Memory Management

Memory Management

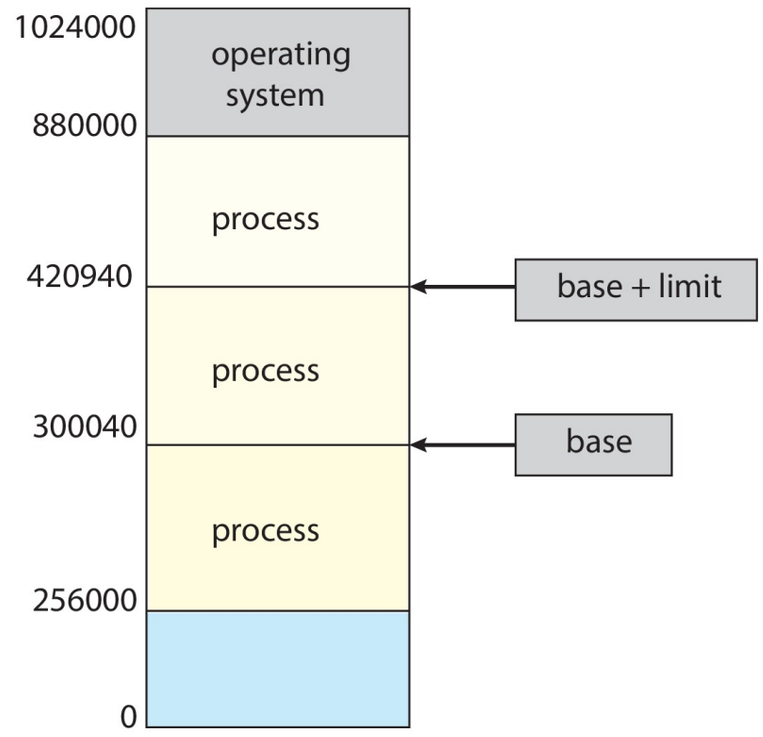
- Ch9.1 Background
- Ch9.2 Contiguous Memory Allocation
- Ch9.3 Paging
- Example: Ch9.6.1.1 Segmentation

Background

- Program must be brought (from disk) into memory and placed within a process for it to be run
- Main memory and registers are only storage CPU can access directly
- Memory unit only sees a stream of: addresses + read requests, or address + data and write requests
- Register access is done in one CPU clock (or less)
- Main memory can take many cycles, causing a **stall**
- **Cache** sits between main memory and CPU registers
- Protection of memory required to ensure correct operation

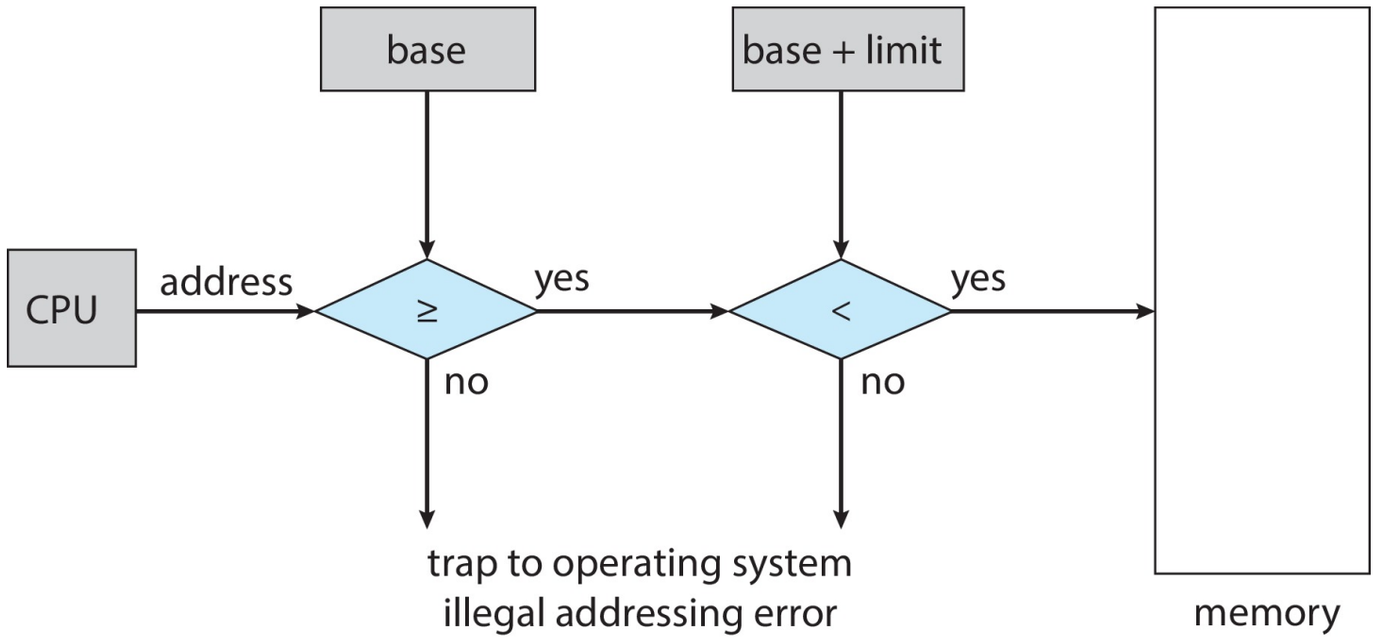
Background

- Protection
  - Need to censure that a process can access only access those addresses in it address space.
  - We can provide this protection by using a pair of **base** and **limit registers** define the logical address space of a process



Background

- Hardware Address Protection
  - CPU must check every memory access generated in user mode to be sure it is between base and limit for that user
  - The instructions to loading the base and limit registers are privileged



