Basics

- Main Memory = Memory Unit
- array of byte/words contain its own address
- each incoming address is stored in the memory address register of the memory unit
- Called the "Main" memory by contrast with registers, caches, which are all managed 100% by the hardware
- Processes share the main memory, therefore the OS must manage the main memory
- CPU only works with registers but it can issue addresses that correspond to words of the main memory

Early System

- each process is allocated a contiguous zone of physical memory

Address Binding

def allocates a physical memory location to a logical pointer by associating a physical address to a logical address

Absolute Addressing

the binary executable contains physical addresses

Issue of Absolute Addressing

With absolute addressing a program must be loaded exactly at the same place into memory each time we run it

Therefore we may not be able to run a program because another program is running and encroaches on the address range!

Corollary

cannot run multiple instances of a single program

Relative Addressing

- Assume the address space starts at some BASE address, and computes all addresses as an offset from the BASE
- The code is now completely relocatable: Only the BASE needs to be determined before running it
- same program can be run anywhere in memory
- Multiple instances can run, each with a different BASE address

RAM Virtualization

- All addresses in the process address space are expressed as an offset relative to the base value

Memory Virtualization

- each program instance has the illusion that it's alone in RAM and that its address space starts at address 0
- Memory Protection
  - program doesn't need to know anything about other programs
  - good because writing code you don't know what other programs will be running anyway
- **Bottom Line** A program references a logical address space, which corresponds to a physical address space in the memory

Virtualizing The Process Address Space

- Some hardware component needs to translate virtual addresses into physical addresses
- Address translation happens very frequently, thus base address is accessed very frequently
- and offsets are added to the base address very frequently
- use a limit register that stores the largest possible logical address

Memory Management Unit

- specialized circuit between the CPU and the memory and integrated with the CPU nowdays

Segmentation

- having a single contiguous segment is wasteful
- Segmentation
  to avoid waste by breaking up the address space into pieces
- each piece has its own base/limit register
- logical address space is now a collection of segments
- compiler/language interpreter handles the segments and the logical addresses are built appropriately
- Segments used by C compiler
  text, data, heap, stacks, C library

Segment Table

- one entry per segment number is used to keep track of segments
- for each entry stores the base (starting address of the segment) and limit (length of segment)
- segment table is stored in memory
  - Segment Table Base Register
    points to the segment table address
  - segment table length register
    gives the length of the segment table; easy to detect an invalid segment offset
  - registers are saved/stored at each context switch

Swapping

def Moving processes back and forth between main memory and the disk
Concept
a process is swapped back in, it may be put into the same physical memory space or not
Swapping and DMA

- a process can be kicked out from RAM to disk by the OS at any time
- DMA controller may have no idea and happily continue to write data (into some other process' address space, which has replaced that of the one that was swapped out!)

Swapping Bad

- The disk is slow
- to cope with slow disks have been used
  - OS could swap in/out only processes with small address space
  - can dedicate disk/partition to swapping
- approach is to just not swap
  - Swapping should be an exceptional occurrence
  - Swapping is now often disabled

Memory Allocation

- Where should the processes be placed in memory?
  The kernel must keep a list of available memory regions or "holes"
  When a process arrives, before scheduling it, it is placed in a "I need memory" input queue
  Kernel must make the decision: pick a process from the input queue or pick a hole in which the process will be placed
- problem is known as the dynamic stroage allocation problem

Memory Allocation Strategies

1. Which process should be picked?
   - First Come First Serve
     easy, fast to compute, may delay small processes
   - allow smaller processes to jump ahead
     slower to compute favors small processes

- Limit the jumping ahead
- Look ahead

2. Which hole should be picked?
   - First Fit
     pick the first hole that is big enough
   - Best fit
     pick the smallest hold that is big enough
   - Worst fit
     pick the biggest hole
3. How should the picked process be placed in the picked hole?

Memory Allocation

- FCFS + First Fit + Top
- Jump Ahead + Worst Fit + Bottom
- above are heuristics that produce decent solutions
- same story to CPU scheduling

External Fragmentation

- Goal: hold as many processes as possible in memory
- External Fragmentation
  defined as number of holes
- For a given amount of available RAM, we're always happier with a single large hole than with several smaller holes
- processes terminate whenever they want to, we cannot avoid external fragmentation

Internal Fragmentation

- Do we want to keep track of tiny holes?
  no
- an OS would allocate slabs that are multiples of some "block size"
- Downside: a process may then not use the whole slab and some space is wasted

Smaller blocks
lower internal fragmentation, but more blocks to keep track of
Larger blocks
higher internal fragmentation, but fewer blocks to keep track of

Making Address Space Smaller

Smaller Address Space

- always a good idea as it is good for swapping
- don't swap as often (because if address spaces are small, then RAM looks bigger), not as slow to swap (because reading/writing a smaller address space from/to disk is faster)
- Three Common Place Techniques
    1. Dynamic Memory Allocation
       Ask programs to tell the OS exactly how much memory they need when they need it (malloc, new) so that we don't always allocate the maximum allowed RAM to each process
    2. Dynamic Loading
    3. Dynamic Linking

Dynamic Loading

- only load code/text when it's needed
- enacted by code written by the programmer for this purpose
- OS is not involved, although it provides tools to make dynamic loading possible

Dynamic/Static Linking

- Static Linking is the historical way of reusing code
- dynamic loading, but the OS loads the code automatically and different running programs can share the code
- code is shared in shared libraries:

Shared Libraries

- dynamic linking is enabled, the linker just puts a stub in the binary for each shared library routine reference
- a stub is a piece of that
    - checks whether the routine is loaded in memory