General Dynamic Storage Allocation Problem

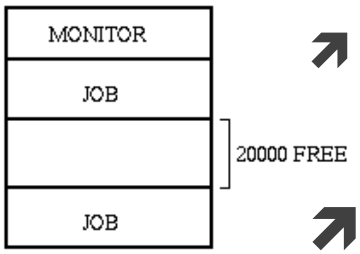
- Problem:
  - allocating contiguous blocks of storage from an initially free, single, large block
- Goals:
  - efficiency of allocation and freeing
  - minimization of fragmentation

General Dynamic Storage Allocation Problem

- **First Fit**: allocate the first block on the list that is big enough to satisfy the request.
  - this method is fast
- **Best Fit**: allocate the smallest free block bigger than the block size that is needed.
  - Tries to reduce fragmentation by minimizing left over space
- **Worst Fit**: allocate the largest block available.
  - Tries to reduce fragmentation by leaving the most usable leftover piece

General Dynamic Storage Allocation Problem

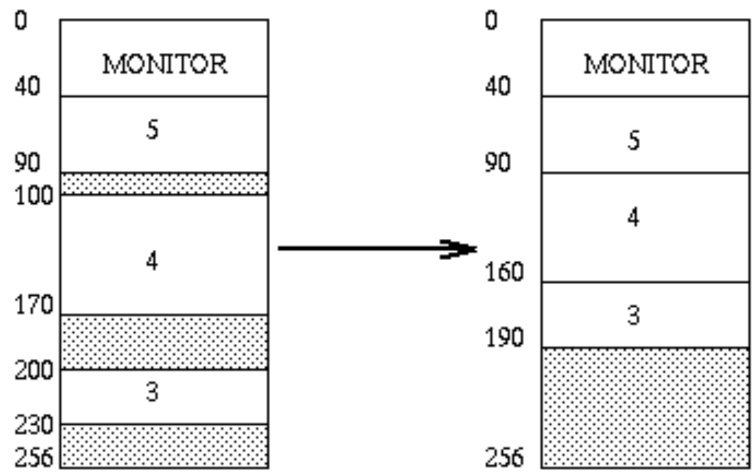
- What if a request is made for 19995 blocks of storage, and there is a 20000 block space free?



  - The overhead to keep track of the wasted five bytes is more than it is worth (more than 5 bytes needed to track a memory block)
  - allocate all 20000 blocks
  - produces a very small amount of internal fragmentation
- There are many algorithms for the general storage allocation problem
  - Many trade off one of allocation/freeing time and fragmentation for the other
  - Some of these algorithms are, or are close to O(1) in both allocation and freeing (at the expense of fragmentation)

Compaction

- With MVT, external fragmentation can get very severe.
  - e.g a small free block on both sides of every used block



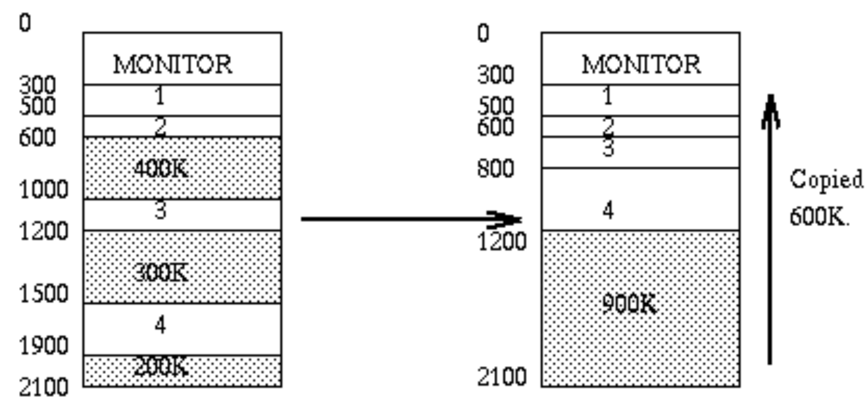
- Compaction:
  - shuffle all used blocks of memory into one, contiguous, large block

Compaction

- External fragmentation eliminated by combining all fragments into one large block
  - may be big enough to run a program while each of the smaller ones were not
- Can compaction always be done?
  What is necessary to make it possible?
  - we need dynamic relocation
    - logical addresses must be bound to physical addresses at run time so we can move the program in memory, once execution has begun
  - We need only move the program and change the base register(s)

Compaction

- Alternatives:
  - move job 4 to the slot above job 3 (copy only 400K)
  - move job 3 to below job four, leaving a large, 900K hole in the middle (copy only 200K)
  - move both jobs 3 and 4 up



- Swapping / Compaction can be used together
  - When the jobs are swapped back in, they can be placed together to leave the largest hole.