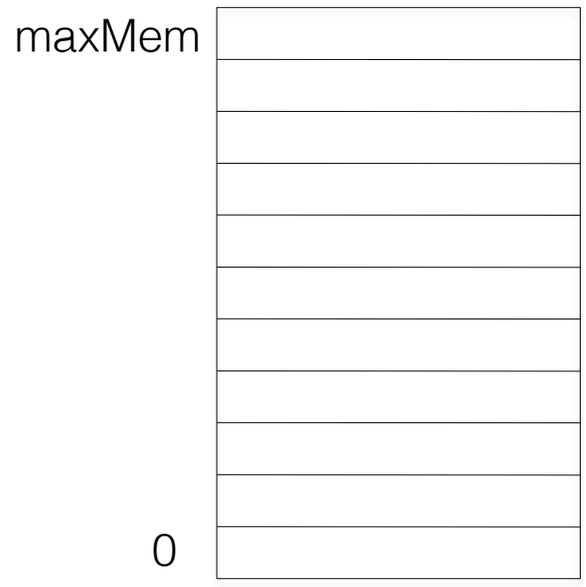
Memory

If we define memory as a place where data/code is stored there are many levels of memory:

- Processor registers
- Primary (or main) memory
  - RAM
- Secondary memory
  - slower and more permanent
  - disks
- Tertiary memory
  - archival
  - removable
- Cache memory
  - one level of memory pretending to be another
    - different levels of cache
      - at the processor
      - at devices
      - and at hosts in distributed systems

Main Memory

- All processes need main memory.
- Instructions and data being worked on are brought into the processor from memory and sent back to memory.
- Traditional view:
  - Addresses are numbers: 0 – maxMem
  - An address goes out of the processor to the address bus of memory.



Address Binding

- We can make the connection between code and data and their addresses at different times:
- **Compile time**
  - Need to know exactly where the value will be stored.
  - Fixed addresses.
  - Not very flexible.
- **Load time**
  - Object modules need tables of offsets.
  - e.g. variable x is 24 bytes from the start of the module.
  - The mapping is done as the module is loaded.
  - Can also reference other modules – linking.
  - More flexible but can't be changed after loading.

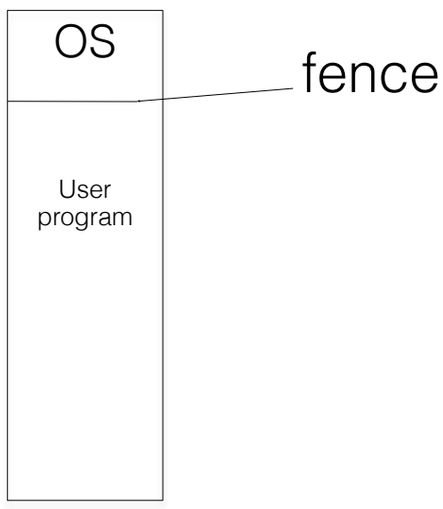
  - Dynamic loading – don't load unless called
  - Dynamic linking similar (useful with shared libraries) – may already be in memory – if shared between processes needs OS help
- **Run time**
  - Mapping to final address maintained on the go by hardware.
  - Commonly this is used in conjunction with the above techniques.

Memory Spaces

Even in simple systems we would like a way of protecting OS memory from our program and enabling programs larger than memory to run.

Split memory

- Can protect with a single fence register.
- Alternatively if the OS is in ROM its code and constant data is safe from overwriting.



Overlays

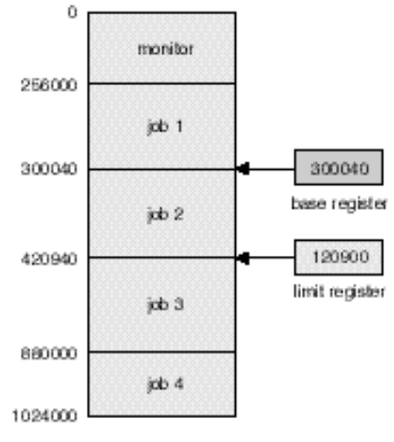
Load needed instructions over the top of ones not needed any more.

Dividing Memory

- With linking loaders we can have multiple programs in memory simultaneously.
- We also want protection.
- In the history section we saw separate partitions and base and limit registers.
- Both require contiguous memory.
- The algorithm for base and limit registers is simple.
- If the address is less than the base or greater than the base + limit – 1 then we have an access violation.
- The base and limit registers are loaded for each process.

We need a lot more memory, maybe more than we have.

- So we could use overlays in each allocated area.



Two Different Addresses

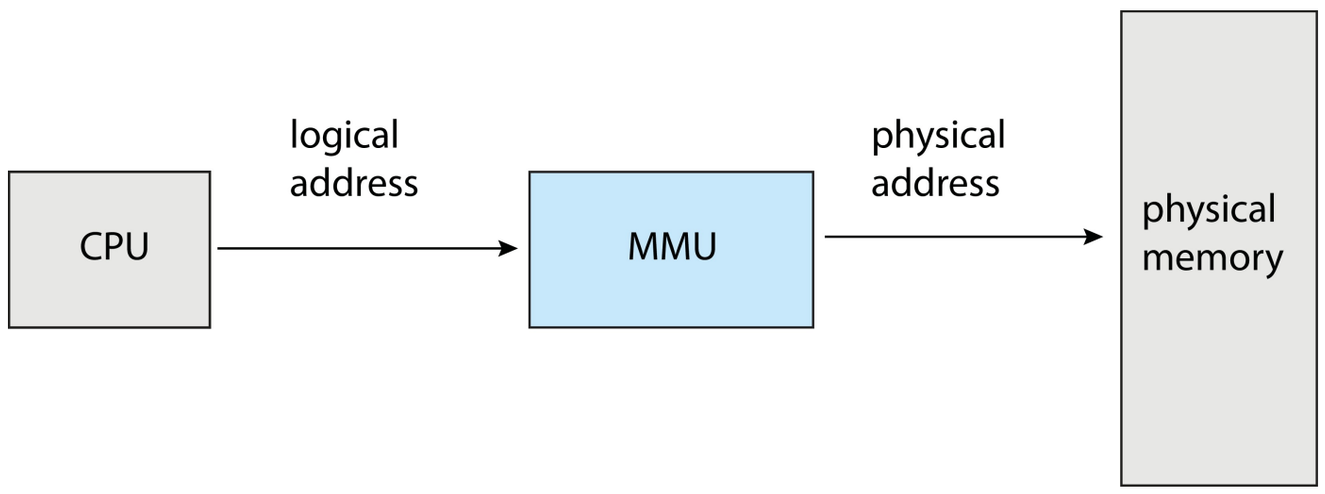
- If we change our base and limit system to produce the address by adding the base register (now called a relocation register) to each address we can make all processes start at address 0.

