanent blocking or starvation)
- Small gaps are created (fragmentation)



57

Question

- In a computer memory, a 100K partition becomes available. In the ready list is a program image of size 300K, plus three others of sizes 100K, 85K and 15K.
- Assuming that our current priority is to avoid starvation of the 300K program, which of those in the list should be swapped into the available partition?

a) The 100K program
b) The 85K program
c) The 15K program
d) Both the 15K and the 85K programs
e) None of the above

Answer: e
If we are avoiding the starvation of the 300K program, then we need to wait for an adjacent partition to become available to allow for the 300K program to run

58

Loading Programs

- To avoid starvation, may need to let a large program hold up queue until enough space becomes available
- In general, may have to make a choice as to which partition to use
- Selection policies:
  - First fit**:** Choose first partition of suitable size
  - Best fit**:** Choose smallest partition which is big enough
  - Worst fit**:** Choose biggest partition

59

Question

- A new program requires 100K of memory to run. The memory management approach adopted is a simple partitioning one, and the operating system has the following list of empty partitions:

60K, 240K, 150K, 600K, 108K, 310K

- Assuming that the 150K partition is chosen, say which of the following selection strategies is being used:

a) First fit
b) Best fit
c) Worst fit
d) All of the above
e) None of the above

Answer: e
First Fit would select 240K
Best Fit would select 108K
Worst Fit would select 600K
… as none of these select the 150K partition, then some other strategy has been used!

60

Problems with Approach

- Fragmentation may be severe
  - 50% rule
    - For first-fit, if amount of memory allocated is N, then the amount unusable owing to fragmentation is 0.5N
  - Overhead to keep track of gap may be bigger than gap itself
  - May have to periodically compact memory
    - Requires programs to be dynamically relocatable
- Difficult dealing with large programs

61

Problems (cont'd)

- Shortage of memory
  - arising from fragmentation and/or anti-starvation policy
  - may not be able to support enough users
  - may have difficulty loading enough programs to obtain good job mix
- Imposes limitations on program structure
  - not suitable for sharing routines and data
  - does not reflect high level language structures
  - does not handle store expansion well
- Swapping is inefficient

62

Swapping

- Would like to start more programs than can fit into physical memory
- To enable this, keep some program images on disk
- During scheduling, a process image may be swapped in, and another swapped out to make room
  - also helps to prevent starvation
- For efficiency, may have dedicated swap space on disk
- However, swapping whole processes adds considerably to time of context switch

63

Today

- Dynamic Loading & Linking
  - Shared Libraries
- Memory organisation models
  - Segmentation
    - Address structure
    - Memory referencing

Dynamic Loading

- Not always necessary to load the entire image
  - Image can consist of:
    - Main program
    - Different routines, classes, etc
    - Error routines
  - Dynamic Loading allows only parts of the image to be loaded, as necessary
    - When a routine calls another routine, it checks to see if it has been loaded…
      - … if not, the relocatable linking loader is called
    - Advantage – unused routines are never loaded, thus the image is kept smaller

Linking

- Linking is the combination of user code with system or 3rd party libraries
  - Typically done as part of, or after the compilation process
- Static Linking
  - Copies of the libraries are included in the final binary program image
    - Can result in large images due to inclusion of many libraries (which in turn might link to other libraries…)
    - Wasteful both in terms of disc storage and main memory
    - Can be managed by dynamic loading, but shared libraries are still repeated in memory multiple times.

Linking (cont)

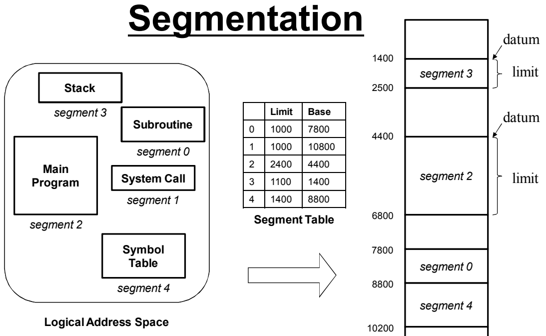
- Dynamic Linking
  - A stub is included in the image for each library routine
    - Indicates how to:
      - Locate memory resident library routine (if already loaded),
      - Load the library (if not loaded)
    - Allows re-entrant code to be shared between processes
      - Supports Library Updates (including versioning)
      - Keeps disc image small
  - Requires some assistance from the OS
    - Lower level memory organisation necessary…

Memory Organisation

To ameliorate some of the software problems arising from the *linear store,* more complex memory models are used which organise the store hierarchically:

