MONASH University
Information Technology

FIT9134
Computer architecture and operating systems

Week 5

Operating Systems III:
Memory Management – Virtual Memory

Topic Summary

- Single-processing vs Multi-processing
- Process Swapping
- Virtual memory
- Memory partitioning
- Paging/Segmentation

Memory management

- Physical main memory is finite (and expensive).

- Single-processing : 1 process in memory at any one time. Easy to implement – either it fits or it doesn't.
- Multi-processing : multiple processes in memory at the same time. Some issues to consider :
  - are they all allocated the same memory size?
  - where are they located in memory, relative to one other?
  - does a process need to be entirely stored in memory in order to run? How much memory is needed in total?
  - when a process finishes, what happen to it? Can we bring in another new process(es) in its place?

- Solutions : ***Swapping***, ***Virtual Memory***.

Swapping

- Swapping is a technique used to run more than one process at once. It allows the computer to rapidly "swap" its CPU between the process by loading and unloading them into/from memory. The switching occurs sufficiently quickly that it gives the user the illusion that the system is multi-tasking.

- In its basic implementation, only one process is in memory, and being executed, at any one time. This is relatively easy to implement, but not efficient, since an *entire process* needs to be swapped in/out at once. An improvement is to swap partial processes, at the cost of increased complexity.

Virtual Memory

- Virtual Memory is a more complicated technique used to solve memory management problems. It allows the computer to separate *logical* program addresses from actual *physical* addresses, using dynamic relocation of program addresses in memory.

- It allows programs to be divided up into small sections stored in different parts of memory, and allows execution of programs larger than physical memory ie. only currently executing portion of program is in memory.

- Virtual memory may be implemented using *paging* and/or *segmentation*.

Memory Fragmentations

- Allowing multiple processes to reside in memory creates the potential problem of ***Memory Fragmentation*** – sections of memory locations which are "free", but are not contiguous (ie. the free spaces, possibly in different sizes, are scattered throughout the memory), leading to possible memory wastage. This happens when processes finish running and are removed from memory at various times.

