Virtual Memory

# Large program

* How do we deal with programs that are larger than the physical available memory?

# Overlays

* Diagram

  Description automatically generatedNot really used today: early solution
* Only part of program loaded at any time
* Programmer breaks **address space into spaces** that fit into memory.
  + This is constrained by **physical memory size**
* Pieces, called **overlays** are loaded and unloaded by the program
* The **overly manager** is a **part of the program**, and it responsible for:
  + loading overlays that are not in RAM
  + Eventually unload overlays previously in RAM
* Overlays mechanism:
  + Always a **root segment** in RAM, which includes the overlay manager
  + 2+ **memory partitions**, and various segments available
  + Only 1 overlay segment can be in a partition at any given time.

# Virtual Memory

* Fully **decouples** address space from physical memory
* Allows for **larger logical address space** than physical memory

# Paged Virtual Memory

* Based on hardware with OS support.
* Transparent to (application) programmer, no programmer involvement
* All pages of the address space **do not need** to be in memory
  + The full address space exists **on disk**
  + Main memory used as **cache**
* Pages that are needed are transferred into free page frames in memory
  + Chart

    Description automatically generatedIf no free page frames are available, one needs to be **evicted** to make space.
* Virtual memory is larger than physical memory
* Pages in RAM can be requested directly; pages not in RAM must be **fetched**
* Table

  Description automatically generatedThe **valid bit** is used to indicate if a page is in virtual memory

## Page Faults

1. Software accesses a page not in memory (before access, relative page table entry is **invalid**)
2. Hardware triggers a **page fault** (Exception)
3. OS checks internal data structures
   1. Invalid reference: abort original software
   2. Not in memory but reference valid, continue
4. OS finds free frame
   1. Swaps page into frame via **scheduled** disk operation
5. OS set internal data structure to indicate that page is now in memory
6. OS restarts instruction that asserted page fault.

## Diagram Description automatically generatedDemand Paging

* Pages brought into memory **when accessed first** i.e. via program demand
  + Program may start with no pages in memory
* Only code/data needed by a process needs to be loaded
  + What’s needed changes over time
* Some systems try to **anticipate**  future needs
* Pages can also be clustered:
  + OS keeps track of pages that should come and go together
  + Bring in **all** once referenced
* Demand paging can be expensive:
  + heavily depends on storage latency (data transfer rate)

# Page Allocation and Replacement

* When you read in a page, where des it go?
  + Use free frames if available: **Page allocation**
  + No free page frames so **eviction** necessary: **Page replacement**, includes:
    - Mechanisms
    - Replacement
* OS tries to keep a pool of free pages around to avoid cost of eviction
* To provide a High degree of multiprogramming, memory must be used as efficiently as possible
  + Evict fairly among programs in memory

### Page Replacement Mechanism

Diagram

Description automatically generated

* Lets say we need to bring in page from disk buyt physical memory is full
* We choose a **victim** page to swap out
* The victim page is swapped out and its page table reference is changed to **invalid**
* The desired page is swapped in and make its entry valid in the page table.

# Replacement Algorithms

* What page do we evict?
  + Reduce page-fault rate by selecting **best victim page** to reduce page-fault overhead (page faults are expensive)
  + **Best victim page** is the one that will **never be touched again**: don’t need in near future
  + **Belady’s Theorem**: Evicting the page that won’t be used for the longest period of time minimises page fault rate
  + Can evaluate using **data access patterns** of application
  + Evict **unmodified** pages
    - No need to write them back to disk
* Examine **page replacement algorithms**:
  + Assume that processes page s against itself (when we choose a page to evict, we choose pages from same process).
  + Assumption that we have **n fixed page frames**.

## String Memory References

* Graphical user interface

  Description automatically generated with low confidenceOrdered list of pages that program will reference
  + 1,2,3,4,1,2,5

## FIFO (First-In First-Out)

* Replace page that has been **inserted first** and is **still in**
* Example: 5 VPs and 3 PPs (Physical Pages)
* Table

  Description automatically generatedReference String: 0, 1, 2, 3, 0, 1, 4, 0, 1, 2, 3, 4
* Easy to implement: maintain a linked list of all pages **in the order** that they come into memory

### Belady’s Anomaly

* When using FIFO an interesting anomaly occurs
* We would expect that if there are more page frames available in memory, there would be less page faults.
* If we use FIFO however, more page frames do not guarantee less page faults