This is a huge conceptual change.

- We now have two types of addresses.
  - The **logical** (virtual) address coming out of the process, produced by the CPU
  - The **physical** (real) address used on the address bus to memory.
- We still have contiguous memory for each process but a process can now be positioned anywhere in physical memory without its addresses needing to be changed.
- We can even move a process around, just copy the memory to the new place and change the relocation register.
- This is useful if we want to defragment memory to give one large free area.
- Have to be careful if moving data (e.g. from a device) into the process' memory space.

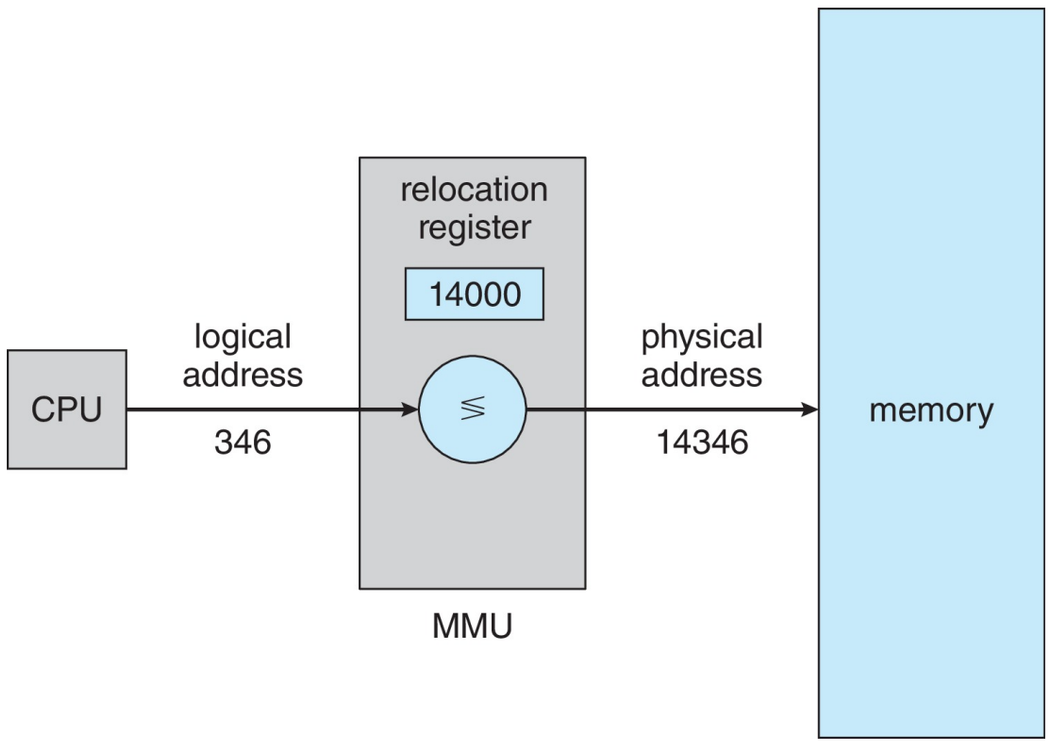
Memory-Management Unit (MMU)

- Hardware device that at run time maps virtual to physical address
- Many methods possible, covered in the rest of this chapter



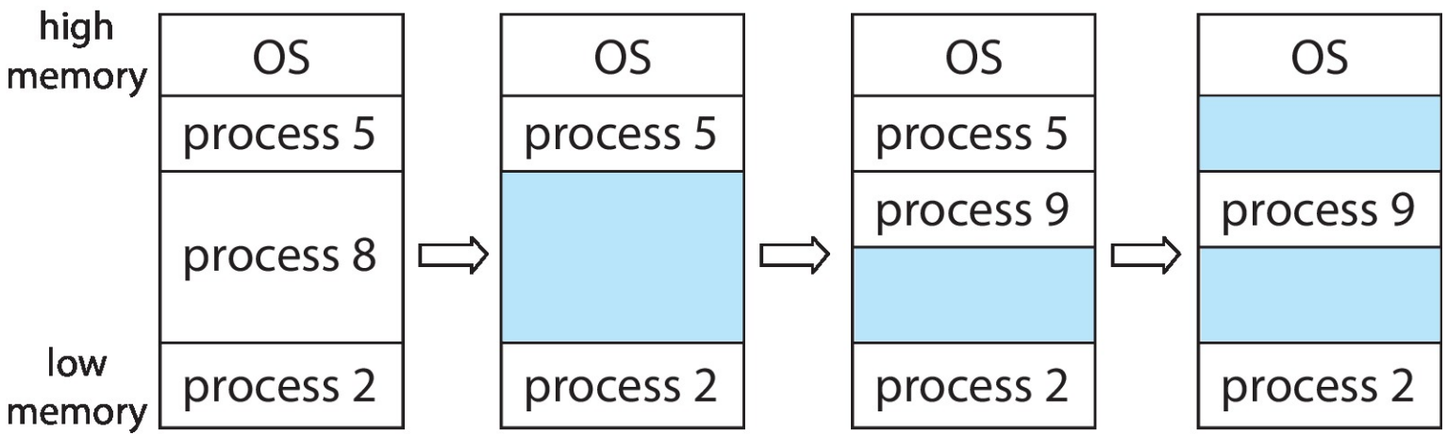
Memory-Management Unit (MMU)

- Consider simple scheme. which is a generalization of the base-register scheme.
- The base register now called relocation register
- The value in the relocation register is added to every address generated by a user process at the time it is sent to memory



Contiguous Allocation

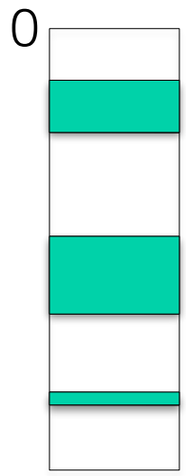
- Main memory must support both OS and user processes
- Limited resource, must allocate efficiently
- Contiguous allocation is one early method
- Main memory usually into two **partitions**:
  - Resident operating system, usually held in low memory with interrupt vector
- User processes then held in high memory
- Each process contained in single contiguous section of memory



Split Memory into Smaller Chunks

- Rather than moving memory around to make big enough spaces for processes we could have more than one section of physical memory accessed by the same process.
- We need either a number of base-limit registers or a table of the information.

