7 Memory Management

- **Logical address**: how the process views its memory space
- **Base register**: start offset
- **Limit register**: range of memory space of current process

7.1 Contiguous Memory Management

- 2 schemes of allocating memory partition:
  - **Fixed-size**
    Pros: Easy to manage, fast to allocate (all free partitions are the same, no need to choose), **Cons**: partition size need to be large enough to contain the largest process (internal fragmentation)
  - **Variable-size**
    Pros: Flexible, removes internal fragmentation, **Cons:** Need to maintain more information in OS, takes more time to locate appropriate region, external fragmentation
- **Merging and Compaction**: consolidate holes (time-consuming)
- **Allocation algo**: **First-Fit**: first large enough hole, **Best-Fit**: smallest large enough hole, **Worst-Fit**: largest hole

7.1.1 Allocation Algo: Buddy System

- Provides efficient: (1) Partition splitting, (2) locating good match of free partition, (3) Partition de-allocation and coalescing
- Free block is split into half repeatedly to meet request, two halves form buddy blocks. When buddy blocks are both free, merge.
- Keep array `A[0..K]` where 2^K = largest allocatable block size.
- `A[J]` is a linked list, keeps tracks of free block(s) of size 2^J
- Blocks B and C are buddy of size S, if the Sth bit of B and C are complements (0 and 1) and the leading bits up to the Sth bit are the same (e.g. 10100 and 10000)
- Allocating a block of size N:

```
S = Math.log2(N).ceil # smallest S s.t. 2^S <= N
while true
  # If got free block, remove from list and allocate
  allocate(A[S].shift) and return if A[S].size > 0
  # Else, find smallest R s.t. A[R] has a free block
  # repeatedly split s.t. A[S..R-1] has a new free block
  R = (S+1).upto(K).find { |i| A[i].size > 0 }
  (R+1).downto(S).each{ |i| split A[i], from: A[i+1] }
end
```

- To free a block B:

```
def free(B)
  S = Math.log2(B.size).ceil
  C = A[S].find_free_buddy_of(B)
  A[S].append(B) and return if C.nil? # buddy not free
  A[S].delete(C)
  B2 = merge(B, C) # merge buddies
  free(B2)
end
```

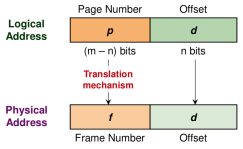
8 Disjoint Memory Schemes

8.1 Paging Scheme

- Physical mem is split into fixed-size regions: **Physical Frame**
- Logical mem of a process is split into regions of the same size: **Logical Page**
- Logical memory space is contiguous but occupied physical memory region can be disjointed.
- Paging removes external but **not internal** fragmentation.

8.1.1 Logical Address Translation

- **Page Table**: lookup table for logical address translation (logical page <-> physical frame). Need to know 2 things: F = physical frame number, Offset = from start of physical frame
- Physical address = $F\times$ physical frame size + Offset
- Given: frame size = 2^n, m bits of logical address



Logical Address LA:
p = most significant $m-n$ bits of LA,
d = remaining n bits of LA

Physical Address
$PA = f \times 2^n + d$

8.1.2 Implementation

- **Pure-software**: OS stores page table information in the PCB
  Issues: require 2 mem access for every mem ref (1) read indexed page table entry to get frame number, (2) access actual mem item
- **Hardware support**: Translation Look-aside Buffer (TLB) = cache of a few page table entries
  - Use page number to search TLB, either TLB-Hit or TLB-Miss
  - TLB is part of hardware context. Thus, context switch -> TLB entries are flushed

1. **(HW)** LA is decomposed into <Page#, Offset>
2. **(HW)** Search TLB for Page#:
   (a) TLB-Hit: Use <Frame#, Offset> to access physical mem. Done.
   (b) TLB-Miss: Trap to OS (TLB Fault)
3. **(OS - TLB Fault)** Access full table of the process (in PCB), <Page#> is the index:
   (a) If memory resident, replace a TLB entry, return from trap, retry step 1
   (b) If non-memory-resident, trap to OS (Page Fault)
4. **(OS - Page Fault)** Locate the page in secondary swap pages. Load the page into a physical frame (applying replacement algo if needed). Update PTE, return from trap, retry step 3

8.1.3 Protection

- **Access Right Bits**: each page table entries has several bits to indicate rwx, mem access is checked against the access right bits
- **Valid bit**: each page table entries has a bit to indicate if the page is valid for the process to access. OOB access caught by OS

