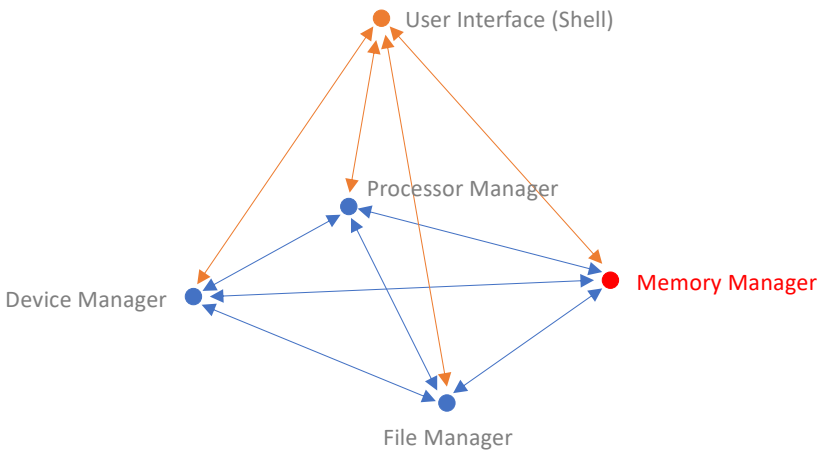
COMP124

Computer Systems

18 | Memory Manager | Addresses | Segmentation

Dr Stuart Thomason

Memory Manager



Memory Manager

- Manages the main memory (RAM) of the system
- Ensures each process gets access to the memory it needs

- Memory manager performs various tasks for the operating system…
  - Preserves and protects the space in memory occupied by the OS itself
  - Checks validity of each request for memory space
  - Allocates free areas of memory for valid requests
  - Keeps track of which users and processes are using which areas of memory
  - Deallocates memory when no longer needed
  - Provides standard system calls so processes can request memory space
  - Implements segmentation and paging models via virtual addressing
  - Manages swap space according to page swapping policy

Memory Addressing

- The memory inside a system (RAM) can be viewed as a linear sequence of bytes
  - Sometimes called the linear store or flat model
  - Addresses range from zero up to the number of bytes available
  - Memory is byte addressable

- Addresses will be stored in CPU registers during process execution
  - So the address space is limited by the bit length of the CPU
  - A 32-bit CPU can address 4,294,967,296 individual bytes
  - Which means it can only support up to 4 GB of memory
  - Some addresses will be used to store the code (kernel image) for the OS itself
  - Other addresses will be used for memory-mapped devices

Process Memory

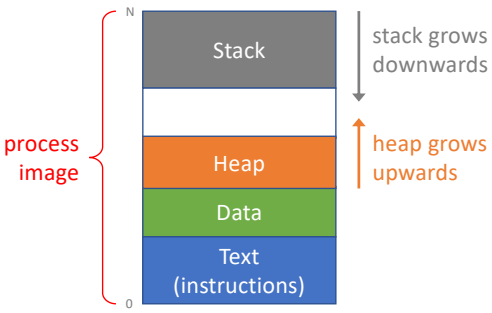
- A program exists on disk as a binary executable file
  - Stores all the program instructions (opcodes/operands) as a sequence of bytes
  - Compiled to target a specific combination of CPU architecture and operating system

- When a program is run (via the shell or process fork)…
  - Binary image is loaded into main memory
  - Extra memory is reserved for variables and the stack (with spare room to grow)
  - This combined memory footprint is the process image
  - Instruction pointer is set to the first byte of the image

- Jumps (to labels) are replaced by real memory addresses
  - Point to a physical location within the process image (can be fixed or relative)
  - Instruction pointer is set according to this real address

Stack and Heap

- We know from a previous lecture that…
  - The stack grows downwards from the top of the process address space
  - The process instructions (often called text) start at the bottom of the image
- The memory footprint of a process includes two other areas
  - Data – memory allocated for variables and objects declared in high-level code
  - Heap – memory available for the programmer to reserve (eg. Java new keyword)



Compile-Time Address Binding

- Loops and jump locations are turned into fixed memory addresses during compilation
- With compile-time binding, a program is assumed to always occupy the same area of memory every time it is executed

```
start: mov eax, 2
       mul 2
       jmp start
```

- The jump location (start label) is turned into a memory address when compiled
  - For example, at compile time this is turned into the memory address 1000
  - This code must always be loaded from address 1000 onwards so the jump works
- This has obvious disadvantages in modern operating systems
  - Impossible to move the program to another memory location
  - Prevents OS from making most efficient use of memory
  - What if another process is also compiled to use memory address 1000?