Chunk	base	limit
0	1024	1024
1	8192	512
2	4096	2048



The process sees 3.5K of contiguous memory.

Two Approaches

This technique evolved in two directions.

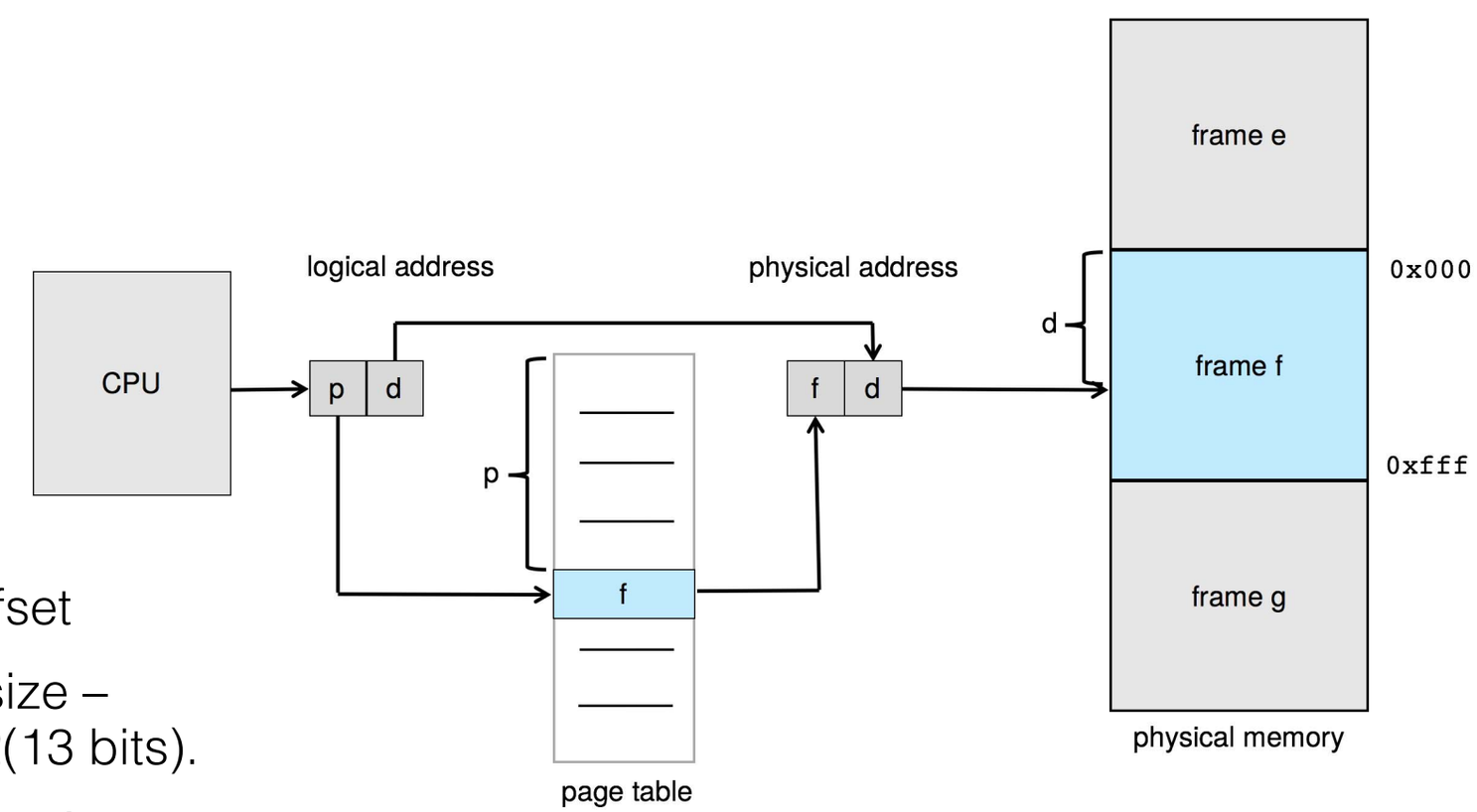
1. Same sized chunks – **pages**
2. Variable sized chunks – **segments**

- Both have advantages and disadvantages.
- Both use Memory Management Units (MMUs) in hardware to do the translation between the logical and the physical address.
- Rather than doing a tedious calculation (where is logical address 2000 on the previous slide?) to find what chunk an address is in, we just split the address into two parts.
- Then the translation is much simpler and looks very similar in both paged and segmented systems.

Paged System Address Translation

Logical address is divided into:

- **Page number (p)** – index into a **page table** which contains base address of each page in physical memory
- **Page offset (d)** – added to base address to get the physical address.



- Constant number of bits for the offset
- Pages are always powers of 2 in size – usually from 2048(11 bits) to 8192(13 bits).
- Some systems allow variable sizes of pages

Exam Type Questions

- Q: Why are page sizes always powers of 2?
- It is most efficient to break the address into X page bits and Y offset bits
- Because each bit position represents a power of 2, splitting an address between bits results in a page size that is a power of 2.