Paging

- Fragmentation is a big problem with MVT
  - The memory occupied by a program must be contiguous
- We could greatly reduce the cost of dealing with fragmentation if the memory didn't need to be contiguous
- Paging allows a logical block to be scattered throughout physical memory
  - any available chunk of memory is usable even if it isn't large enough to hold the entire object

Paging

- Requires hardware support
- The load module is broken into a series of fixed-size pages
  - size determined by hardware (2k and 4k are common)
  - all pages but the last page are the same sizes
    - e.g. a 15K program might be broken down into three 4k pages and one 3K page
- Main memory divided into frames
  - frame size = page size
  - one page fits into one frame exactly
  - last page for each object may be smaller, therefore occupies only part of one frame (remainder is wasted)

Paging

- Each logical address consists of two parts:
  - page number: = (offset from top of module) / (page size)
  - displacement within the page:

  = (offset from top of module) % (page size) (modulo)

Paging

- How do we derive these page numbers and displacements?
  - We use a page size of 2^N
  - Displacement → least significant N bits of logical address
  - Page number → remaining (most significant) bits
  - E.g. logical address

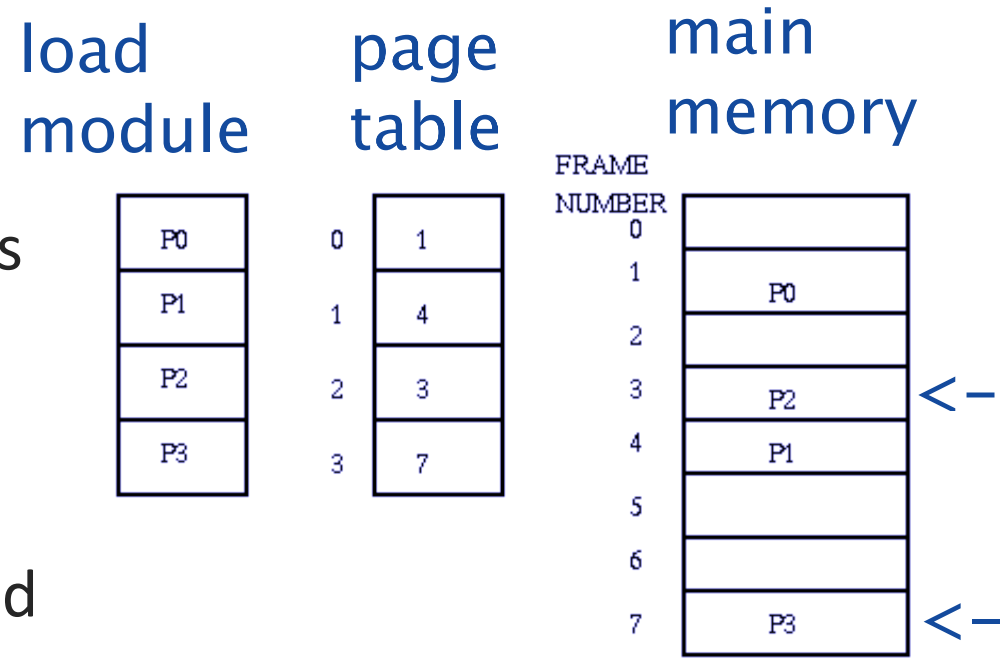
page #	displacement

n−1 0

- Why do we do it this way?
  - Can determine page # and displacement efficiently in hardware

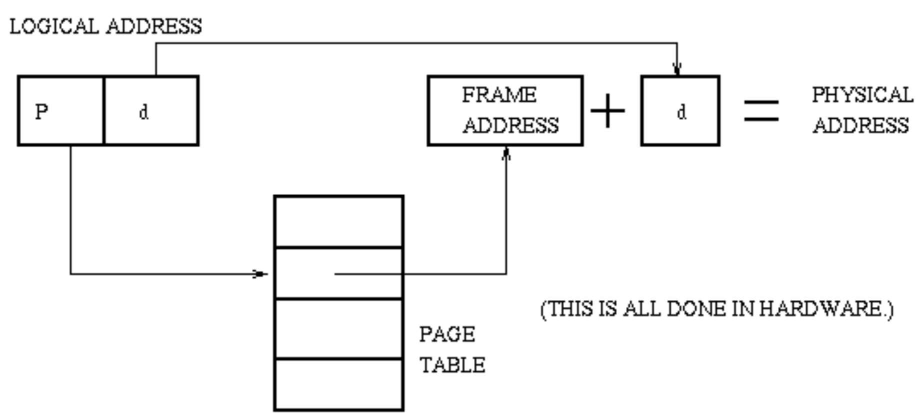
Paging

- **page table**: data structure for each process
  - entries for the base address of each page in physical memory
  - map showing where each part of the process is loaded in memory



Paging

- physical address calculation:
  - use page number as an index into the page table
    - derives the base address of the frame
  - add the displacement to derive the physical address



Paging

- When a job is to be loaded:
  - We see how many frames are required for the job
  - If enough frames are available, we load the job into the free frames, and update the page table for this job
- What about fragmentation with this paging scheme? Can we have:
  - External fragmentation? yes, but very little
  - Internal fragmentation? yes, but only for last page of each process
- The page table for each job is kept in its PCB
  - There's also a hardware register that has a pointer to the page table for the currently executing process

Page Table Implementation

- Where do we store the page table?
  - hardware registers – theres not enough X
  - use memory + cache called Translation Lookaside Buffer
- Average memory access time:

cache lookup time = 50 ns
main memory access time = 750 ns
cache hit ratio is 80%
avg memory access time =
0.8 * (50 + 750) + 0.2 * (50 + 750 + 750) = 950 ns ~ 26%
slowdown compared to non-paging lookup

Page Table Implementation

- What if we increase the cache size?

E.g hit ratio of 95%
avg memory access time =
(0.95) * (50 + 750) + 0.05 * (50 + 750 + 750) = 837.5 ns

~ 11% slowdown

- Paging allows code (pages) to be shared
  - E.g. editor can be loaded by the first user to load it, and subsequent users page table will point to the previously loaded code
  - code must be "reentrant"
- Paging separates logical (contiguous) memory and physical memory