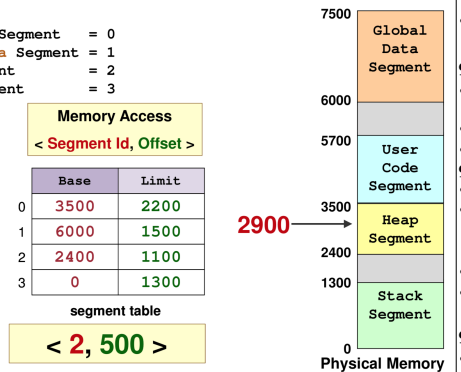
8.1.4 Page sharing

Deduplication. Possible usage:

- **Shared code page**: used by many processes, e.g. C stdlib, syscalls
- **Implement CoW**: upon forking, before any mem value mutation

8.2 Segmentation Scheme

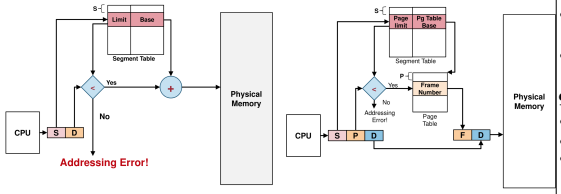
- Some regions may grow/shrink at exec time. Thus putting them in non-contiguous mem space to allow them to grow/shrink freely and easier check whether access is in-range.
- Mem space of a process separated into different segments: user code, global vars, heap, stack, lib code, etc.
- Logical mem space of a process is a collection of segments
- Each segment: has a name and a limit
- All mem ref is `Segment Name + Offset` (e.g. Heap + 245)
- Each segment mapped to a contiguous physical mem region (base address + limit)
- **segment id** = single number representing segment name
- Logical address: <SegID, Offset> (SegID used to look up <Base, Limit> in a segment table)
- PA = Base + Offset, Offset < Limit for valid access
- **Pros**: Each segment is an independent contiguous mem space (can grow/shrink, protected/shared independently)
- **Cons**: Requires variable-size contiguous memory regions, external fragmentation



8.3 Segmentation: Hardware Support and combined with Paging

Each segment is now composed of several pages instead of a contiguous memory region (each segment has a page table)

8.3.1 Table: (Segment#, Page Limit, Page Table Base)



9 Virtual Memory Management

- Use page table to translate **virtual** -> physical address
- Extend paging scheme: Some pages may be in physical memory, others in secondary storage.
- Use a `memory_resident?` bit to distinguish between 2 page-types: **memory-resident** and **non-memory-resident**
- CPU can only access memory-resident pages. **Page Fault** (PF) when CPU tries to access non-memory-resident page. OS needs to bring them into physical memory.
- Just like cache, exploit **Temporal** & **Spatial** Locality
- Accessing page X:
  1. If page X **memory resident**: Access physical mem. Done.
  2. Else **PF**: Trap to OS
  3. Locate page X in secondary storage
  4. Load page X into a physical memory frame
  5. Update page table
  6. Go to step 1

9.1 Page Table Structure

9.1.1 Direct Paging

- Keep all entries in a single table
- With 2^p pages in logical mem space, p bits to specify one unique page, 2^p PTE (each containing physical frame number + additional info bits), e.g. Virtual Address: 32 bits, page size = 4 KB, $p = 32 - 12 = 20$, size of PTE = 2 B, PT Size = $2^{20} \times 2$ bytes = 2MB

9.1.2 2-Level Paging

- Split page table into smaller page tables, each with a **PT number**
- Total 2^P entries, with 2^M smaller PTs, M bits to identify 1 PT, each smaller PT contains $2^{(P-M)}$ entries
- Use **page directory** to keep track, containing 2^M indices. Ideally, size of page dir = page size
- **Advantages**: corresponding PT to empty entries in page dir need not be allocated

9.1.3 Inverted Page Table

- Instead of keeping per-process PT, only a **single** mapping of physical frame to <pid, page#> (ordered by frame# instead of page#)
- **Advantage**: Memory saving, 1 table for all processes
- **Disadvantage**: Slow translation ($O(n)$ to look up a page#)

9.2 Page Replacement Algorithms

- When a page is evicted: **clean page** (not modified), **dirty page** (modified, must write back)
- Memory access time: $T_{access} = (1-p) \times T_{mem} + p \times T_{page\_fault}$ where p = prob of PF, T_{mem} = access time for mem-resident page, T_{page_fault} = access time if PF occurs
- Since $T_{page\_fault} >> T_{mem}$, need to reduce p to keep T_{access} reasonable (reduce total no of PF)

9.2.1 Optimal Page Replacement (OPT)

- Replace the page that will not be used again for the longest period of time. Guarantees min no of PF
- However, not realisable, need **future knowledge** of mem refs
- Base comparison for other algorithms

9.2.2 FIFO

- Mem pages are evicted based on their loading time
- Implementation: OS maintain a queue of resident page numbers. Remove the first page in queue if replacement is needed. Update the queue during PF trap.
- **Advantage**: simple to implement (no HW support needed)
- **Problems**: Belady's Anomaly (more frames, more PF) because FIFO does not exploit temporal locality

9.2.3 Least Recently Used (LRU)

- Make use of temporal locality. Does not suffer from Belady's Anomaly
- **Impl 1**: A logical time counter, incremented for every mem ref. PTE has a time of use field, replace page with smallest time of use.
- However, need to search through all pages, and counter is forever increasing (overflow)
- **Impl 2**: Use a stack. If page X is referenced, remove from stack, push on top of stack. Replace the page at bottom of stack.
- However, not a pure stack. Entries can be removed from anywhere in the stack. Hard to implement in hardware.

