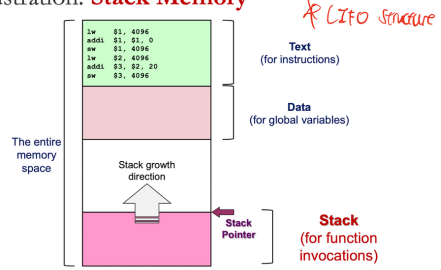
CS2106 Finals cheatsheet

By: Khoo Jing Hong, Derrick

Stack Memory:
Stack either grows towards higher or lower addresses -> Platform-dependent (Hardware dependent)

Illustration: Stack Memory

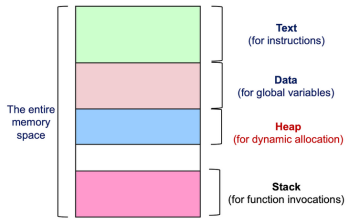


Frame Pointer(FP) is used to traverse and access items in a stack. It is for convenience when there are conditional branches.

GPRs are limited, eg MIPS has 32 while x86 has 16. When GPRs are exhausted, use memory to temporarily hold GPR value, then that GPR can be reused for other purpose and then the GPR value is restored afterwards (**register spilling**).

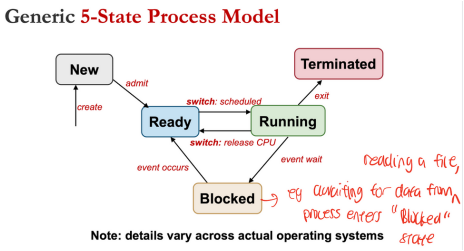
Stack items include:
- return PC
- saved FP (optional, depends if FP avail)
- saved SP
- saved GPR registers from caller
- Parameters
- Local variables

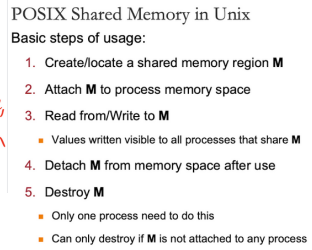
Heap memory(Dynamically allocated memory):
Separate in memory, in heap memory region.



** Process giving up CPU voluntarily -> Running to Ready **ONLY** (eg time quantum used up)

Process State





Given n processes,
- With 1 CPU core
  - <= 1 process in running state
  - conceptually 1 transition at a time
- With m CPU cores
  - <= m process in running state
  - parallel transitions possible

Next process to run is determined by OS scheduling algorithm.
Process Control Block(PCB) maintained by Kernel in OS.
Process Table keeps track of all PCBs.

Interrupts. OS does not interfere with hardware operations when an interrupt occurs.
Status registers are registers that saves current state of CPU and enables/disables interrupt

Process Creation in UNIX
fork(): returns PID child process = 0; parent = pid of child process.
Child process is a **duplicate** of current executable image (same code, same address space etc) but data in child is a **COPY**(local variable)
exec(): Execute another existing program in child process instead. However, **parent-child relationship not destroyed.**

Master process init process. Created in kernel at bootup time and has PID = 1.
Process Termination using exit(value). value == 0 means successful execution while value != 0 means problematic execution.
Parent-Child synchronisation:
Parent process needs to block until all child terminates, if not there will be zombie processes
Zombie processes Child processes that have terminated but resources are still present in the system because parent did not call wait.
Orphan processes Parent process terminated before child process. This causes child process to send signal to init which utilises wait() to cleanup.

Memory Copy Optimisation:
Copy on Write: Only duplicate a "memory location" when it is written to, otherwise parent and child share same memory location.

Inter-Process Communication:
2 IPC Mechanisms: **Shared-Memory** and **Message Passing**
Shared Memory OS is involved only in creation and attaching of shared memory
Adv: Efficient because OS needed only to setup shared regions.
Ease of use because there is just simple reads and writes to arbitrary data types
Disadv: Limited to a single machine, and requires synchronisation (need to synchronise accesses if not data racing occurs -> incorrect behaviour)
Race conditions due to possible interleaving of read and write operations.
** Remember that LOAD always comes before STORE so thus when asked for permutations/ always divide by 2.

