Memory Allocation Schemes

The first four schemes were used in early computer systems that require the whole job to be loaded into a contiguous memory space before being executed.

Single-User Contiguous

- Support for only one job at a time irrespective of available space.
- Each job loaded in its entirety and allocated as much contiguous space as needed.
- If the program is too large to fit into available memory, it cannot be executed.
- Memory allocation algorithm is simple, and minimal:
  1. Evaluate incoming process to check that it fits in available memory
     - if it fits, load into memory; else, reject and evaluate next incoming process
  2. Monitor execution in memory
     - when process is finished, deallocate entire memory space and return to step 1

Fixed/Static Partitions

- First to allow multiprogramming.
- As many jobs can reside in memory as there are partitions.
- Partition size is static; system must be restarted to adjust size.
- Entire job must be loaded into memory before execution.
- Same job size restrictions as single-user (i.e., if job can't fit, it can't be executed).
- Introduced security concern of preventing jobs from accessing other partition memory space.
- Memory Table tracks partition sizes and statuses, memory addresses, and loaded jobs
- Problem of optimal partition sizes:
  - If too small, execution of large jobs will be delayed indefinitely
  - If too large, memory wastage is high
- Causes **internal fragmentation** (unused memory within occupied partitions).
- Slightly more involved algorithm than single-user:
  1. Check incoming job memory requirements
     - if greater than largest partition, reject job and go to next waiting job
     - if less than largest partition, go to Step 2
  2. Check the job size against the size of the first partition
     - if job fits, and partition is available, load the job into that partition
     - if partition is busy with another job, go to Step 3
  3. Check the job size against the size of the next partition
     - if job fits, and partition is available, load the incoming job into that partition
     - if partition is busy with another job, go to Step 4
  4. Repeat Step 3 till job is loaded into an available partition
     - if no partition is free, place job in waiting queue for loading at a later time
     - return to Step 1 to evaluate the next incoming job

Dynamic Partitions

- Jobs loaded into dynamically sized partitions that precisely fit each job.
- Efficient memory usage when first loaded with jobs.
- Introduces **external fragmentation** with subsequent loading.

- Same constraints of entire job residing in memory before execution, and if no partition large enough, job can't be loaded.
- Busy-, and free-lists are used to track partitions.
- Memory allocation policies used to optimise efficient memory use:
  - Best-fit
    - More efficient use of memory space
    - busy/free lists ordered by size from smallest to largest
  - First-fit
    - Faster allocation of jobs
    - busy/free lists ordered by memory location from lowest to highest
  - Worst-fit
  - Next-fit
  - Memory deallocation algorithms

Relocatable Dynamic Partitions

- Makes more efficient use of memory by relocating partitions.
- Jobs moved to one location to make free memory one large contiguous space to accomodate waiting jobs.
- Relocation and Bounds Registers used to assign new memory addresses for relocated jobs.
- Compaction (defragmentation) performs complex task of relocation:
  - Begin compaction when 75% of memory is in use
    - Disadvantage of additional overhead if no jobs waiting at the time
  - Begin compaction when jobs are waiting to be loaded
    - Disadvantage of constant monitoring of waiting queues
  - Begin compaction after predetermined duration
    - If time period too short, system resources wasted
    - If too long, excessive wait times for jobs

The next four schemes removed the requirement of the entire job being loaded into contiguous memory locations, introducing virtual memory.

Paged Memory Allocation

- Jobs divided into pages to fit into physical memory page frames.
  - pages too small results in excessively long Page Map Tables and increased overhead
  - pages too large results in excessive internal fragmentation
- Pages don't have to be in adjacent frames, removing need for jobs to be contiguous in memory.
- Memory is used more efficiently because empty page frames can be used by pages from any job.
- Memory Manager determines number of pages needed before locating enough empty page frames and loading the pages into them.
- External fragmentation is eliminated.
- Internal fragmentation returns because most jobs don't entirely fill the last page it occupies.
- Entire job still required to be loaded into memory before being executed.
- Increased complexity and overhead for the Memory Manager:
  - Job Table: one table for the whole system containing two values for each job in memory
    - size of job
    - memory address of its Page Map Table

- Page Map Table: one for every active job, with entries for each of its pages
  - page number
  - corresponding physical page frame number
- Memory Map Table: one table for the whole system with one entry for each page frame
  - physical page frame location
  - free/busy status