Load-Time Address Binding

- Loops and jump locations are turned into relative addresses during compilation
- With load-time binding, a program can be loaded anywhere in memory

```
start: mov eax, 2
       mul 2
       jmp start
```

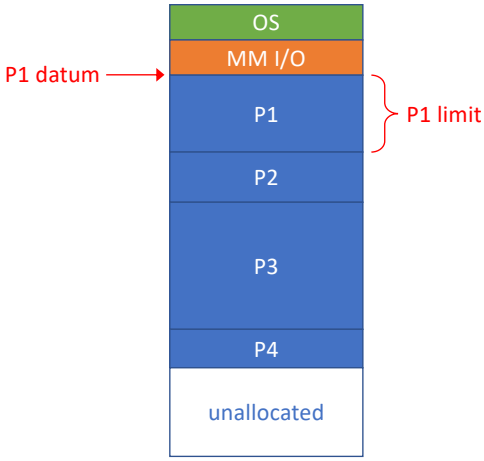
- The jump location (start label) is encoded as a relative address when compiled
  - For example, at compile time this jump is encoded as JUMP −8
    (in our simplified approach where each instruction uses 4 bytes)
  - Actual address will be inserted into code when the process is loaded into memory
- Offers some advantages to the memory manager
  - Processes can be loaded anywhere in the address space
  - Two processes will never want to jump to the same address

Dynamic (Run-Time) Address Binding

- Loops and jump locations are turned into relative addresses during compilation
- But all processes have their own address space that starts at location zero
  - For example, a process takes up N bytes of memory when loaded
  - Its address space will be numbered from 0 to N regardless of where it really is
  - These are logical (virtual) addresses in a logical address space
- Virtual addresses are mapped to the real physical addresses at run-time by MMU
  - CPU has a relocation register that holds real address of start of process
  - Logical addresses are mapped relative to this base value
- Address mapping depends on the type of address binding used
  - Compile-time binding means that logical and physical addresses are the same
  - Load-time binding means that logical and physical addresses are the same
  - Run-time binding means that logical and physical addresses are different
- Modern operating systems use run-time (dynamic) address binding

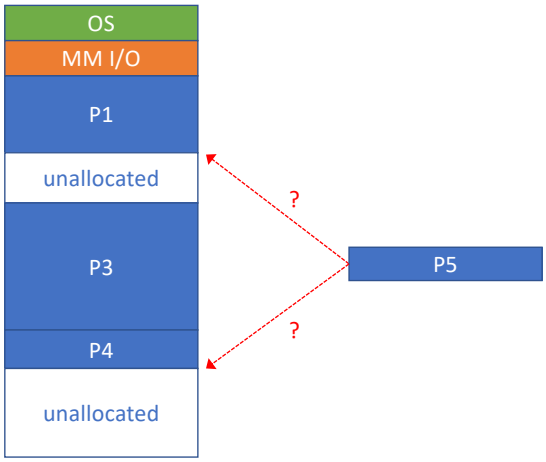
Simple Memory Management

- Processes are loaded into contiguous memory locations
- Each process has a base or datum (where it starts) and a limit (its length)



Choosing Free Space

- When a process terminates, its memory space is deallocated and becomes free again
- Memory manager needs to decide where to load new processes



Memory Allocation Issues

- The memory space will become fragmented with small unallocated gaps
  - There might not be a big enough free space for a large process to fit into
  - Some processes might suffer starvation if they cannot be loaded and executed

- Memory manager is responsible for choosing free space for each process
  - First fit – just choose the first free space that's big enough
  - Best fit – choose the smallest free space that's big enough
  - Worst fit – deliberately choose the biggest area of free space

- Fragmentation can become severe with the first fit policy
  - If amount of memory currently allocated is N bytes
  - Then the amount of unusable memory (due to fragmentation) is N/2 bytes
  - Known as the 50% rule

Memory Allocation Issues

- Memory manager can move processes to defragment space and make more room
  - Only works if processes are compiled to use dynamic binding
  - Must update the relocation register so logical addresses map correctly
  - Processes must be paused while the move takes place
  - Will require some CPU time to carry out the moves

- Eventually the system will run out of memory
  - Only a finite amount of free space to support a finite number of processes
  - Will not be able to run all the processes it needs to

- Contiguous linear memory model imposes limitations on program structure
  - Cannot share subroutines between processes
  - Memory allocation doesn't map onto high-level program structures

Swapping

- The system can execute more processes than will fit in physical memory
  - Processor manager (scheduler) works with memory manager
  - Process images can be stored on disk when not being executed
  - A process (eg. blocked state) can be swapped out to disk to make more room
  - This adds to the complexity and time needed for a context switch

- Operating system reserves part of a disk (or an entire disk partition) as swap space
  - Allows RAM to be freed up for processes that need it
  - Increases system stability because there will always be room for new processes

- Accessing the disk is much slower than accessing main memory
  - A system with a lot of processes in swap space will run more slowly
  - As memory fills up, the disk will "thrash" as processes are swapped in and out