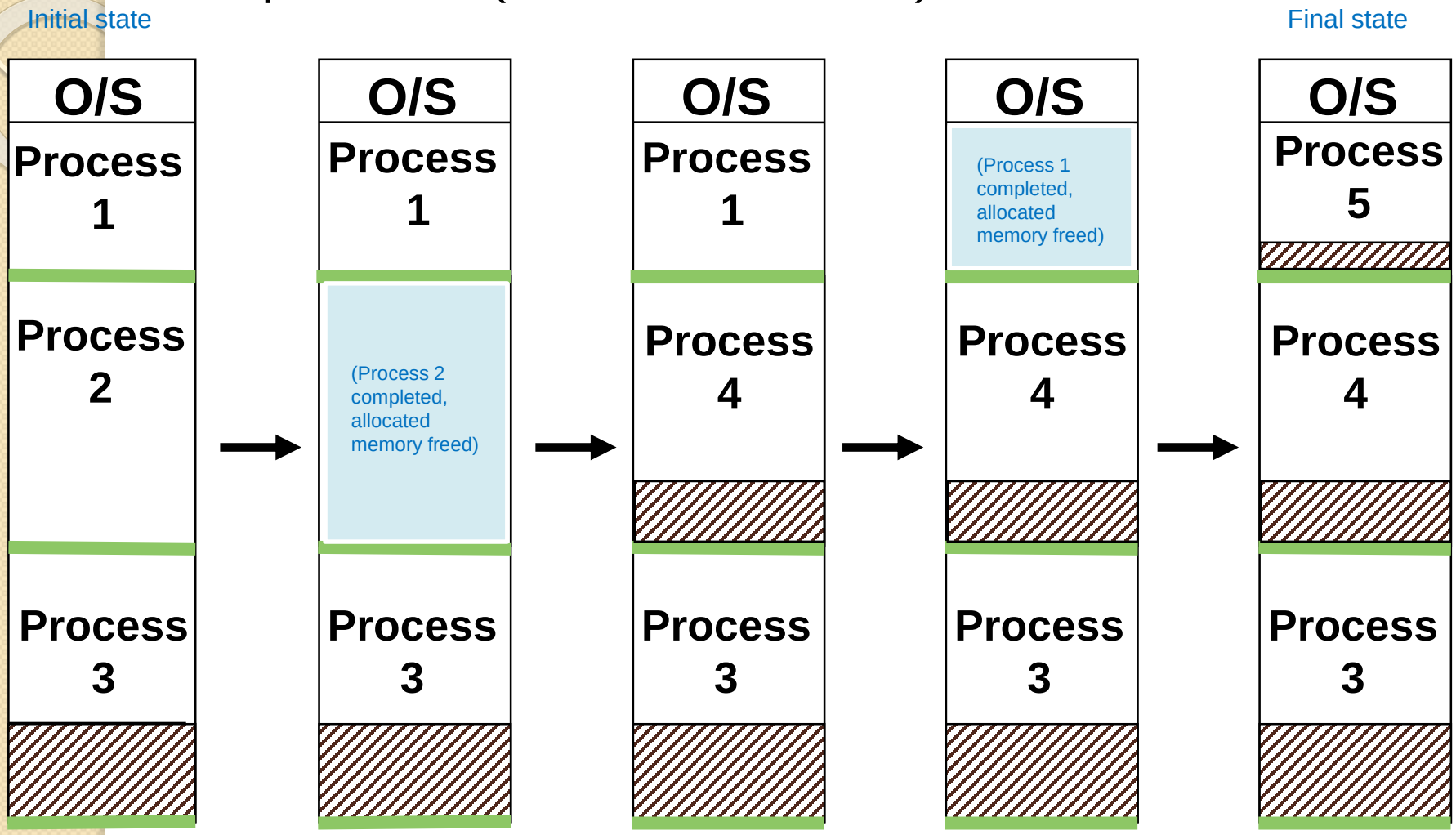
- *External Fragmentation* – memory which is **unallocated** (to any processes) and unused.
- *Internal Fragmentation* – memory which is **allocated** (to a process) and unused.

Memory Partitioning Schemes

- ***Fixed Partitioning*** :
  - main memory is divided into fix-sized partitions
  - the partitions may be all equal in size, or be of different sizes. They *do not change in real-time*.
  - simple implementation, but prone to ***Internal Fragmentation*** (***allocated*** memory which cannot be used).