Demand Paging Memory Allocation

- Made virtual memory feasible.
- Utilises memory more efficiently at the expense of greater overhead and interrupt frequency.
- Removes restriction of having the entire job in memory.
- Provides the appearance of vast amounts of physical memory.
- Jobs are divided into equally sized pages residing on disk, and loaded into memory as needed.
- Pages that are never needed are never loaded, for example:
  - error handling is only processed in the event of errors
  - printing module is only processed if print output selected
- Takes advantage of sequential nature of programs; while one section is processed in memory, the rest remain idle on disk:
  - pages of different sections aren't accessed simultaneously (e.g., init, input, output, menus)
- Page replacement algorithms determine when, which, and how pages are swapped:
  - FIFO (first-in, first-out): oldest pages swapped out
  - LRU (least recently used): swaps out pages showing the least recent activity
  - Clock Replacement: LRU variant; uses circular queue
  - Bit Shifting: LRU variant; sets MSB of reference byte if page is accessed, and performs right shift each cycle (e.g., 10000000, 11000000, 01100000 = first loaded into memory, next cycle accessed, next cycle not accessed)
- Three tables are updated with each page swap:
  - Page Map Table (for both the victim page and replacement page)
  - Memory Map Table
- Same tables as paged memory allocation used, with three additional fields in the PMT:
  1. *status bit*: is the page currently in memory
     - Quicker to scan the table than retrieve a page from disk
  2. *dirty bit*: have page contents been modified while in memory
     - Used to save time by only writing modified pages back to disk
  3. *referenced bit*: has the page recently been referenced
     - Used by several page-swapping policies to determine which pages to swap
- Presents problem of **thrashing** when excessive page swapping occurs:
- Created the concept of a **working set** of frequently accessed pages that reside in memory to be directly accessed without incurring page faults.
  - working set changes as program execution progresses
  - the right working set can produce 90% success rates by maximising **locality of reference**
  - failure rate formula: *page faults* / *page requests*
- Requires high-speed DASD (direct access storage device) to quickly swap pages between memory and disk.

Segmented Memory Allocation

- Jobs are divided into segments of varying size:
  - capitalises on program structure (i.e., logical groupings of code)
  - program is segmented into modules of code that perform related functions
- Job segments don't need to be contiguous in memory.
- Physical memory no longer divided into frames.
- Each job now has a Segment Map Table instead of a Page Map Table:
  - Segment number
  - Segment length
  - Permissions
  - Status bit
  - Dirty bit
  - Reference bit
  - Location in physical memory
- Still uses one Job Table and one Memory Map Table for the whole system:
  - JT lists every job being processed
  - MMT monitors allocation of physical memory
- Due to dynamic segment sizes, external fragmentation returns, which requires compaction.

Segmented/Demand Paged Memory Allocation

- Combines segmentation and demand paging.
- Combines logical benefits of segmentation and physical benefits of paging:
  - Program logically divided into segments of related code
    - Maximises locality of reference
  - Segments are divided into fixed-size pages
    - Eliminates external fragmentation
- One Job Table and one Memory Map Table for the whole system.
  - MMT monitors allocation of page frames in main memory
- One Segment Map Table for each job, and one Page Map Table for each segment (Figure 1):
  - Job Table contains one entry per job with a pointer to the job's SMT
  - Segment Map Table lists details about each segment, including a pointer to its PMT
    - permissions (e.g., read, write, execute)
    - authenticated users and processes
    - status, last modified, and last reference details
  - Page Map Table lists each page, with a pointer to its page frame number
- When a job is allocated to the CPU, its SMT is loaded into memory, while its PMTs are loaded as needed.
- Address resolution requires segment and page numbers, plus the page displacement.
- Associative memory (hardware registers) stores most-recently-used pages, and is concurrently searched when a page request is issued.
- Sequence of events when accessing a location in memory as a result of a program instruction:
  - SMT and associative memory are searched to locate desired PMT
  - PMT is loaded (if not already in memory) and searched to find its frame number
    - If the page isn't in memory, page fault occurs and the page is swapped in from disk
  - Page Map and Memory Map Tables are updated with the changes
  - If PMT was found in the SMT, its reference is stored in one of the associative registers using an LRU (or other) algorithm to make one empty if they're all full

Table 1. PMT: {status,dirty,reference} bit and frame number for each page of the job