Linking

- Programs make use of third-party library routines (via import, include, etc.)
- The code for these routines is stored in library images

- Static linking…
  - Copies of the library code are added to the final program image
  - Can result in large images if lots of libraries are used
  - Wasteful of both disk storage and memory space
- Dynamic linking…
  - A small stub is included in the final program image
  - Tells the memory manager where to find library code (if already in memory)
  - Allows re-entrant library code to be shared between processes
  - Keeps disk image small and makes better use of main memory
  - Operating system must be designed to support this (eg. Windows DLL files)

Dynamic Loading

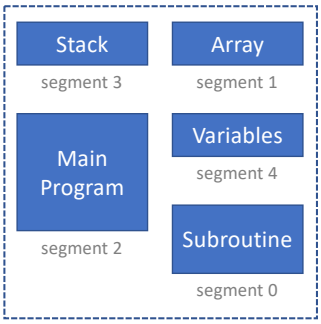
- An alternative to dynamic linking is dynamic loading
  - Dynamic linking is controlled by the operating system (memory manager)
  - Dynamic loading is controlled by the programmer
  - A process can use both techniques at the same time

- Allows process to execute without required libraries being in memory
  - Programmatically attach a library to the process image while running
  - Call the relevant subroutine from the library
  - Then detach the library code from the process image

- Library code must be compiled with metadata to support dynamic loading
- Program must be linked (at compile time) to library code that implements dynamic loading (eg. C programs must include the dlfcn.h library and call dlopen subroutine)

Segmentation

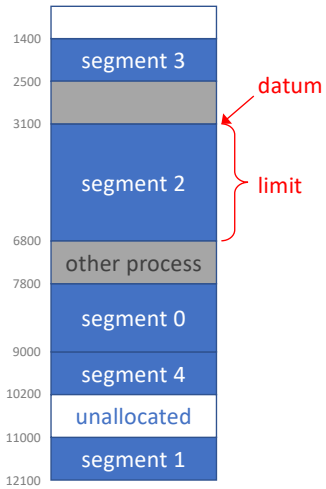
- Segmentation allows individual parts of a process to occupy different memory locations
  - Not constrained to loading into contiguous addresses
  - Segments can slot into smaller free regions of memory
  - Process has a segment table (also stored in memory)
  - Table address stored in segment table base register



Logical Address Space

Seg	Limit	Datum
0	1200	7800
1	1100	11000
2	3700	3100
3	1100	1400
4	1200	9000

Segment Table



Physical Memory

Segmentation Advantages

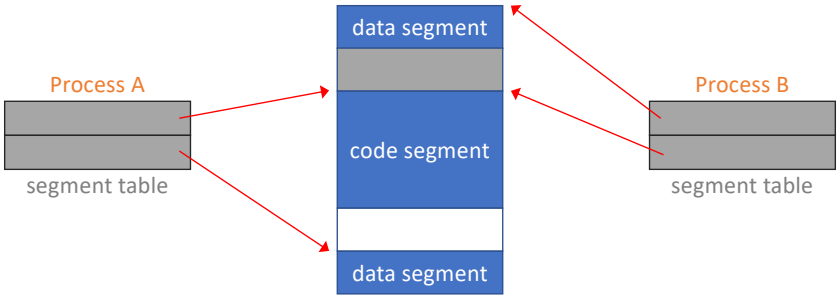
- Memory allocation reflects the high-level program structure
  - Segment for each subroutine, data structure, main program, stack, etc.
  - Compiler generates separate segments within the program image

- Memory protection is enhanced
  - Array bounds checking can be performed easily
  - Code segments can be protected from being overwritten by data segments
  - Data segments can be prevented from being executed
  - Hardware (CPU) can perform these checks whenever memory is accessed

- Segments can be swapped out to disk when they are not needed
  - Large programs are split into more manageable parts
  - Only bring code segments into memory when they need to be executed

Segment Sharing

- Suppose two processes run the same program with different data
  - Maybe two different users run the same program on our Linux farm
  - Or the same user opens Word twice in Windows (to edit two documents)



- Processes can share re-entrant code segments and have separate data segments
  - Segment table for each process can point to the same code segment locations
  - Economises on memory usage

Intel Segment Registers

- The Intel CPU includes six registers that point to segments used by the current process
  - CS – code segment
  - DS – data segment
  - SS – stack segment
  - ES – extra segment (often used for string processing – defined by programmer)
  - FS – additional register
  - GS – additional register

  } no specific use (defined by programmer)

- Segmentation has largely fallen out of use in modern operating systems
  - 64-bit CPUs still have these registers for processes running in 32-bit mode
  - Replaced with the paging memory model (see next lecture)
  - Windows and Linux still use the FS and GS registers for other purposes (mainly related to support for multi-threaded processes)