- Consider a logical address space of 32 pages of 2048 words each, mapped onto a physical memory of 64 frames.
- Q: How many bits are there in the logical address?
- Addressing within a 2048-word page requires 11 bits: 2^11
- Since the logical address space consists of 32 = 2^5 pages, the logical addresses must be 11+5 = 16 bits

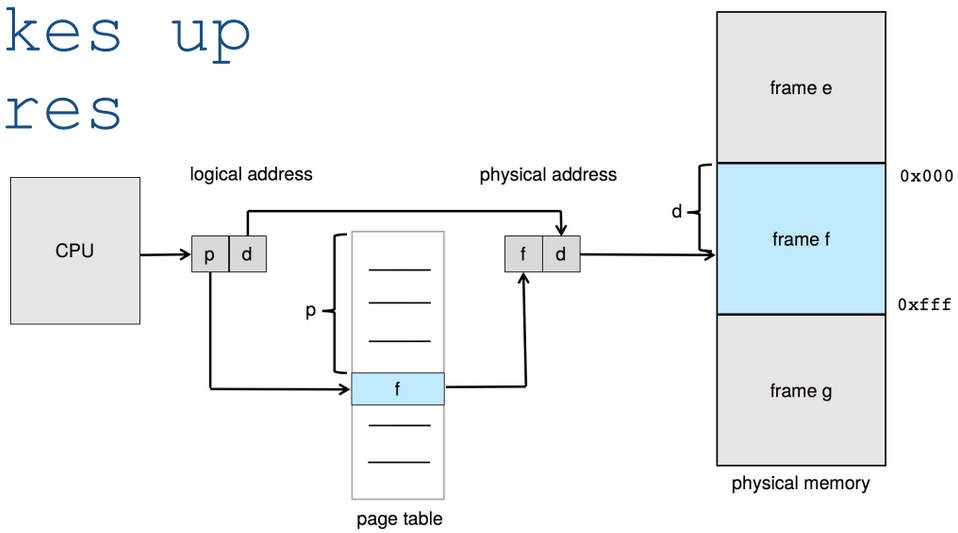
Exam Type Question

- In a system with 64 bit addresses and pages of size 4K (2^{12}) bytes, how many pages can a process access?

```
2^64/2^12 = 2^52 pages
```

- If each page is represented in the page table by 8 bytes, how big would the page table have to be to hold entries for a process which used all its possible memory?

```
We need a page table takes up
2^52 * 2^3 = 2^55 memoires
```



Frames and Pages

- The textbook makes a distinction between pages and frames.
- A frame is a page sized chunk of physical memory that can be allocated to a page from a process.
- A page is a frame sized chunk of logical memory that fits in a frame.
- It is common to refer to both simply as pages (physical and logical).

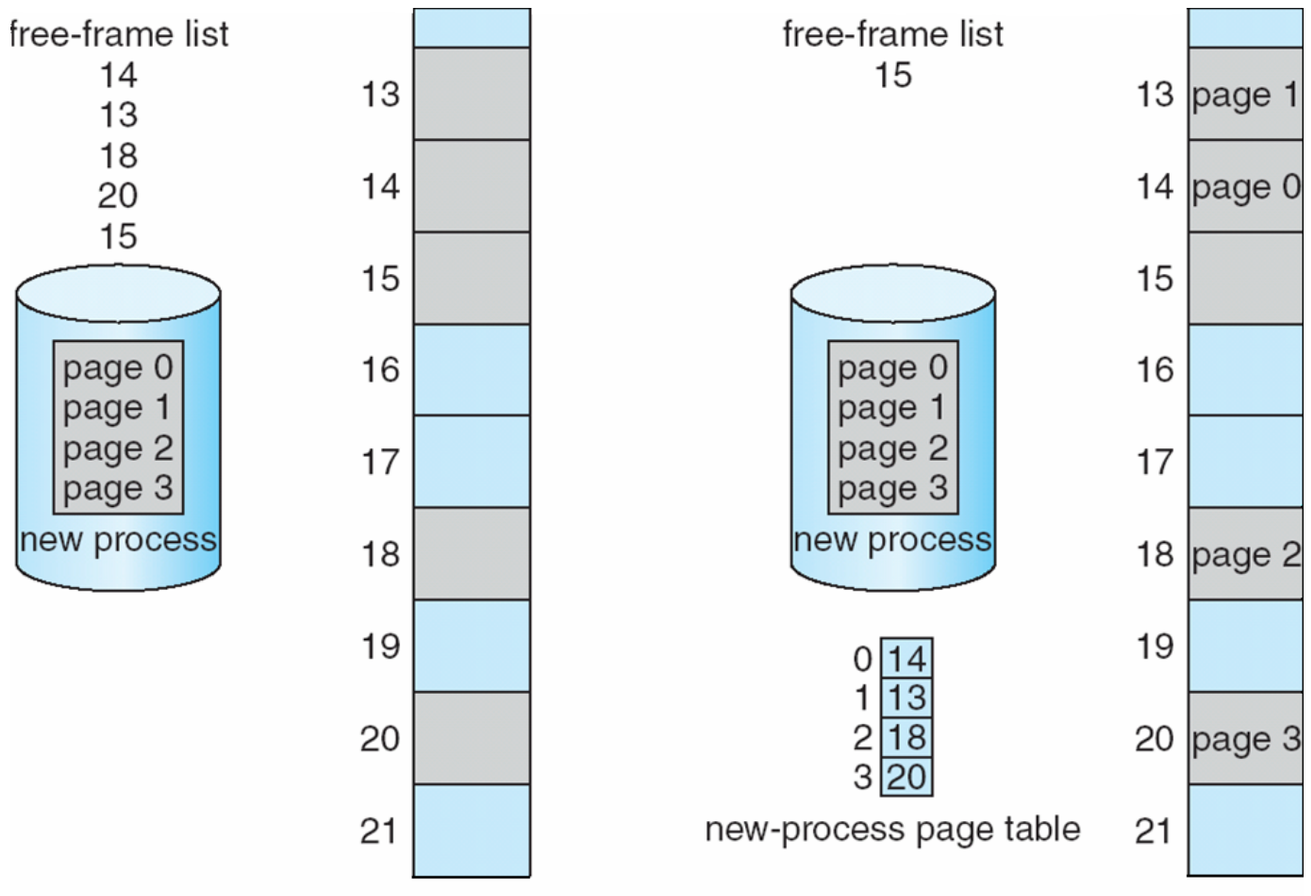
Fragmentation

- No external fragmentation between pages.
- Internal fragmentation in any pages that are not completely used.
- 50-percent rule: Simulation shows that for every 2N allocated block, N blocks are lost due to fragmentation (i.e., 1/3 of memory space is wasted)
- Small pages save space but require larger page tables.
- In x64 machines it is possible to use page sizes of 1GB (but normally we work in 4KB pages).

