Virtual Memory

Chapters: 10 (10.3 discussed previously, 10.2.3, 10.4.8, 10.5.4, 10.7, 10.9, 10.10 are optional)

Compulsory: 10.1, 10.2, 10.3, 10.4, 10.5, 10.6, 10,8

# Graphical user interface, text, application Description automatically generatedSummary

# Large program

* How do we deal with programs that are larger than the physical available memory?
* Many cases where entire program is not needed:
  + Code that handles unusual error conditions: rarely ever runs at all.
  + Arrays, lists and tables often allocated more memory than needed. May be declared 100 by 100 elements even though it is seldom larger that 10 by 10 elements.
  + Certain options and features used rarely.
* Benefits of having a program partially in memory include:
  + Programs no longer constrained by amount of physically available memory. Programs can have extremely large virtual address spaces
  + More programs could be run at the same time, with a corresponding increase is CPU utilisation and throughput, but with no increase in response or turnaround time.
  + Less I/O needed to load or swap portions of programs into memory, so each program runs faster.
* **Virtual memory**: separates logical memory (as perceived by developers) from physical memory.
* **Virtual Address Space**: refers to logical/virtual view of how a process is stored in memory.
  + Usually appears to start at 0 and is contiguous
  + MMUs job to map logical pages to physical page frames in memory.
* Gap between heap and stack in virtual address space of process will only be allocated physical pages if the heap or stack grows.
* Virtual addressing also allows processes to share files and memory through page sharing:
  + System libraries e.g. standard C can be shared by several processes by mapping shared object into virtual address space of each process.
  + Processes can **share memory**.
  + Pages can be shared during process creation with fork() syscall, speeding up process creation.

# 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

* Chart

  Description automatically generatedBased 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
  + If no free page frames are available, one needs to be **evicted** to make space.

Table

Description automatically generated

* Virtual memory is larger than physical memory
* Pages in RAM can be requested directly; pages not in RAM must be **fetched**
* The **valid bit** is used to indicate if a page is in virtual memory
  + Valid: page is **legal and in memory**
  + Invalid page is either **not valid or valid**, but currently in **secondary storage**.

## Page Faults

1. Diagram

   Description automatically generatedSoftware accesses a page not in memory (before access, relative page table entry is **invalid**)
2. Paging hardware triggers a **page fault** (Exception) and traps to OS.
3. OS checks internal data structure (table in PCB):
   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 modifies internal data structure to indicate that page is now in memory
6. OS restarts instruction that asserted page fault. Process can now access page.

# Demand Paging

* Pages brought into memory **when accessed first** i.e. via program demand
  + In extreme case, 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)
* Theoretically some programs may attempt to access several new pages of memory with each instruction execution:
  + One page for instruction and many for data
* This cause multiple page faults per instruction.
* Fortunately this is very unlikely as programs have **locality of reference**.

## Hardware Support

* Hardware and support the same for paging and swapping:
  + Page table: ability to mark an entry invalid through valid-invalid bit or special value of protection bits
  + Secondary memory: holds pages that are not present in main memory. Usually high-speed disk or NVM device. Also called **swap device**, and section of storage used for this purpose is called **swap space**.

## Free Frame List

* To resolve page faults, most OSs maintain a **free-frame list**: pool of free frames allowing pages to be brought in.
* Used when new pages are demanded or when stack/heap expand beyond their current allocated spaces.
* Normally allocate frames using **zero-fill-on-demand**: these frames are “zeroed out” before allocation to current process.
* When system starts, all frames are in free-frame list. Over time this will shrink, and either list falls to zero or below a certain threshold. A this point, we must repopulate it.

## Performance of Demand Paging

* To evaluate its performance, we can use **effective access time**.
* Let be probability of a page fault, and PFT be page fault time.
* Ideally is close to 0. The EAT is:
* How do we find out how much time is needed to service a page fault? Page faults can cause following sequence to occur

1. Trap to OS
2. Save registers and process state
3. Determine interrupt (trap) was a page fault
4. Check page reference was **legal**, and determine location of page in secondary storage
5. Issue read from storage to a free page:
   1. Wait in queue until read request is serviced.
   2. Wait for device to seek and/or latency time
   3. Begin transfer of page to free frame
6. Allocate CPU to another process while waiting
7. Receive interrupt from storage I/O subsystem that I/O is complete
8. Save registers and process state for other process
9. Determine interrupt was from secondary storage.
10. Correct page table and other tables to show that desired page is now in memory.
11. Wait for CPU to be reallocated,
12. Restore registers, process state and new page table, and then resume interrupted instruction.