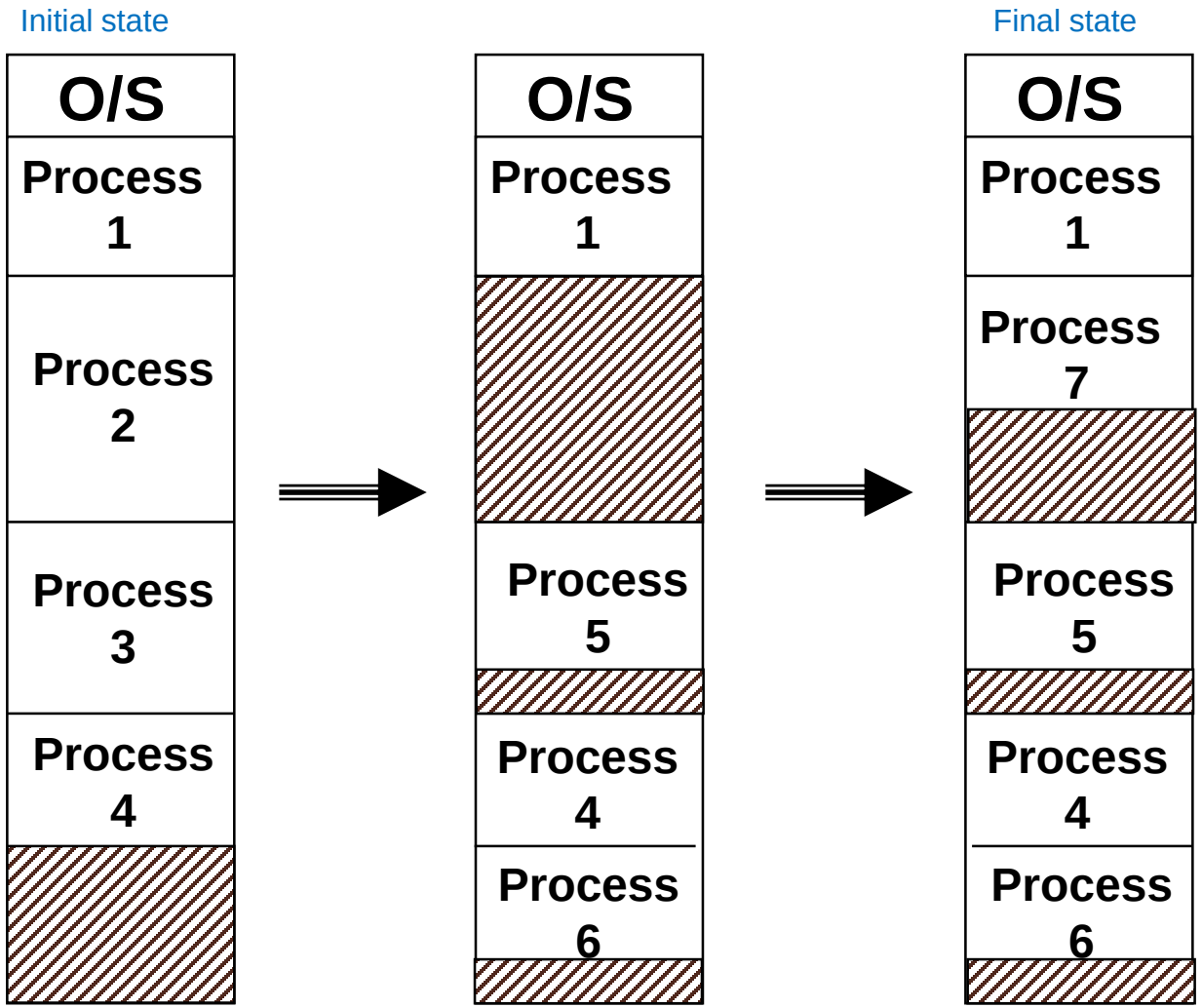
- ***Dynamic Partitioning*** :
  - main memory is dynamically divided into variable-sized partitions, which *can be changed in real-time*.
  - more flexible, but more complex implementation, and prone to ***External Fragmentation*** (***unallocated*** memory which cannot be used).

Example : **Internal Fragmentation** with fixed partitions (but different sizes)



Example : **External Fragmentation** with variable sized partitions

In this example, what is the order in which the processes arrive/finish, in states #2 & #3?



Possibly unusable unallocated memory

Memory Management – Paging

- ***Paging***: Programs/data divided into logical sections called *PAGES* and physical memory divided into areas called page *FRAMES*. A ***Page-Table*** then translates the pages to their corresponding frames at run-time.

- Page size and Frame size are equal.

- When a program is running, its pages are brought into memory as required – ie. only the portion being executed is in memory, the rest stays on the hard drive. So we can now run programs which may be larger than the available physical memory.

Memory Management – Paging

- Page size is usually hardware-dependent, and is typically in powers of 2 (e.g. 512Bytes, 1024Bytes, etc) to aid address translation. 2-4KB size is common.

- Large page sizes will :
  - increase ***internal fragmentation*** (ie. more likely that a frame will have unused portion, as a process/page may not need the entire frame), but
  - reduce maintenance overhead of page tables since there are less pages to maintain

- Small page sizes will have the opposite effect.

- So it's important to have the correct page size.