Message Passing involves OS in every message to and fro (syscalls)
Message stored in kernel memory space
Direct Communication Sender/Receiver explicitly names other party(eg by PID)
Characteristics:
One link per pair of communicating processes
Need to know identity of other party
Indirect Communication Message are sent to/received from mailbox
Characteristics: OS handles who to recv msg
One mailbox can be shared among processes
Advantages
- Applicable beyond a single machine
- Portable
- Easier synchronisation

Disadvantages:
- Inefficient (requires OS intervention upon every send/receive)
- Harder to use (requires packing / unpacking data into a supported msg format)

Redirection of Output in UNIX fd values: stdin = 0, stdout = 1, stderr = 2 Use dup2(int oldFileDescriptor, int newFileDescriptor) to align oldFd to newFd(output in newFd will show in oldFd)

Process Scheduling

Realise that context switching causes OS to incur overhead in switching betwn processes.

Refer to time when process uses CPU as **CPU burst.**

Refer to time when process uses I/O as **I/O burst.**

Compute-Bound processes: Majority of time spent using CPU

I/O Bound processes: Majority of time spent using I/O

Criteria for all processing environments:

- Fairness (also means no starvation)
- Utilization (all parts of computing system should be utilized)

Scheduling policies:

- Non-preemptive(Cooperative)
  - Process stays in running state until it blocks or gives up CPU **voluntarily**
- Preemptive
  - Process gets forcefully removed once time quantum is reached

Batch Processing Algorithms

- No user interaction (Don't care about response time)
- Non-preemptive scheduling is predominant

Criterias for batch processing:

1. **Turnaround time** (Finish time - arrival time) or total time taken
2. **Waiting time** (Time spent waiting for CPU)
3. **Throughput** (Number of tasks finished per unit time)
4. **Makespan** (Total time from start to finish to process all tasks)
5. **CPU Utilization** (Percentage of time when CPU is working on a task)

Batch Process Scheduling Algorithms (No time quantum discussion)

1) **First-Come First Served(FCFS)**

Tasks are placed in **FIFO ready queue**, based on arrival time

- Task will run until it is done or is blocked. When ready again, rejoin queue from back

Guaranteed to have **no starvation**

However, reordering tasks may **reduce average waiting time. (Shorter jobs at front of queue)**

Problem

Convoy effect eg first task is CPU-bound then remaining tasks are I/O bound. Then CPU is idle when first task blocks on I/O and remaining tasks run on I/O. (Poor utilization of resources)

2) **Shortest Job First (SJF)** -> Minimize average waiting time

Select task with **smallest total CPU time**

Problem

Starvation due to bias towards shorter jobs

Prediction of CPU time needed.

Shortest Job First: Predicting CPU Time

- A task usually goes through several phases of CPU-Activity:
  - Possible to guess the future CPU time requirement by the previous CPU-Bound phases
- Common approach (Exponential Average):

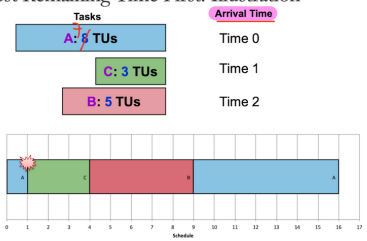
$Predicted_{n+1} = \alpha Actual_n + (1-\alpha) Predicted_n$

  - $Actual_n$ = The most recent CPU time consumed
  - $Predicted_n$ = The past history of CPU Time consumed
  - α = Weight placed on recent event or past history
  - $Predicted_{n+1}$ = Latest prediction

3) **Shortest Remaining Time (SRT)** -> Variation of SJF that is preemptive instead

New job that arrives with shorter remaining time can preempt current running job

Shortest Remaining Time First: Illustration



Interactive Systems Scheduling Algorithms (Time quantum discussion)

Criterias for interactive systems:

- Response time (**minimized through shorter time quantum**)
- Predictability (Variation in response time, lesser variation == more predictable)

Preemptive scheduling is predominant here to ensure good response time

Time Quantum is the execution duration given to a process. **Short time quantum** = Minimise response time but bigger overhead in context switching. **Long time quantum** = Reduce overhead in context switching for OS, but better CPU utilization.