- See http://wiki.osdev.org/Page_Tables

Tables

- Each process has its own page table.
- And commonly the OS has a frame table with one entry for each frame showing whether it is free or not and which process is currently using it.



Before Allocation

After Allocation

Question

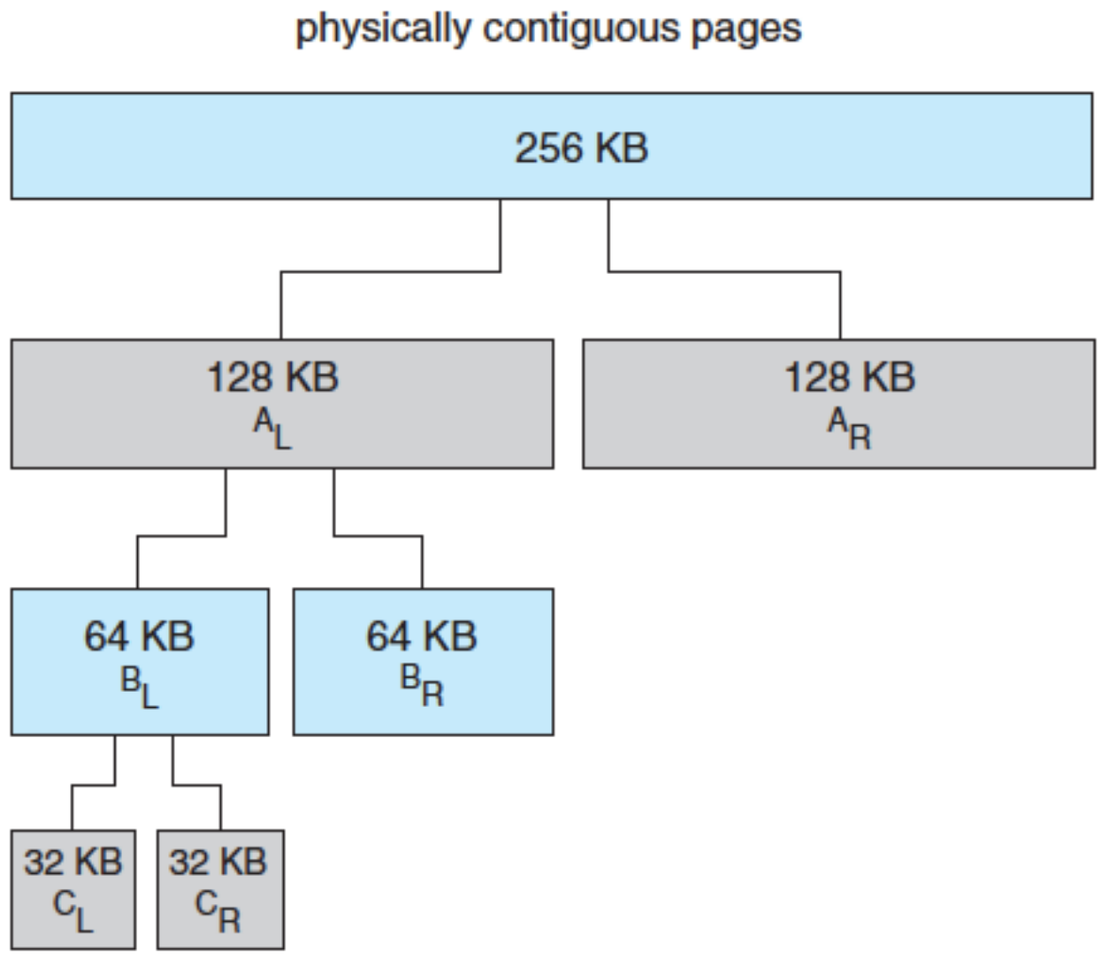
- Reasons for small pages
  - less internal segmentation