Memory management - Paging

- During program execution, some program "pages" stay in memory in page frames, the rest stay on secondary storage (in special area, sometimes known as ***swap space***) until required (when they are then "swapped" back in).

- Program logical addresses are converted to the form:

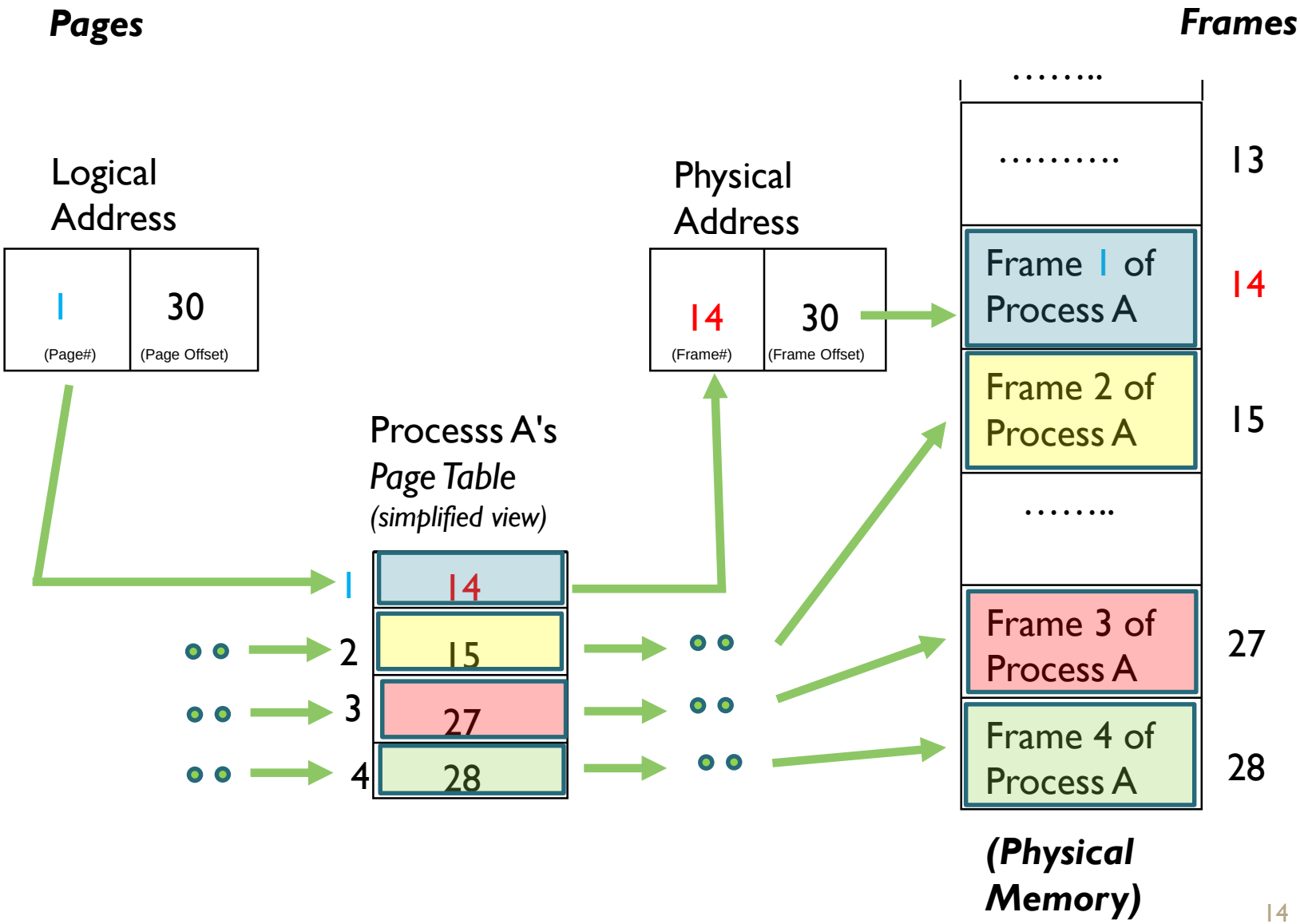
  page_number **:** ***page_offset***

  e.g. a 32-bit address may use 21 bits for page # and 11 bits for offset (so total of 2^{21}, or ~2.1 million pages).