1) **Round Robin (RR)**

Tasks are stored in FIFO queue. Tasks are picked from front of queue and runs until fixed time slice(quantum) elapsed || task gives up CPU voluntarily || task blocks

- Preemptive version of FCFS
- Response time guarantee
  - Given n tasks and quantum q
  - Time before a task gets CPU is bounded by (n-1)q

2) **Priority Scheduling (No time quantum)**

2 variants: **Preemptive** where higher priority process can preempt current running process with lower priority or **Non-preemptive** where higher priority process has to wait for next round of scheduling.

Problem

Starvation of lower priority processes if high priority process keep hogging CPU

Priority Inversion problem

Priority Scheduling: Priority Inversion

- Consider the scenario:
  - Priority: {A = 1, B=3, C= 5} (1 is highest)
  - Task **C** starts and locks a resource (e.g., file)
  - Task **B** preempts **C**
    - **C** is unable to unlock the resource
  - Task **A** arrives and needs the same resource as **C**
    - but the resource is locked!
  - → Task **B** continues execution even if Task **A** has higher priority
- Known as **Priority Inversion**:
  - Lower priority task preempts higher priority task

Interactive Systems Scheduling Algorithms (Time quantum discussion)

3) **Multi-Level Feedback Queue(MLFQ)**

MLFQ **minimises both response time** for I/O bound processes(interactive systems) and **turnaround time** for CPU bound processes (batch scheduling)

MLFQ: Rules

- Basic rules:
  1. If Priority(A) > Priority(B) ➔ A runs
  2. If Priority(A) == Priority(B) ➔ A and B runs in RR
- Priority Setting/Changing rules:
  1. New job ➔ Highest priority
  2. If a job fully utilized its time quantum ➔ priority reduced
  3. If a job gives up / blocks before finishes its time quantum ➔ priority retained

Advantage:

Since I/O jobs are usually short jobs, by having an appropriate time quantum, I/O bound processes are always high priority -> **Minimize response time**

Problem

Able to abuse MLFQ by blocking before time quantum.

4) **Lottery Scheduling**

Lottery Scheduling

- General Idea:
  - Give out "lottery tickets" to processes for various system resources
    - E.g., CPU time, I/O devices, etc.
  - When a scheduling decision is needed:
    - A lottery ticket is chosen randomly among eligible tickets
    - The winner is granted the resource
  - In the long run, a process holding **X**% of tickets
    - Can win **X**% of the lottery held
    - Use the resource **X**% of the time

Lottery Scheduling: Properties

- Responsive:
  - A newly created process can participate in the next lottery
- Provides good level of control:
  - A process can be given Y lottery tickets
    - It can then distribute to its child process
  - An important process can be given more lottery tickets
    - Can control the proportion of usage
  - Each resource can have its own set of tickets
    - Different proportion of usage per resource per task
- Simple Implementation

Synchronisation primitives

Synchronisation is needed to control the interleaving of accesses to a shared resource among processes

Solution Critical Section(CS) for segment of code to write or read a resource.

Overhead Performance due to enforcing of CS.

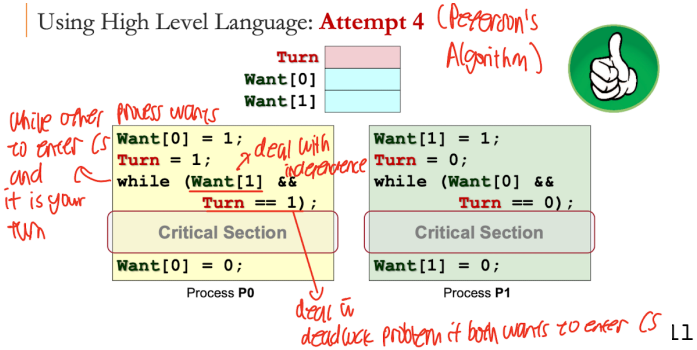
Properties of Correct CS:

- Mutual Exclusion
- Progress (If no process in CS, one of the waiting processes to enter CS should enter)
- Bounded Wait (there exists an upper bound on the number of times other processes can enter CS before Pi, after Pi requests to enter CS)
- Independence (Process not executing in CS should not block other process from entering CS)

Symptoms of wrong implementation of CS:

- **Deadlock** (all processes blocked)
- **Livelock** (when processes keep switching state to avoid deadlock but make no progress)
  - Usually a result of deadlock avoidance mechanisms
- **Starvation**

Peterson's Algorithm(No semaphore, code level)



not work due to **violation of mutual exclusivity.**

Disadvantages of Peterson:

- Busy waiting -> Unwanted usage of CPU
- Too low-level -> Error prone
- Not general -> Hard to solve all synchronisation problems for n processes

Synchronisation primitives

Test-And-Set (Assembly level sync)

TestAndSet Register, MemoryLocation: Assembly level, we load the value at MemoryLocation into Register **AND** change the value at MemoryLocation to 1.