* 3 major task components of page fault service time:

1. service page fault interrupt
2. Read in page
3. Restart Process

* Tasks 1 and 3 can be reduced to several 100 instructions with careful coding, taking up tot 1 to 100 microseconds each.
* For task 2: we can calculate this by looking at how fast data is read from secondary storage:
  + if SS is HDD, then read time is (rotational) latency + seek time + data transfer time
* Another aspect of demand paging is handling and usage of **swap space**.
* I/O to swap space is faster than I/O to file system.
* Windows and Linux will write **replaced pages** to swap space, meaning only needed pages are read from the file system, but all subsequent paging is done from swap space.

# Copy-on-Write

* Technique for forking.
* Instead of copying entire address space to forked child process, they share the same pages of memory.
* These pages are marked as **copy-on-write** pages.
  + If a process or its children write to a copy-on-write page, it will be copied for use by that particular process, hence copying pages only when needed.
* vfork(): parent is suspended and child uses parents address space. Process must be careful not to modify parent’s address space.

# Page Allocation and Replacement

* When you read in a page, where does it go?
* Use free frames if available: **Page allocation**
* No free page frames. Two options:
  + Terminate program: demand paging should improve CPU utilisation and throughput, and be logically transparent to user. Hence process termination not ideal
  + **Use eviction** necessary: **Page replacement**, includes:
    - Mechanisms
    - Replacement
* **Over-allocation**: when the full size of all the running process exceeds the number of free pages.
  + Leads to higher CPU utilisation and throughput, but can result in a high demand for unavailable memory.
* 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 but 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.
* Modify page-fault service from before:

1. Find location of desired page on secondary storage.
2. Find free frame:
   1. if available use free frame
   2. No Free frame available:
      1. Use Page-replacement algorithm to select victim page.
      2. Write victim page to secondary storage (if necessary)
      3. Change page and frame tables accordingly
3. Read desired page into newly freed frame and edit page and fame tables
4. Continue process from when page fault occurred.

* This mechanism would need two page transfers if there are no free pages: one for page out and another for page fault.
* This overhead is reduced by using a **modify bit** or **dirty bit**. Set by hardware if read-in page is modified. If not modified, no need to write back to secondary storage.

# 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 confidenceReference String:** Ordered list of pages that program will reference
  + 1,2,3,4,1,2,5
* Used to evaluate algorithms.

## 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.
* Chart, line chart

  Description automatically generatedIf we use FIFO however, more page frames do not guarantee less page faults
* The reason for this is that FIFO is not a **stack based algorithm**: wherethe set of pagesin memory for frames is **always a subset** of the set of pages that would be in memory for pages.
* Essentially different page frame sizes **changes the order** in which items are removed, which in some cases increases the fault rate.
* The fault rate increase happens because recently requested pages remain at the bottom of the FIFO queue for longer.
* Note this applies to **some** access patterns, not all

## Optimal Algorithm

* Idea: replace page that will **not be used** for the **longest time**.
  + Lowest page-fault rate
  + Never suffers from Belady’s Anomaly as it is **stack based**.
* Example:
  + 3 physical pages, 8 virtual pages
  + Graphical user interface, application

    Description automatically generatedReference String: 7,0,1,2,0,3,0,4,2,3,0,3,0,3, 2,1,2,0,1,7,0,1
* The issue with this is that it is not always possible to read the future.
* Could be learned with learning algorithms though.

## Least Recently Used (LRU) Algorithm

* LRU keeps track of what pages have been used by the application
* This is not completely trivial because we may need hardware support to track what pages have been accessed, or record a history of how frequently or how long a page has been accessed for.
* Looks at past and replaces page that has **not be used in the most amount of time**. Like optimal algorithm looking back in time.
  + Never suffers from Belady’s algorithm as it is also stack based
* Example
  + 3 physical pages, 8 virtual pages
  + Graphical user interface, application

    Description automatically generatedReference string: 7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1
* Generally sound algorithm and frequently used.
* Main problem is how to implement it. Two possible options:
  + **Counters**:
    - Associate each page-table entry a time-of-use field and add CPU a logical clock or counter.
    - Clock is incremented for every memory reference
    - Whenever reference is made to a page, contents of clock register are copied to time-of-use field in page table entry for said page.
    - Requires search through page table to find LRU page and write to memory.
    - Times must also be maintained when page table changed. Overflow of clock must be considered
  + **Linked List**: add page numbers to list like a stack, so that bottom is always LRU
    - Worst case scenario: move time from list back to top. Takes length of list time to change pointer for each element.
* In both scenarios **hardware assistance is required**