9.2.4 Second-Chance Page Replacement (CLOCK)

- Modified FIFO to give a second chance to pages that are accessed.
- Each PTE maintains a reference bit: **1** = accessed, **0** = not accessed
- Algo:
  1. Select oldest FIFO page.
  2. If ref_bit == 0, page is replaced.
  3. If ref_bit == 1, page is given 2nd chance: ref_bit cleared to 0. Arrival time reset (as if newly loaded). Next FIFO page is selected, go to step 2
- Impl: use a **circular queue** to maintain the pages, with a pointer to the oldest page. To find a page to be replaced, advance until a page with ref_bit == 0, and clear reference bit as pointer passes.
- When all ref_bit == 1, degenerate into FIFO

9.3 Frame Allocation

- N physical mem frames, M processes competing for frames.
- **Equal allocation** (each process gets $\frac{N}{M}$ frames), **Proportional Allocation** (Let $size_p$ = size of process p, $size_{total}$ = total size of all processes, each process gets $\frac{size_p}{size_{total}} \times N$ frames)
- Insufficient physical frame -> Thrashing (heavy IO to bring non-resident pages into RAM)

9.4 Page Replacement

- **Local Replacement**: victim page selected among pages of the process that causes PF
  Pros: Frames allocated to a process remain constant, perf is stable between runs. **Cons**: if frame allocated not enough, hinder progress of a process
  A thrashing process steals page from other process causing other process to thrash (**cascading thrashing**)
- **Global Replacement**: victim page chosen among all physical frames
  Pros: Allow self-adjustment between processes (process that needs more can get from other). **Cons**: badly-behaved process can affect others, frames allocated to a process can be different from run to run (thrashing can be limited to one process, but that process can hog the IO and degrade perf of other processes)

9.5 Working Set Model

- In a new locality, a process will cause PF for the set of pages. Afterwards no/few PF until process transits to a new locality
- Defines **Working Set Windows** Δ = time interval
- $W(t, \Delta)$ = active pages in the interval at time t
- interval = looking back
- Allocate enough frames for pages in $W(t, \Delta)$ to reduce prob of PF
- Accuracy of working set model directly affected by choice of Δ. **Too small**: may miss pages in the current locality, **Too big**: may contain pages from different locality

10 File System

General Criteria: **Self-contained**, **persistent**, **efficient**

10.1 File System Abstraction

10.1.1 File

- logical unit created by process, contains data and metadata (name, identifier, type [regular {ASCII, Binary}, directories, special files], size, access rights, timedate, owner, table of content)
- File **protection**: permission bits (owner, group, universe) (rwx), Access Control List (minimal ACL – same as permission bits/extended ACL – added named users/group)
- **Operations** on metadata: Rename, Change/Read attributes
- **Structure**: **Array of bytes**, **Fixed-length records** (array of records, can grow/shrink/jump to any record easily), **Variable length records** (flexible, but harder to locate a record)
- **Access methods: Sequential** (data read in order from beginning, cannot skip but can rewind), **Random** (data can be read in any order, **Direct access** (for file containing fixed-length records, allow random access to any record directly)
- **Generic operations**: create, open, r/w, repositioning, truncate
- OS provides file ops as syscalls. Information kept for opened file: **File pointer** current loc in file, **disk location**: actual file loc on disk, **open count**: how many process opens this file
- Organise open file information: **System-wide open-file table** (1 entry/unique file), **Per-process open-file table** (1 entry/file used in the process, each entry points to the system-wide table)
- Processes sharing file in unix: (1) diff file descriptors, I/O can occur at indep offsets, (2) same file descriptor, only 1 offset, I/O changes the offset for the other process (e.g. `fork` after file is opened)

10.1.2 Directory

- Used to (1) provide logical grouping of files, keep track of files
- Ways to structure directory:
  - **Single-level**
  - **Tree-structured**
    Dirs can be recursively embedded in other directories. 2 ways of referring to the file: **absolute/relative pathname**
  - **DAG**
    A file can be shared, only 1 copy of actual content appearing in

multiple directories with diff path names. In Unix: hard link (pointers to the same actual file on disk, pros: low overhead, cons: deltion problems)/symbolic link (special link file containing pathanme, pros: simple deletion, cons: larger overhead)

– **General Graph**
Generally not desirable: hard to traverse (need to prevent infinite looping), hard to determine when to remove a file/dir.