Page	Status	Dirty	Reference	Frame
0	1	1	1	5
1	1	0	0	9
2	1	0	0	7
3	1	0	1	12

Table 2. SMT: {status,dirty,reference} bit, segment size, permissions, and address in memory

Segment	Size	Status	Dirty	Reference	Permission	Address
0	350	1	1	1	X	4000
1	200	1	0	0	X	7000
2	100	0	0	0	RWX	-

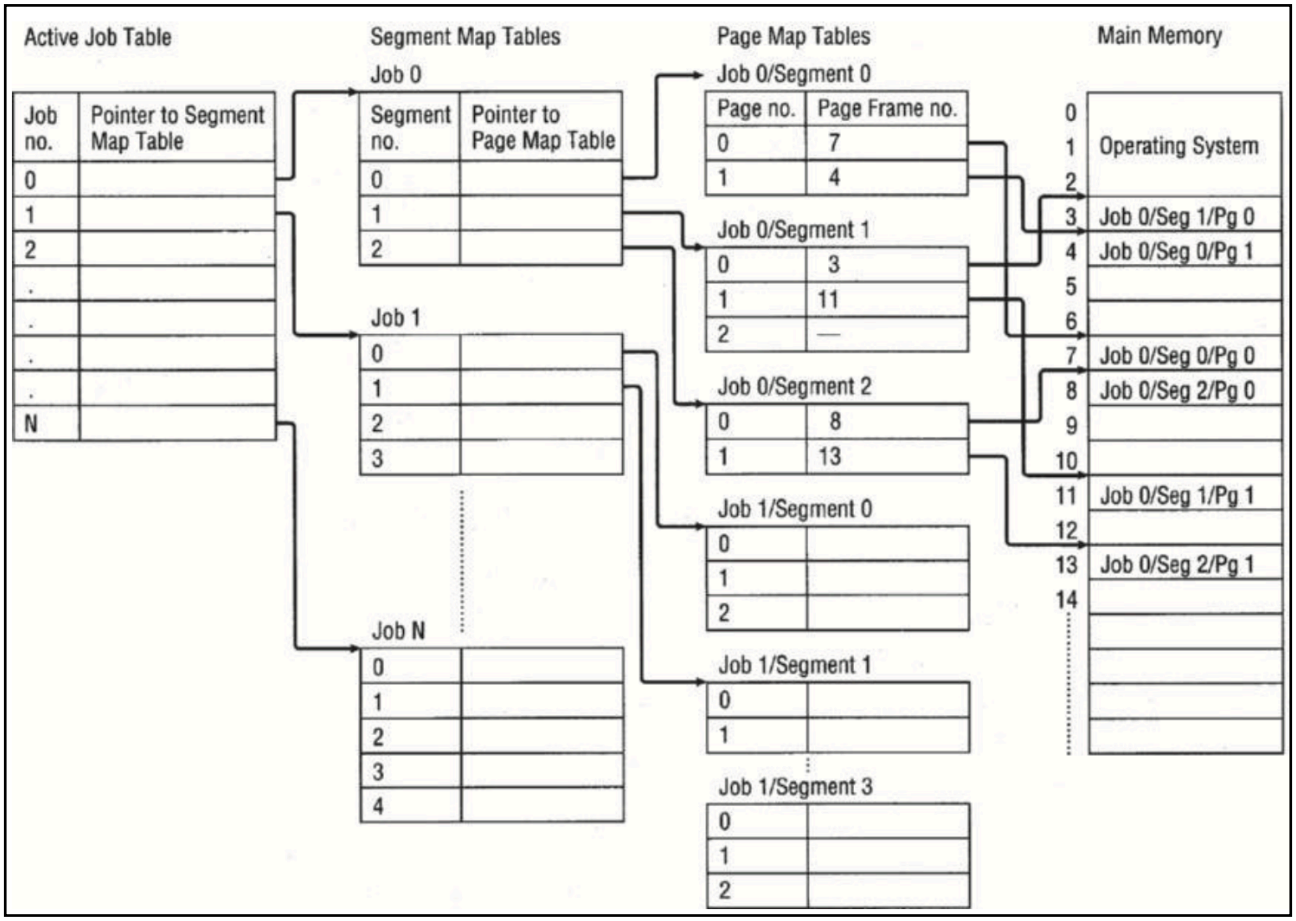

FIGURE 1. (SIMPLIFIED) SEGMENTED/DEMAND PAGED MEMORY TABLES