- Reasons for large pages
  - smaller page tables
  - few page fault

- Page sizes are growing because …

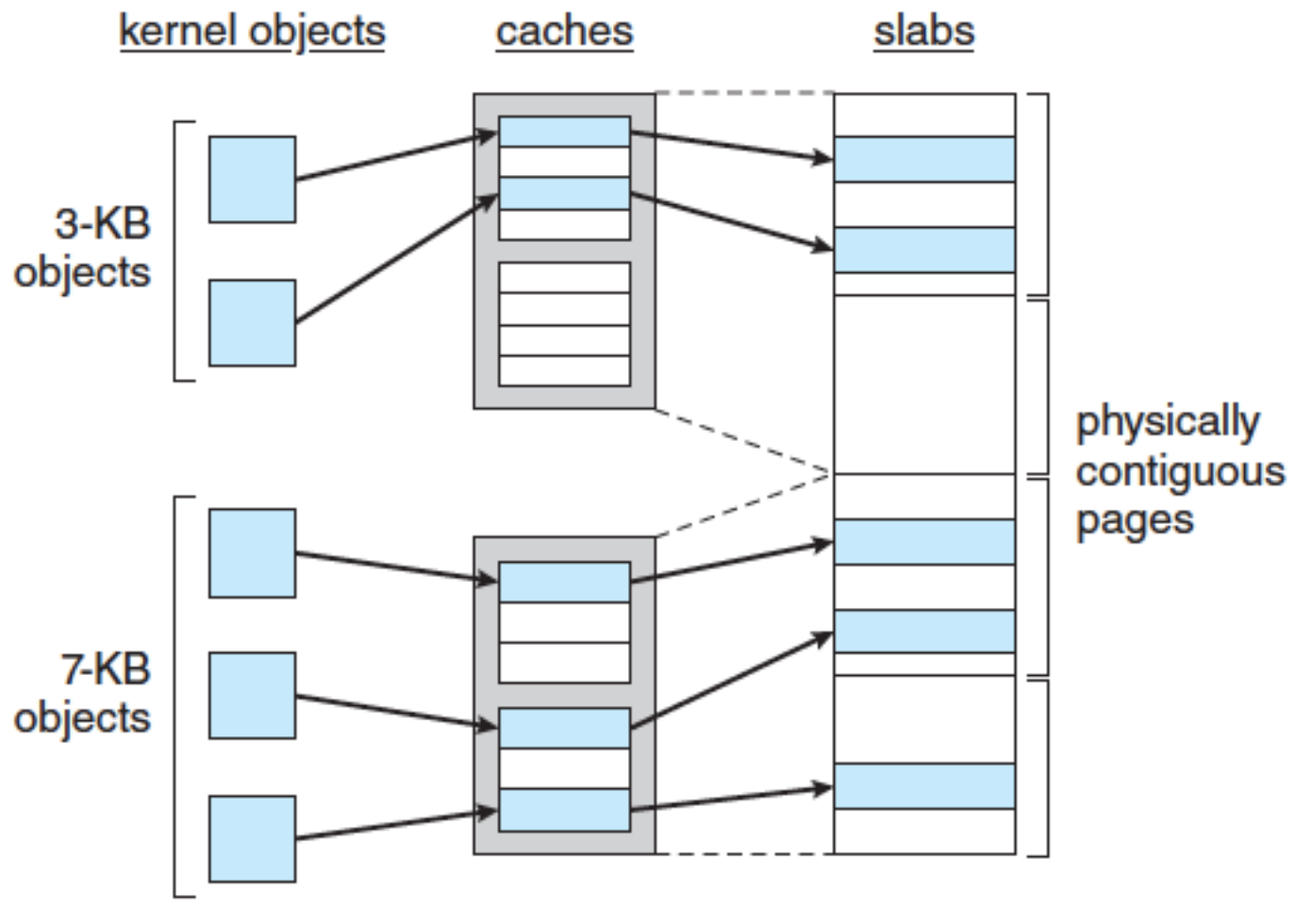
Page Allocator

- Uses the **Buddy** algorithm (Ch10.8.1)



Page Allocator

- Uses the **Slab** algorithm (Ch10.8.2)



Buddy and Slab

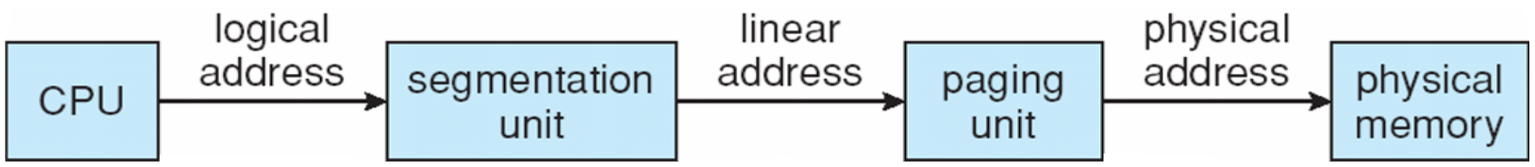
- Buddy Allocator: Can lead to internal fragmentation
  - If you want 65K you get 128K
  - End up with lots of little allocations
- Slab Allocator: Used for common data structures e.g. file information, task information (PCB)
  - A slab allocator allocates data structures of the same size
  - Slabs can be **empty**, **partially**, or **full**
  - Request a file data structure from an empty or partially full slab
  - Return it when no longer needed - available for future allocation
  - Extra slabs are allocated when needed using the buddy algorithm

Different Sized Chunks

- Rather than constant sized pages we could design our hardware to work with variable sized chunks – these are known as segments. (Not to be confused with variable sized pages.)

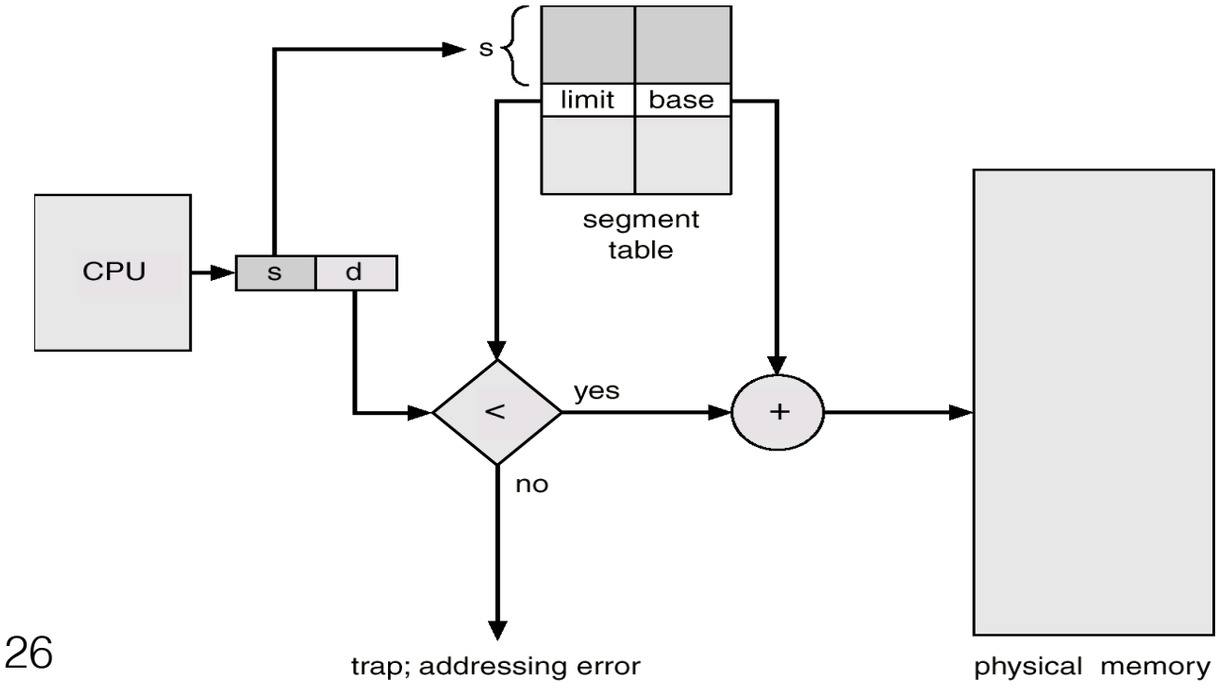
Memory model

- How memory appears to be organised to the program (and programmer) is sometimes referred to as the memory model.
- A **segmented memory model** is when we look at memory as a group of logical units all with addresses starting at zero.
- Processes can have lots of segments
  - functions
  - global variables
  - activation records
  - large data structures (arrays)
  - objects