Atomic instruction even on multi-core systems

Problem

Busy Waiting -> Wasteful CPU usage

No guarantee on bounded-wait -> Unless scheduling algorithm is fair

Using Semaphores

Must be initialised to a non-negative value

Invariant: Sem(curr) = Sem(initial) + numSignal - numWait

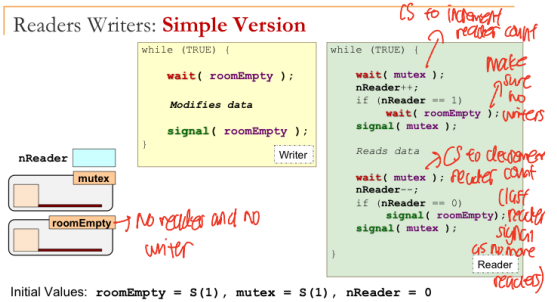
General Semaphores used in Safe-Distancing problem, allowing up to n number of processes to execute

Binary Semaphores used in enforcing Critical Section**(Mutex)**

General Synchronisation where there exists a Producer-Consumer relationship, initialise semaphore to 0 and let consumer call wait.

Classical Synchronisation Problems

Readers-Writers problem



Dining Philosophers problem

Solved using Tanenbaum's solution

- Each philosopher is waiting on its own semaphore
- Once philosopher is done, have to notify neighbours(signal) that they can eat

Threads - Alternative to Process

Threads share same memory context - Text, Data, Heap and OS context - PID, files etc

Threads have unique
- Identification (Thread ID)
- Registers (GPR and Special)
- Stack

Context Switching only involves hardware context, where registers, SP and FP pointers are changed

Benefits
- Economy

Much less resources needed for multiple threads compared to multiple resources
- Resource Sharing

Threads share most of the resources, no need to pass information around
- Responsiveness

Multithreaded processes seem more responsiveness
- Scalability

Problems
- Syscall concurrency

Parallel execution of multiple threads -> interleaving of syscalls -> parallel syscalls -> behaviour unintended without synchronisation
- **Execv**

If any thread calls exec() or its variants, all threads are terminated and replaced with new image of new process.

Implementations
- **User thread**

Thread is implemented as user library, **Kernel** not aware of the threads in the process
  - **Advantages**
    - Can have multithreaded program on **ANY OS**
    - Thread operations are just library calls
    - More configurable and flexible
  - **Disadvantages**
    - OS not aware of threads, scheduling performed at process level e.g. 1 thread blocked -> process blocked -> all threads blocked, and cannot exploit multiple CPUs
- **Kernel thread**

Thread is implemented in OS, thread operations handled as syscalls. **Thread-level scheduling** is possible.

Advantages:
- Can schedule on thread levels -> More than 1 thread of same process can run simultaneously on multiple CPUs

Disadvantages:
- Thread operations are now syscalls -> More resources needed and slower
- Less flexible

Hybrid Model both user and kernel threads
- OS schedule on kernel threads
- User threads bind to kernel threads

Simultaneous Multi-threading hardware support on modern processors: Supply multiple sets of registers(GPRs and special registers) to allow threads to run natively and in parallel on same core.

Memory Abstraction

Usage of **special registers (Base, Limit)** to store starting address of process memory space and the maximum range of the memory space of current process respectively.

Contiguous Memory Allocation

Generally, contiguous memory allocations will lead to **external fragmentation**.

Assumptions for following algorithms:
1. Each process occupies a **contiguous memory region**
2. **Physical memory large enough** to contain one or more processes with **complete memory space**

Fixed Size Partitions

Physical memory split into fixed number of partitions of equal size, a process will occupy one of such partitions.