- Segmentation
  - subdivision of the address space into logically separate regions
- Paging
  - physical subdivision of the address space for memory management

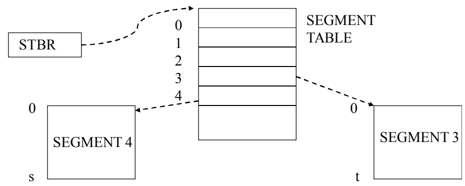
Segmentation



NOTES: *Each segment has its own partition*
Segments need not be contiguous

Segmented Address Structure

- Each process has a segment table
  - contains datum and limit values of segments
- Address of table held in segment table base register (STBR) (saved during context switch)



Memory Referencing

- For linear store, machine code is
  LOAD ADDR
- For segmented store, machine code is
  LOAD SEG, ADDR
  - Hardware looks up segment base address in table, then adds in-segment address to produce absolute address

Question

- A program is split into 3 segments. The segment table contains the following information:

segment	datum	limit
0	1700	5500
1	5600	8100
2	8300	9985

- where 'limit' is the physical address following the end of the segment, and instructions take the form opcode segment, offset
- If the program executes
  - **LOAD 1, 135**
- what physical address is accessed?

a) 1835
b) 5735
c) 8435
d) 8235
e) 5635

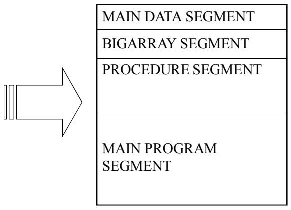
Answer: b
5600 (from segment 1) + 135 (offset)

72

Advantages of Segmentation

- Memory allocation reflects program structure

```
begin
  int a, b, c; float …; etc.
  [1:n] float bigarray;
  procedure something …
  begin
     …
  end
  # MAIN PROGRAM #
end
```

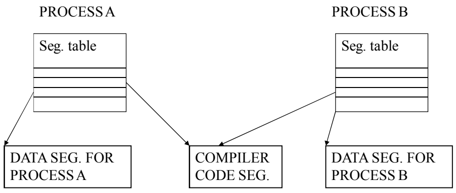


This means...

- It is easier to organise separate compilation of procedures
- Protection is facilitated
  - array bound checking can be done in hardware
  - code segments can be protected from being overwritten
  - data segments can be protected from execution
- Segments can be shared between processes

Segment Sharing

- Suppose 2 users compiling different programs

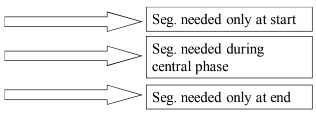


Processes have own data segments, but can share compiler code segment if it is pure (re-entrant), i.e. the code never modifies itself during execution.
-- Economises on memory, and more efficient.

Other Advantages

- Large programs broken into manageable units
  - Store allocation more flexible
  - Not all of a program need be in store at same time

```
begin
  procedure initialise
  { … }
  procedure mainpart
  { … }
  procedure finish
  { … }
end
```



Segments can be kept on backing store until needed, then swapped in
-- Allows more programs to be kept in memory
-- Swapping more efficient

Question

- Process A and process B both share the same code segment S. Which of the following statements is (are) true?

I. An entry for S appears in both segment tables
II. The segment code must be re-entrant
III. The segment code must be recursive

a) I only
b) II only
c) I and II
d) I and III
e) I, II and III

Answer: c
I and II – as the segment is shared, both processes need to index it (i.e. include an entry in their respective segment tables, and, the code must not be changed by its use (i.e. it should be re-entrant). Recursion is irrelevant to this issue.

77

But...

- Swapping can only work effectively if programs sensibly segmented
- Still have space allocation problems for awkward-sized segments
  - made worse by frequent need to find space whenever swapping-in occurs
  - fragmentation and blocking remain problems

Today

- Paging
  - Paged memory
  - Virtual addressing
- Page replacement
  - Principle of Locality

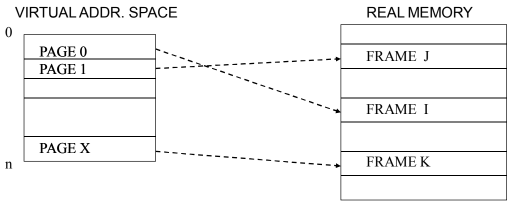
79

Paging

- Paging is the physical division of a program's (or segment's) address space into equal-sized blocks called pages
- Each page resides within a page frame in the real memory
- Pages which are consecutive in the address space (virtual memory) need not occupy consecutive page frames in real memory