11 File System Implementations

11.1 Disk Organisation

- **Master Boot Record** at sector 0 with partition table, followed by 1/more partitions, each containing an independent file system.
- Logical view, file = a collection of logical blocks
- When file size ≠ multiple of logical blocks, last block may contain wasted space (**internal fragmentation**)
- Good file implementation: keep track of the logical blocks, allow efficient access, disk space is utilised effectively. (focus on how to allocate file data on disk)

11.2 File Block Allocation

11.2.1 Contiguous

- Allocate consecutive disk blocks to a file
- **Pros**: simple to keep track (each file only needs startinb block number + length), fast access (only need to seek to first block)
- **Cons**: External frag, file size needs to be specified in advance

11.2.2 Linked List

- Keep a linked list of disk blocks, each disk block stores the next disk block numbers (pointer) and actual file data
- File information stores first and last disk block number
- **Pros**: solve fragmentation
- **Cons**: slow random access in a file, part of disk block is used for pointer, less reliable

11.2.3 Linked List Variant: FAT

- All block pointers in a single table: **File Allocation Table** (FAT)
- **Pros**: Faster Random Access (linked list done in memory)
- **Cons**: FAT can be huge when disk is large

11.2.4 Indexed Allocation

- Each file has an **index block** (an array of disk block addresses where `indexBlock[N]` = Nth block address)
- **Pros**: Lesser memory overhead (only index block of opened file needs to be in memory), fast direct access
- Cons: Limited maximum file size (max no of blocks == no of index block entries), index block overhead

11.2.5 Indexed Allocation Variations

- Schemes to allow larger file size
- **Linked scheme**: keep a linked list of index blocks
- **Multilevel index**: 1st level index block points to a number of 2nd level index blocks. Each 2nd level index blocks point to actual disk block. This can be generalised to any number of levels.
- **Combined scheme**: combination of direct indexing and multi-level index scheme, e.g. Unix I-node has 12 direct pointers, 1 single indirect, 1 double indirect, 1 triple indirect

11.3 Free Space Management

Free space list to know which disk block is free during file alloc

11.3.1 Bitmap

- Each disk block represented by a bit, 1 = free, 0 = occupied
- **Pros**: provide a good set of manipulations using bit-level ops
- **Cons**: need to keep in memory for efficiency reason

11.3.2 Linked List

- A linked list of disk blocks, each disk block contains a number of free disk block numbers; a pointer to next free space disk block
- **Pros**: Easy to locate block, only 1st pointer is needed in memory
- **Cons**: High overhead (mitigated by storing the list in free blocks)

11.4 Directory Structure

- Given a full path name, recursively search the directories along the path to arrive at the file information
- Each dir entry stores file name, metadata, (pointer to) file information, disk blocks information

11.4.1 Linear List

- Directory = a list of files
- Locate a file using list requires a linear search (inefficient for large directories or deep tree traversal). **Common solution**: use cache to remember the latest few searches

11.4.2 Hash Table

- Directory = hash table of size N
- Locate a file by file name: File name is hashed into index $K \in [0, N)$. **HashTable[K]** inspected to match file name, using chained collision resolution
- **Pros**: fast lookup
- **Cons**: hash table has limited size, depends on good hash function

11.5 File System in Action

11.5.1 Create /.../parent/F

1. Use full path name to locate parent dir, search for filename F to avoid duplicates. Search could be on cached dir structure
2. Use free space list to find free disk block(s)
3. Add an entry to parent directory with relevant file information

11.5.2 Process *P* open file /.../F

1. Search system-wide table for existing entry E. Not found, use full pathname to locate file F and load its file information to a new entry E in system-wide table. If not found terminate
2. Create an entry in the P's table, pointing to E
3. Return pointer to this entry.

12 Additional Notes

12.1 Internal vs External Fragmentation

Internal fragmentation is the wasted space within each allocated block because of rounding up from the actual requested allocation to the allocation granularity.

External fragmentation is the various free spaced holes that are generated in either your memory or disk space. External fragmented blocks are available for allocation, but may be too small to be of any use.

12.2 Memory Calculations

- **Physical Address** = [frame number] [offset] | **Virtual Address** = [page number] [offset]
- **Physical -> Virtual OR vice versa**:
- Offset = addr $\mod 2^{\text{n-offset-bits}}$
- $\frac{\text{p\_or\_vAddr} - \text{Offset}}{2^{\text{n\_offset\_bits}}}$ = page OR frame number
- **Page/Frame Number + offset -> P or V address**:
- address = page or frame number $* 2^{\text{offset bits}} + \text{Offset}$