- A *Page #* is used as an index into the Page Table, to find the corresponding *Frame #*.

- Page Tables reside in memory. ***Each process has its own Page Table.***

Example : Logical Pages versus Physical Frames



Page Table

- ***The O/S maintains a Page Table for each process***. A Page table has one entry for each page of a process. If a page table is large, it may need to be paged also.

- Apart from frame #'s, a page table will typically also contain other data, such as whether page is in memory or on disk, whether page has been modified, etc.

Process Table

- ***The O/S also maintains a Process Table*** for all the processes. A *Process Table* contains entries called ***Process Control Blocks*** (***PCB***). Each PCB represents one process.

- A typical PCB entry contains data such as :
  - process state & process ID
  - program counter
  - memory address of the ***Page Table***
  - resources in-used/needed
  - etc

We will discuss Process Tables again when we discuss Process Management

Improving Paging performance by Caching

- Data in page table is constantly accessed when the process is running. For efficiency, special high-speed memory is provided for the "active" part of page table, called ***caches***, ***associative memory*** or ***Translation Lookaside Buffer (TLB)***.

- Each location in the caches can be searched simultaneously rather than sequentially.

- The cache memory is small and stores only active pages. It is searched first for page #. If not found, search then goes to the page table in conventional memory.

Paging with Cache

ie. Pages are found in the cache 80% of the time

- ***Hit ratio*** = percentage of times a page is found in the cache. Highly-dependent on the number of registers in the cache.
- Eg. If we use a 16-register cache with an 80% hit rate, 50 ns cache access time and 750 ns memory access time, the comparative access times would be:

Hence in this case, the use of caching results in approx. *36.7%* decrease in single memory access time

With no caching :

750 + 750 = 1500 ns for every access

Assessing Page Table

Assessing physical Memory Frames

OR,

With caching :

0.8 * (50 + 750) + 0.2 * (50 + 750 + 750) => 950 ns for every access

Assessing Cache memory

Assessing Page Table

Page Replacement/Swapping

- If a required page is not found in memory, then an interrupt called Page Fault results. This causes the required page to be loaded (from secondary memory) into main memory, and the page table updated. Page table indicates whether page is in memory, using a "valid/invalid" bit entry.

- If process memory is full (all allocated frames used) some form of page replacement policy is needed i.e. replace current frame with new one. This is called Page Replacing/Swapping. If evicted frame modified, a write to disk is necessary before replacing page.

Paging Algorithms

ie. O/S would need to keep track of each memory access with a time-stamp

Some common Page replacement algorithms :

- Least Recently Used (LRU) page swapped out - each access time stamped ➔ extra overhead.
- Least Frequently Used (LFU) page swapped out - each access adds to count, but this causes problems with new pages that have just been brought in (which would naturally have a low frequency count!)
- First In First Out (F.I.F.O) - oldest page swapped out first; simplest but has the disadvantage that the most heavily used page may be replaced.

ie. O/S would need to keep track of how many times each page has been accessed

Paging Algorithm Variations

- Some algorithms use a "reference" bit which is set when page is used, but periodically reset by system.
- Not Used Recently (NUR) - modification of LRU, that also looks at whether page has been modified (in addition to being accessed).
- Second Chance algorithms are modification of FIFO to allow second chance if reference bit is set. Page is time stamped again as though new.
- Modern O/S'es often use a combination of different paging algorithms.