80

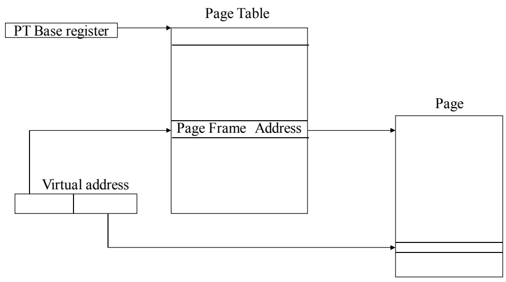
Page Mapping



- Translation of virtual addresses (used by program) into real addresses performed by hardware via a page table per process

81

Paged Memory



82

Virtual Addressing

Virtual Address

Virtual Page Number	In-page address

Storage Mapping function

1. PTBR addresses Page Table in memory.

2. Virtual page number indexes page table to produce real page address

3. In-page address indexes real page.

83

Question

- In a paged memory system, why are page sizes invariably a power of 2?

a) Because computer memory is usually a multiple of 1K, which is a power of 2.
b) Because pages have to begin at address boundaries that are even.
c) Because virtual address spaces are usually a power of 2 in size
d) Because it simplifies indexing of the page table and the calculation of the page offset.
e) Because most data types occupy even numbers of bytes.

Answer: d
Because it simplifies indexing of the page table and the calculation of the page offset.

84

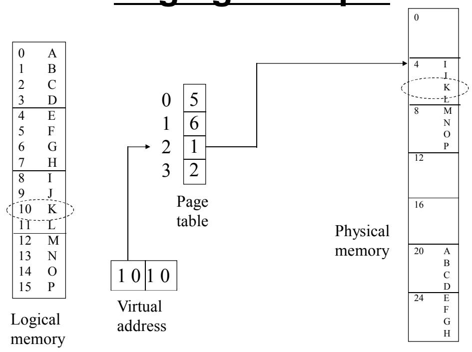
Question

- A computer uses 16-bit addressing. Its page size is 512 bytes. What is the maximum number of entries that the page table must be capable of holding?

  a) 16
  b) 64
  c) 128
  d) 256
  e) 512

Answer: c
128; 9 bits are used for addressing the 512 bytes in each page, so the remaining 7 bits are used to address the pages

Paging Example



Question

- The page table shown below is for a job in a paged virtual storage system with a page size of 1K:

segment	datum
0	4
1	2
2	0
3	1

- A virtual address of [1, 352] would map on to what actual address?

  a) 354
  b) 1376
  c) 2352
  d) 2400
  e) 4448

Answer: d
2048 (from page 1) + 352 (offset)

Segmentation & Paging

- Segmentation:
  - Logical division of address space
  - varying sized units
  - units 'visible' to program
- Paging:
  - Physical division of address space
  - fixed-size units
  - units bear no relation to program structure

Segmentation & Paging

- Either may be used as a basis for a swapping system
- Store may be both segmented and paged
  - more complex mapping function using 2 tables
- Advantages of paging:
  - fixed-size units make space allocation simpler
  - normal fragmentation eliminated, but still some internal fragmentation, i.e. space wasted within frames

89

Example: The Intel Pentium

- Supports segmentation with paging
- CPU generates logical address, which is passed to segmentation unit
- Segmentation unit produces a linear address, which is passed to paging unit
- Paging unit generates physical address in main memory

90

Virtual Memory

- The maximum logical address space per process may be smaller than physical memory
- Conversely, it may be larger!
  - May want to run a 100MB process on a machine with 64MB memory
- Possible with paging
  - Not all pages need be in memory
  - Unneeded pages can reside on disk
  - Memory becomes virtual, i.e. not restricted by physical memory

91

Problem 1

- What happens if a process references a page that is not in main store?
  - A page fault ensues
- Page fault generates an interrupt because address references cannot be satisfied until page swapped in
- O.S. response is normally to fetch page required (demand paging)

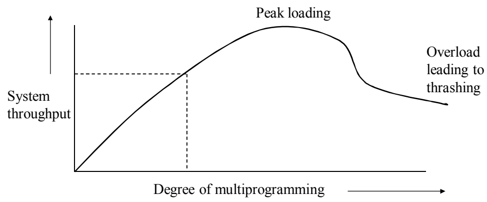
92

Problem 2

- How do we make room for fetched page? (page replacement problem)
- Would like to swap out pages not immediately needed
  - have to guess!
- A poor guess will quickly lead to a page fault
- Many poor guesses will lead to persistent swapping (thrashing)

93

Thrashing



Thrashing comes about as result of system overload
- can be delayed by good page replacement policy

94

Page Replacement

- Optimal policy: swap out page that will not be needed for longest time in the future
- To estimate this, use...