Advantages
- Easy to manage, fast to allocate

Disadvantages
- Partition size must be large enough to contain largest of processes -> Smaller process larger internal fragmentation

Variable sized Partitions

Partition created based on actual size of process, OS keeps track of occupied and free memory regions to perform splitting and merging where necessary.

Internal fragmentation: There exists unused space within a partition when it is occupied

External fragmentation: When holes between partitions become unusable, wasting space. Can be fixed via compaction(Move partitions around), but slow. **More so when more than required amount of memory is available, but not contiguous hence considered not enough space.**

Advantages:
- Flexible and solves **internal fragmentation problem**

Disadvantages:
- Need to maintain information in OS -> overhead
- Takes more time to locate appropriate region

Algorithms for Variable sized partitions

First-Fit

Take the first hole that is large enough

Best-Fit

Find the smallest hole that is large enough

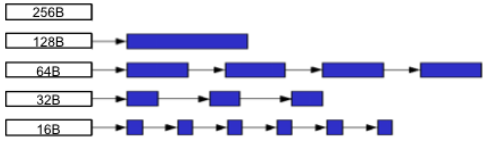
Worst-Fit

Find the largest hole

Reducing external fragmentation using **Compaction, Merging.** When an occupied partition is freed, merge with adjacent hole if possible. Move occupied partitions around to create bigger, consolidated holes(but problem here is that such invocation is very time consuming)

Partition Information

Multiple Free Lists keep multiple lists of different hole sizes. Take hole from a list that most closely matches request size. Partition size increases exponentially and this results in faster allocation.



Buddy System idea is to have a free block split into half repeatedly to meet request size. 2 halves form a buddy and when buddy blocks are both free, merge together to form larger block

Buddy System: Implementation

- Keep an array `A[0…K]`, where 2^K is the largest allocable block size
  - Each array element `A[J]` is a linked list which keep tracks of free block(s) of the size 2^J
  - Each free block is indicated just by the starting address
- In actual implementation, there may be a smallest allocable block size as well
  - A block that is too small is not cost effective to manage
  - We will ignore this in the discussion

Buddy System: Allocation Algorithm

- To allocate a block of size **N**:

1. Find the smallest **S**, such that 2^S >= **N**
2. Access `A[S]` for a free block
   a. If free block exists:
      - Remove the block from free block list
      - Allocate the block
   b. Else
      - Find the smallest `R` from `S+1` to `K`, such that `A[R]` has a free block `B`
      - For (`R-1` to `S`)
        - Repeatedly split `B` → `A[S...R-1]` has a new free block
      - Goto Step 2

Buddy System: Deallocation Algorithm

- To free a block `B`:

1. Check in `A[S]`, where 2^S == size of `B`
2. If the buddy `C` of `B` exists (also free)
   a. Remove `B` and C from list
   b. Merge `B` and `C` to get a larger block `B'`
   c. Goto step 1, where `B` ← `B'`
3. Else (buddy of `B` is not free yet)
   - Insert `B` to the list in `A[S]`

- To find corresponding buddy, just invert nth bit

Key idea: Buddy system will have lesser external fragmentation than dynamic partitioning because there **exists a minimum size of hole in powers of 2**, while dynamic allocation there only **exists arbitrarily small holes of very small value that is unusable.**

Disjoint Memory Allocation

Remove the first assumption that process occupies single contiguous piece of memory.

- General idea here is that paging is about allowing contiguous logical addresses to be stored in disjoint memory spaces (But overhead here from **requisite of a page table to store mappings**).
- **Physical frame size == Page size**

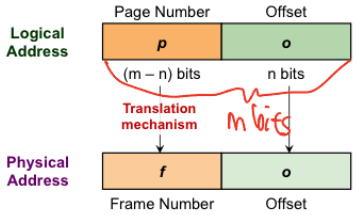
Paging Physical memory is now split into **physical frames** and logical memory of a process is split into **logical pages** of the same size. Logical memory can remain contiguous, but each page now can be mapped into disjoint frames.

