Memory Management (Physical Memory)

# Graphical user interface, text, application, chat or text message Description automatically generatedSummary

# Goals of Memory Management

* **Allocate** memory resources among **processes and OS**, maximising memory utilisation and system throughput
* Provide **convenient abstraction** for **processes** (applications) and **OS**  programmers.
  + Involves simplifying memory utilisation and addressability.
* Provide **isolation** and **protection**  between orthogonal problems

# Basics

* Program must be brought from persistent storage **into memory**, and placed within a **process context** to be run.
* Main memory and CPU registers (and cache) are the only storage that the CPU can access directly, hence we need to take the program into RAM.
  + Memory unit in CPU only sees:
    - Physical memory address + read request
    - Physical memory address + data and write request
* What is the cost of accessing memory?
  + Registers: 1 CPU clock cycle
  + Main memory takes multiple CPU cycles, can be expensive and cause CPU to **stall**.
  + Between these two we have **cache**, sitting between main memory and CPU registers
    - Reduces CPU cycles to access memory
    - Transparent to (assembly) programmer
* Without memory abstraction (see lecture example) programs could access the same address space as the OS and other programs.

## Diagram Description automatically generatedBinding

* **Binding** is the process of mapping i.e. assigning addresses to variables
* Different types of addresses:
  + Relocatable addresses: expressed as a base address + offset
    - Base address may be a symbol
    - offset is a number
  + Absolute addresses: numbers
* Right picture shows how a source program is compiled:
  + Compiler: from source to object file
    - Binds symbolic addresses to relocatable address
  + Linker: from object file to executable file
    - Binds relocatable to absolute addresses
      * If final memory location is known
      * Cannot be moved without hardware support
  + Loader: from executable file image to program in memory
    - * Binds relocatable to absolute addresses
      * After final memory location is decided
      * Cannot be moved without hardware support
  + Execution
* Note that loader and linker can do the same thing, depending on the compiler.
* The above logic applies to C, **not python or java**.

# Addressing

* Table

  Description automatically generatedThe **relocation problem** is where programs using absolute addresses end up clashing as a result of using the same fixed addresses on physical memory.
* Programs must be **relocatable** (address within can be changed):
  + written to be placed and run **at any memory address**
  + Extra information in executable
* Loader decides where to place them based on physical memory available. It handles relocation.
* It solves relocatable addresses by adding an **offset**, obtained via its base address.
  + Give two programs running:
    - The initial first instruction will run at a base address. This becomes the offset for the other addresses used within the program
* This relocation can be expensive as for each process, every variable must be readdressed with the offset
* When there are multiple programs on physical memory, it is all exposed to all processes:
  + User processes may interfere with OS and each other
  + Processes may access OS’s and each other secrets
  + Programs must be relocated when loaded
* This results in a **lack of protection**  and **expensive relocation**

# Protection: Base and Limit Registers

* Diagram

  Description automatically generatedBase and Limit registers are used to define the **valid** address space for a process.
* CPU checks every memory access generated by process
  + Diagram

    Description automatically generatedIf the address is **not valid**, the process is trapped to the OS

# Memory Abstraction

* So far we have looked at the use of purely physical addresses: not useful to developers
* By **abstracting** memory way can make it far easier for the user to access and control
* We define an **address space**:
  + Abstraction from the physical space
  + Set of memory addresses available to a process, independent from other processes
  + Just like a process abstracts physical CPU, address space abstracts physical memory
  + OS provides both of these abstractions

## Logical Addresses

* Every program has a set of logical addresses are independent of physical addresses:
  + Data lives in physical addresses
  + OS manages physical memory and allocates it to different applications
* Instructions from CPU are logical addresses (pointers, arguments to load/store instructions, PC etc.)
* logical addresses are then **translated by hardware** into physical addresses.
* Diagram

  Description automatically generatedThe OS configures the translation.
* The set of logical addresses a program can access is its **address space**.
* Program issues addresses in logical address space
  + Must be translated into physical address
  + Problem has
    - Contiguous logical address starting at 0
    - Contiguous physical address starting somewhere

Timeline

Description automatically generated

## Memory Management Unit (MMU)

* The “**black box**” shown above is the **Memory Management Unit**. It translated CPU generated addresses into physical addresses
* Logical addresses are **bound to**  physical addresses
* Many implementations:
  + Relocation, limit registers, paging etc.