## Approximating the LRU: Usage

* Not many computers systems provide sufficient hardware for true LRU, so we **approximate** it/
* Two methods for tracking time page frame has been used.
  + Use **page table entry bits** maintained by **hardware**. Bits can be:
    - Page referenced (if accessed or not)
    - Page modified (if access was in write)
  + Keep **history/counter** for each page in **software**
* To approximate the LRU, there are several **history-based** page replacement algorithms that can be used
  + These record the reference bits at regular intervals, such that a history of bits is kept in a table in memory.
  + Algorithms that do this include:
    - Aging
    - Second-chance (clock)
    - Enhanced second- chance
* There are also **counting-based** page replacement algorithms that keep a counter of the **number of references** made to each page.
  + Algorithms that use this are:
    - LRU
    - MFU

### Additional Reference-Bits Algorithm

* Record reference bits at **regular intervals**.
* Use byte for each page in a table in memory.
* At regular intervals, timer interrupt transfers control to OS.
* OS shifts reference bit for each page to higher-order bit (left bit) of its byte, shifting other bits right by 1 and discarding the low-order bit (right bit).
* Byte now represents history of page usage. Viewing bytes as unsigned integers means the page with the smallest number if the LRU.
* If there are multiple LRUs, use FIFO or swap all LRU pages out.
* Number of history bits can vary depending on hardware available. In the event that there is only the reference bit, we are using the **second-chance page-replacement algorithm**.

### Second Chance

* FIFO variant: adds concept of usage such as memory references
* Examine the pages in FIFO order starting from the beginning of the list
  + Each page has a reference bit:
    - 1: referenced
    - 0: not referenced
* The first frames are loaded into memory.
* At each page fault, for the page at the front of the queue:

1. If for the next page to be removed:
   1. the page has not been re-referenced by the program. Remove the page.
   2. Add the new page at the end of the FIFO with .
2. If :
   1. The page has been re-referenced since it was moved to the back of the queue. We will give it a second chance. Set and place at end of queue.
   2. Re-run on next item in the queue.

* Note that this method would involve a lot of pushing/popping in the queue.
* To mitigate for this we can use **pointers** instead in a round-robin fashion. This is called the S**econd Chance Clock**.
  + When a page is needed, pointer indicating LRU shifts forward until page with reference bit 0 is found.
  + Page is replaced and new page is inserted in this position.
* If all bits are set, will degenerate to FIFO
* This can be further **enhanced** to use a reference and **modify** **bit**.
* Creates finer-grained hierarchy for selecting LRU.

Chart, radar chart

Description automatically generated

## Counting-Based Page Replacement

* Follows two schemes:
  + LRU: pages with larger reference count should be kept. Can lead to issues if a page is temporarily used heavily, but solve is to shift counts right by one bit at regular intervals
  + MFU: argument here is that page with smallest count was probably just brought in and has yet to be used.
* Not very common.

## Page-Buffering Algorithms

* Use of free-frame pools.
  + New pages are read into free frames for immediate processes.
  + Victim page is later written out and its frame is added to the free-frame pool.
* Expanded idea: maintain list of modified pages and save regularly.
  + Whenever paging device is idle, a modified page is selected and written to secondary storage.
  + Modify bit.
  + Increases probability that victim page will be clean.
* Keep pool of free frames but remember which page was in each frame:
  + Frame contents are not modified when they are written to secondary storage.
  + If these frames are written to free-frame pool, we can reuse them directly if it needs to be reused, so no I/O needed
  + When page fault occurs, free-frame list is checked first.
  + Can reduce penalty if wrong victim page is selected.
  + Some versions of UNIX uses this in conjunction with second change algorithm.

# Allocation and Sharing Frames among Processes

* So far we have discussed one process and eviction of pages from this one process.
* This is not always possible as we may have multiple processes with different priorities.
* Must select **minimum frame number**:
  + The lower the number of frames allocated to a process, the higher the page fault rate.
  + Must be enough frames to hold all different pages that any single instruction can reference:
    - One-level indirect instruction can refer to address which refers to another address, so at least 3 frames are necessary.
    - Depends on system architecture.
* Frame allocation amongst these processes can either be:
  + **Equal**: an equal share to each
  + **Proportional**: share based on program size:
    - Let size of virtual process be :
    - If the total number of available frames is , we allocate frames to process , where is approximately
    - must also be larger than minimum page number, but lower than no of free pages.
  + **Priority**: share based on priority of process
  + etc.
* Equal and Proportional frame allocation vary with multiprogramming. The high the level of multiprogramming, the less frames that will be allocated to each process.
* Proportional can be adjusted to account for priority. For priority , where a larger priority is given a higher number:
* Page-replacement algorithms can then fall into two categories:
  + **Local**: each process is given a limit of pages it can use.
    - Process evicts its own pages
    - Advantage: does not affect other processes.
    - Disadvantage: Poor utilisation of all free page frames and long access time.
  + **Global**: victim page is chosen from all free page frames
    - Irrelevant of owner
    - Advantage: Processes’ page frame allocation can vary dynamically and priority tasks can use frames from lower-priority tasks. system throughput is increased.
    - Disadvantage: Risk of **global thrashing**
* Example of global replacement strategy: use kernel routines called **reapers** to reclaim frames from processes when lower threshold is reached. Stops when a different upper threshold is reached.
* Linux use extreme example of this, where they kill processes to free memory by using an **out-of-memory** (**OOM**) **killer**.
  + Each process has an OOM score, calculated with percentage of memory process is using.