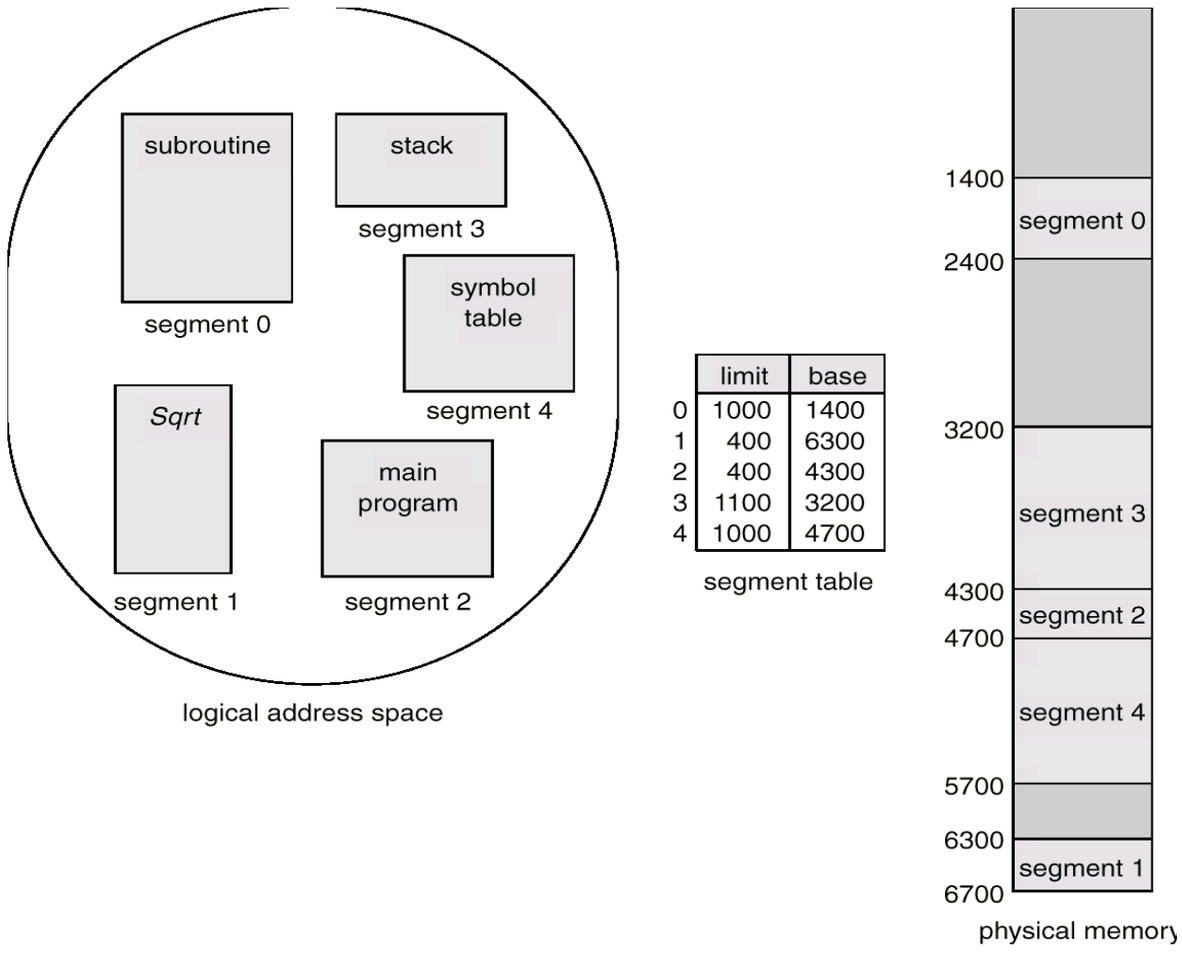
Segments

- These logical units fit nicely into hardware specified segments where different amounts of memory can be allocated in one chunk.
- Segments are contiguous blocks of memory.
- We will get problems of external fragmentation.
- Our memory addresses become : <segment name, displacement>.
- Translation process looks very much like paging, except there is a length associated with each segment. We have a segment descriptor table rather than a page table.
- The physical address is the result of an addition rather than a concatenation as it is in a paged system.



Example

- Ideally segments should be able to cover all of the addressable memory – this could mean that logical addresses might have more bits than physical addresses.
- Some segmented memory systems restrict segment sizes to prevent this.



Allocation Strategies

- Segments have no internal fragmentation – we only allocate the amount of space we want.
- But we get external fragmentation.
- We have seen the allocation strategies before:
  - First fit
  - Next fit (first fit but starting from where we were up to)
  - Best fit
  - Worst fit
- We can defragment memory if we want to find large enough chunks. Faster than defragmenting disks.

How Much Space in the Holes

- Knuth's 50% rule
- If there are n segments there are n/2 holes.

- Each segment is either below another segment or below a hole.
- We always combine adjacent holes.

Each segment is released independently – so in a steady state system the space above each segment will alternate between being used and being free. 50% of the time there will be a hole above each segment.

If the average size of a hole is the same as the average size of a segment we need about 1/3 of the memory free to keep this system running.

Before Next Time

Read from the textbook

- Ch9.5 Swapping
- Ch9.3 Paging
- Ch9.4 Structure of the Page Table
- Segmentation with Paging
- Ch10.2 Demand Paging
- Ch20.6.2 Linux Virtual Memory