### MMU as Relocation Register

* Diagram

  Description automatically generatedAs shown, the MMU can be as simple as creating an offset using a relocation register.
* Not secure, all addresses are positive values that can be accessed depending on relocation register values.

### MMU as Relocation and Limit Registers

Diagram

Description automatically generated

* In this scenario, we don’t use a loader but instead **hardware**, toprovide support, easing relocation.
  + Relocation register
  + Limit Register
* The whole point is the **program is not relocatable**, resulting a simple loader and faster load times.
* As a result, the **hardware** is checking and not the software
* Same as shown previously.
* This also provides protection as each process has its own private address space.
* Note that processes/programs may have varying sizes, so they are signed **variable size partitions**
  + Relocation register and limit register arithmetic
  + Arithmetic performed for each memory access

Diagram

Description automatically generated

# Memory Allocation

## Diagram, timeline Description automatically generatedContiguous Allocation

* Main memory split into **two partitions**:
  + OS (usually) held in low memory with interrupt vector
  + Processes held in high memory.
    - Each section is contained in a single contiguous section of memory
* Can do this by using relocation and limit registers. This **protects user processes** from each other and from changing OS code and data
  + Relocation registers contain value of **smallest physical address**
  + Limit register contains **range of logical addresses**. Each logical address must be less that limit register.
* MMU translates logical address transparently during execution

### Multiple Partition Allocation

Diagram

Description automatically generated

* Here we use contiguous allocation to manage multiple partitions
* Variable partition size for **efficiency**
* Degree of multiprogramming **limited** by number of partitions
* The gap formed above from process 8 finishing is a **memory hole**
* They can come up anywhere and vary in size
  + When a process arrives, OS allocates memory from a hole large enough to accommodate it
  + Process exiting returns partition to OS, adjected free partitions are combined to form larger hole.
* OS system therefore maintains list of:
  + allocated partitions
  + Free partitions (holes)

## Dynamic Storage-Allocation Problem

* How do we satisfy a request of ***n*** from a list of free holes
* **First-fit**: allocate **first hole** big enough
* **Best-fit**: Allocate **smallest** hole big enough: must search through entire list unless ordered by size (this produces smallest leftover hole)
* **Worst-fit**: Allocate **largest** hole, also searches through entire list (produces largest leftover hole)
* First fit and best-fit better than worst-fit in terms of speed and storage utilisation

## Swapping

* **Diagram

  Description automatically generated**What happens when we don’t have space left in memory?
* We can use swapping: take a snapshot of an idle process’s address space, remove it from memory and put it into disk.
* Three operations:
  + **Swap-in**: Bring in memory of process in its entirety from disk
  + **Run**: run process
  + **Swap-out**: moving memory into disk
* Works with **processes: not programs**
* Idle processes are stored on disk, meaning that no memory taken up while they are not running

## Growing applications

* So far we have assumed programs have a fixed address space
* In general we cannot always guarantee how big an application throughout its executional lifetime.
* We could:
  + Allocate space for growing program **at end**: more space needed then relocate process
  + Allocate space for growing program in its address space, increase heap and decrease stack size.

Diagram

Description automatically generated

* Neither of these solutions are ideal

## Memory Fragmentation

* A picture containing shape

  Description automatically generatedProcess creation and exit, swapping process growing and shrinking: all problems we need to account for
* Say we have a new process, and there are lots of memory holes but none of them are big enough for the new process
* This is the **fragmentation** problem: we have the space but our program won’t fit
* We can use **compaction**, where we combine all of the holes into a bigger hole via:
  + Memory copy
  + Swapping out and in
* This however can be computationally expensive
* There are two types of fragmentation:
  + **External Fragmentation**
    - Allocate **exact** amount of memory requested
    - Total memory space exists to satisfy request, but not contiguous
  + **Internal Fragmentation**
    - Allocate **more** than what is required
    - Allocated memory may be slightly larger than requested memory
* Analysis shows that memory is always lost to fragmentation