### Working Set Model

* **Working set** of a process is used to model the dynamic locality of its memory usage
  + **Working set**: set of pages process currently “needs”
* Definition: where:
  + : time
  + : working set **window** (measured in page references)
  + A page is in WS only if it was referenced in the last references
* **Diagram

  Description automatically generated**Diagram, text

  Description automatically generatedWorking set varies over lifetime of program, as does working set size
* (working set size) changes with **program locality**.
* During periods of poor locality:
  + More pages are referenced
  + Working set size is larger
* The working set must all be in memory, else there will be
  + Heavy page faulting
  + **Thrashing**: when the process spends increasing time fetching pages in and out than doing its task.

### Hypothesising Working Frame

* Estimating for a process
  + Allow process to start only if OS can provide number of frames
* We can use a **local replacement algorithm** to ensure that the working set are occupying the process’s frames.
* We also track each process’s working set size and reallocate page frames among processes dynamically.
* To keep track of a process’s WSs, we can use the **reference bit** with a fixed-interval timer interrupt.

## Working Sets and Page Fault Rates

* As the working set changes over time, the page-fault rate peaks and then troughs.
* Shape, histogram

  Description automatically generatedThe page fault rate can then be used to **steer allocations**

### Page-Fault Frequency Allocation

* Goal: to establish “acceptable” **page fault frequency (PFF)** rate
* We can use a local replacement algorithm:
  + If actual rate is too low, allocate less frames
  + Line chart

    Description automatically generated with low confidenceIf actual rate is too high, allocate more frames

### Thrashing

* When the system spends most of its time **servicing page faults**, and little time doing useful work.
* Two reasons:
  + Poor replacement algorithm that is incompatible with program behaviour.
  + Memory is **over-committed:**
    - OS observes CPU is poorly utilised and adds more processes.
    - Many active processes all losing and re-requesting memory.
    - Paging device must be used and as processes wait for paging device, CPU utilisation is decreased. Creates endless cycle.

# Kernel Memory Allocation

* Kernel memory is often allocated from a free-memory pool separate from that used for the user.
* Two reasons for this:
  + Kernel requests memory for varying sizes of data structures, some of which are less than a page in size. Kernel must minimise waste via fragmentation.
  + Certain hardware devices interact directly with physical memory, and need memory residing in physically contiguous pages as they lack virtual memory interface.

## Buddy System

* Diagram

  Description automatically generatedPower of 2 allocator of physically contiguous pages.
  + Requests are in unit sized powers of 2, rounded up.
* Starts with largest size page using largest order .
* If request size such that :
  + Break down into 2 buddies of size
  + Repeat
* To then free pages, we can merge a page with its buddy.

## Slab System

* A **slab** is made up of one or more **physically contiguous pages**.
* There is a **cache** for each **unique kernel data structure**, and contains instantiations of the data structure:
  + this means that the page allocation is fixed size within each cache.
* If there are free slabs, the allocation to the instantiated object is immediate.
* If there are no free slabs, we search memory to add a slab to the cache.
* SLOB and SLUB variations available in Linux
* Advantages:
  + No memory wasted via fragmentation.
  + Memory requests can be satisfied quickly as objects are created in advance, and when released they are released to the cache for reallocation.

# CPU Cache vs. Virtual Memory “as a cache

* CPU cache is a hardware component
  + It is completely transparent to the programmer
  + It holds incoming data **from memory**
* Virtual memory can also act as a cache: this needs a lot of code from OS software.
  + It is transparent to the application, but not to the OS
  + Virtual memory as a cache holds data coming **from storage**.
  + The OS then moves the cached data from storage to memory.
  + Diagram

    Description automatically generatedHence the OS is involved in this movement.