PRINCIPLE OF LOCALITY

Over any short period of time, a program's memory references tend to be spatially localised

Consequence is that program's memory needs in next time period are likely to be close to those during last time period

95

Question

- Which of the following programming constructs tend to contribute to the phenomenon expressed in the Principle of Locality?

I. Iteration (e.g. FOR and WHILE loops)
II. Selection (e.g. IF-statements)
III. Recursion

a) I only
b) III only
c) I and III only
d) II and III only
e) I and II only

Answer: c
I and III only

96

Today

- Virtual Memory
- Page Replacement
  - Working set model
  - Page replacement policies

97

Working Set Model

- The working set (Denning, 1968) of a process is defined to be the set of resources (pages) *W(T,s)* defined in the time interval *[T, T+s]*
- By the principle of locality,

$$W(T,s) \approx W(T-s,s)$$

- Hence, for each process:
  - try to ensure its working set is in memory
  - estimate working set by recording set of pages referenced in preceding time interval

98

Question

- Consider the following sequence of page references in a paged memory management system:

page **| p | q | r | q | q | q | p | r | r | q |**
time **0 1 2 3 4 5 6 7 8 9 10**

- What is the working set expressed as W(3,4)?

a) q
b) r
c) qr
d) pq
e) pqr

Answer: d
pq

99

Question

- Consider the following sequence of page references in a paged memory management system:

page **| p | q | r | q | q | q | p | r | r | q |**
time **0 1 2 3 4 5 6 7 8 9 10**

- What would be the predicted working set expressed as W(10,3)?

a) q
b) r
c) qr
d) pq
e) pqr

Answer: c
qr

100

Working Set

- The accuracy of the working set depends upon its size
  - If set too small will not cover entire locality
  - If set too large will cover several localities
- Over time the working set of a process will change as references to data and code sections move from one locality to another
  - Page fault rates will vary with these transitions

101

Related Policies

- Replacement policies for use with demand paging, based loosely on working set principles, include
  - Least Recently Used (LRU). Replace least-recently used page
  - First-In-First-Out (FIFO). Replace page longest in memory
  - Least Frequently Used (LFU). Replace page with fewest references in recent time period

102

Question 14

- Consider the following sequence of page references in a paged memory management system:

page	| p | q | r | q | q | q | p | r | r | q |
time	**0 1 2 3 4 5 6 7 8 9 10**

- Page s arrives at time 10. Which of the following policies suggests we should throw out page p to make room for s?

I. LRU
II. LFU
III. FIFO

a) I only
b) III only
c) I and II
d) I and III
e) I, II and III

Answer: e
I, II and III

103

Anticipatory Paging

- Above policies not immune from thrashing
- A policy which more closely follows working set principles may require anticipatory paging
  - pages in working set are pre-fetched in anticipation of their need

104

Frame Allocation

- The fixed amount of free memory must be allocated amongst the various processes
  - Need to determine how many frames each process should get
- Each process will need a minimum number of pages
  - Dependent upon the architecture
- Allocation schemes
  - Equal allocation: each process gets an equal share of frames
  - Proportional allocation: allocate frames according to the size of the process
    - Could also implement proportional allocation based on process priorities

105

Performance Considerations

- Segmentation and paging overcome many limitations of linear store model, but…
- There is a performance hit
  - Each memory reference may require 2-3 store accesses
- Special hardware may help
  - registers to hold base address of current code and data segments may allow tables to be bypassed
  - special memory can aid fast table look-up (cache memory, associative store)

106

Page Size

- A large page size means a smaller page table
- However, large pages mean more wastage
  - On average, 50% of a page per process lost to internal fragmentation
- Small pages give better resolution
  - Can bring in only the code/data that is needed for working set
- However, small pages increase chances of page faults

107

Example: Windows XP

- Virtual memory implemented using demand paging
- Also implements clustering
  - When page fault occurs, bring in a number of additional pages following page required
- Each process has a working set minimum
  - Guaranteed number of pages in memory (e.g. 50)
- Also has a working set maximum (e.g. 345)
  - If page fault occurs and max has been reached, one of the process's own pages must be swapped out
- If free memory becomes too low, virtual memory manager removes pages from processes (but not below minimum)

108

End of Section

- Memory Management
  - Linear store model and its problems
  - Segmentation and paging
  - Virtual memory and page replacement
- The next section of the module will be Files and I/O

109