Program Locality

Why Virtual Memory works:

- Programs tend to work within sections
  - As the program proceeds, sections change, but at any one time, for a period of time, programs will work within the same section. This is the concept of *Locality*.
  - Eg: a program may spend a lot of time performing a loop, or accessing consecutive elements of a list, etc…

- "Working Set"
  - = the minimum number of pages that meets the locality requirement (ie. the amount of primary memory, measured in pages, that is required for a program to make effective progress without excessive page-swapping).

  - Some operating systems maintain estimates of the working set for all running processes.

The Working Set Model

- As the number of pages allocated to a process decreases, number of page faults increase. Need to store enough pages of the process so that the CPU may be used effectively.

- Minimum set is called ***Working Set*** (or *Resident Set*) of the process. If number of pages allocated falls below this level then Thrashing will result.

- Thrashing is when so many page faults occur that most activity is just paging in and out while the CPU is blocked waiting on I/O. This will have a very serious performance impact, since I/O access time is typically ***much*** slower than memory access time.

The Working Set Model

- Idea of working set relies on 'locality of execution' principle - that only a small number of pages from few modules in program are in use at one time. The O/S must keep track of current working set.

- Tuning a system that uses paging involves setting initial size on working set.

- Pages in working set may be :
  - loaded on demand,
  - as a result of direct page faults (***demand paging***), or
  - anticipated in advance (***pre-paging***) based on some common access pattern.

Page Replacement Policy

Local vs Global Page Replacement Policies

- Whenever there is a need to replace pages in memory, what is the range of pages that the replacement policy is applied to?
  - local - replaces pages actually owned by the process
  - global - replaces pages from any process

- Global replacement is generally more flexible, because there is a larger set of replacement pages to choose from;
  - disadvantage : can reduce the working set size for unrelated processes.

- Crucial pages (eg. the actual disk driver, video driver, etc) can be marked as "locked" so that they are never swapped out.

Segmentation

- ***Segmentation*** is another approach to memory management, similar to Paging.

- Reflects the *Logical* division in programs and data, eg. may have segment for global variables, code portions of functions and procedures, etc
  - the division is decided by the programmer, and the O/S will then build the appropriate ***Segment Table*** (as opposed to a Page Table) to mirror the division.

- Program addresses are now of form:
  segment_number : ***offset***

Segmentation

- The address mapping for logical to physical addresses is maintained in a ***Segment Table*** (similar to a Page Table).
  - *each segment may vary in size* depending on its function (ie. logical division)
  - each segment is a logical entity of the program it represents

- compare this approach to *Paging : where each page/frame is of the same size*.

- Segmentations is more complicated to program than paging; so less popular in use to implement virtual memory.

Virtual Memory Technique

Advantages:

- Provides a large logical memory space to physical memory space ratio.
- Allows more processes to run concurrently.
- Process isolations - protect processes from each other. Each process has its own virtual memory space.
- Less I/O resource, as we load in only the required sections of a user process.

Disadvantages:

- Adds complexity to memory management
- Can cause performance degradation if not implemented properly - eg. Thrashing due to excessive page faults

Memory Management

Order of increasing complexity:

1. Single-processing – process loaded into memory and stays there until it has finished.
2. Multi-processing - all processes staying in memory.
3. Swapping - system can handle more processes than it has room for in memory, by swapping processes to and from disk.
4. Virtual memory - can handle processes bigger than physical memory using paging or segmentation.
5. Paging/Segmentation - memory subdivided into pages or segments.

Some Useful Texts/Resources

- **Harvey M. Deitel, Paul J. Deitel and David R. Choffnes -** *Operating Systems (3rd Edition)*
- **Lister A.M. & Eager R.D. -** *Fundamentals of Operating Systems, Macmillan*
- **Andrew S. Tanenbaum and Albert S. Woodhull -** *Operating Systems Design and Implementation (3rd Edition)*
- **free online resources** :

  http://www.linfo.org/virtual_memory.html

  http://www.faqs.org/docs/linux_admin/x1752.html

  http://www.linuxjournal.com/article/8178