Page Table Design

- Keep frame size (page size) as a power-of-2 for bit size addressing
- Physical frame size == Logical page size

Address translation

Given logical address of m bits, frame size of 2^n



Physical address = Frame number * sizeof(physical frame) + offset

Analysis of Paging

- No external fragmentation -> all free pages can be used
- Insignificant internal fragmentation (at most 1 page per process has internal fragmentation)
- Clear separation of logical and physical address -> Allows flexibility and simple address translation

Implementation of Paging

OS stores page information **in PCB** (but store pointer to page table only)

- Need 2 memory access (RAM) for every memory reference (1 to read indexed page table entry to get frame number, 1 to access actual memory item)

Translation Look-Aside Buffer(TLB)

Acts as cache for page table entries -> Use page number to lookup TLB

Fast. 1ns(TLB) vs 50ns(RAM) access time

TLB-Miss vs TLB-Hit RAM is only accessed upon a miss, and TLB is updated after that

Upon context-switch TLB entries are flushed. Hence, when process resumes running state, will encounter many TLB misses to fill up TLB.

Protection of Paging Scheme

To support memory protection, usage of **Access-Right bits and Valid bit**

Access-Right bits each page table entry (PTE) has {writable, readable, executable}. **Every memory access is checked against access right bits in hardware.**

Valid bit is included in each PTE. Indicates whether page is valid to access by process. OS will set the valid bits as memory is being allocated to the process and **every memory access is checked against valid bit in hardware.** (To check if out-of-range or not)

Page Sharing

Naturally, page table can allow several processes to share same physical memory frame. e.g. standard libraries being used by different processes

We can use a **shared bit** in a PTE to indicate if page is shared. Then, implement **Copy-On-Write ideology.**

Segmentation

Splitting of logical memory into different segments -> One each for Text, Data, Heap, Stack

This is because some regions may grow/shrink at execution time.

Each segment given a segment ID(SegID)

- Logical address `<SegID, Offset>`:
  - `SegID` is used to look up `<Base, Limit>` of the segment in a **segment table**
  - Physical Address `PA = Base + Offset`
  - `Offset < Limit` for valid access

Validity check access is valid iff offset < limit

If offset > limit: Segmentation fault(SIGSEGV)

Segment table indexed by SegID, stores <Base, Limit>

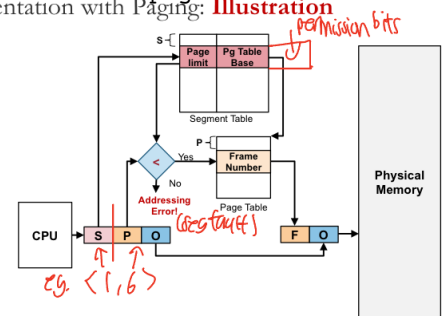
Analysis of Segmentation

- Segments are independent (each segment is an independent contiguous memory space)
- More efficient bookkeeping (entire segment has same access rights)
- Segments can grow/shrink and be protected/shared independently.
- **External fragmentation problem** due to contiguous nature -> Solved by segmentation with paging below

Segmentation with Paging

Each segment is now composed of several pages instead of 1 contiguous memory region. Each segment also now has a page table.

Segmentation with Paging: **Illustration**

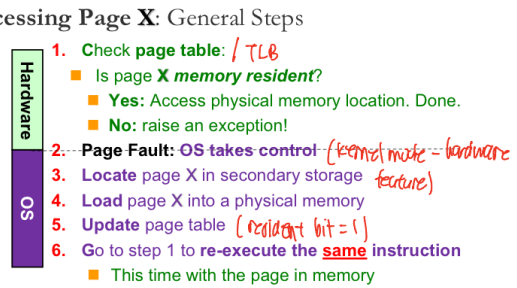


Virtual Memory Allocation

Remove the last assumption. Logical memory space of process >> physical memory now
Following paging, some pages are now in physical memory while the remaining pages are now in secondary storage(Hard disk).

Extended Paging Scheme
Have a **Resident bit** for each PTE. When CPU tries to access a non-resident page, page fault(interrupt) occurs -> Process goes to **BLOCKED** state.

Accessing Page Algorithm



Problem here is that theres always possibility of thrashing: Page fault occurs most of the time. However, this is unlikely due to locality.
Temporal Locality Memory address used now is likely to be used again. Cost of bringing page into memory is amortized.
Spacial Locality Memory addresses close to the address that is used now is likely to be used soon (They are included in one page)

Demand Paging
Process starts with **no memory resident page**, only allocate a page when a page fault occurs.
Advantages
- Fast startup time for new processes
- Small memory footprint

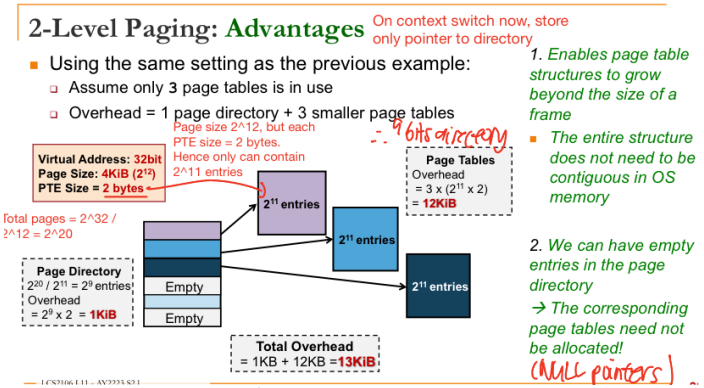
Disadvantages
- Process may appear to be sluggish at start due to multiple page faults
- Page faults may have cascading effect on other processes(i.e thrashing)

Page Table Structure
Problem with huge page table: High overhead and page table can occupy several physical frames in memory

Direct Paging
All entries kept in a single page table, and allocate each process this huge page table. To compute size of this table:
1. Lets say there are 2^p pages in logical memory space
2. We need p bits to specify a page
3. If virtual address has 32 bits, each page/frame being 4KiB = 2^12B, then p = 32-12 = 20
4. This means we have 2^20 pages and PTEs
5. If each PTE is 2bytes, then page table size = 2^20 x 2B = 2MB

However, not all processes use the virtual memory space, so much of this table is unused.

2-level Paging Paging the page table
Split page table into smaller page tables, each with a **page table number**. Need a **Page Directory**, each PTE in directory points to base address of next page table.



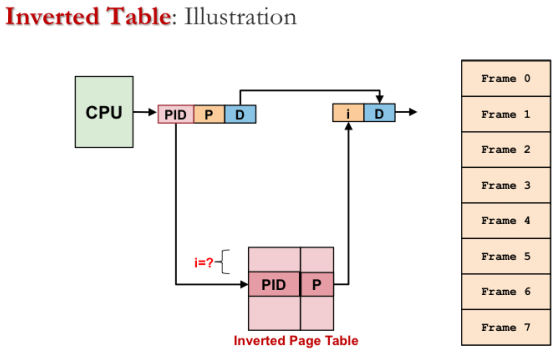
Overhead Calculation:
1. Lets say all tables/pages are 4KB and each PTE is 2B, and virtual address is 32 bits.
2. Since each page/frame is 4KB, 2^12B, offset is thus 12 bits. Hence we only look at first 32-12 = 20 bits to determine **Page Number**
3. Since each page/frame is 4KB, and each entry is 2B, there are a total of 2^12 / 2 = 2^11 entries.
4. That means that the directory has 2^20 / 2^11 = 2^9 entries. This means that it will take up to 2^9 x 2B = 2^10B = 1KB
5. Lets say we only have 3 page tables and 1 directory. Then total overhead = 4KB x 3 + 1KB = 13KB.

Problem
Now need 2 serialized memory access to get frame #
Solution: **MMU caches** in hardware that caches frequent page directory entries

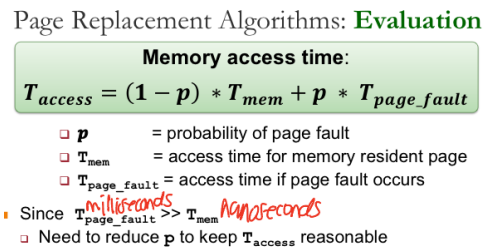
Purpose of MMU is to speed up the page-table walk upon a TLB miss (Recall: TLB caches frequently accessed pages)

Inverted Page Table
Page table that tells us a process is using which frame, from the PID of a process.
In each entry, we store **PID** and **Page Number**.
Advantages
- Huge saving, one table for all processes

Disadvantages
- Slow translation, have to lookup entire table



Page Replacement Algorithms
There is a need to evict a memory page if memory is full and there is a page fault. **Usage of Dirty bit** for PTEs, if page is modified, need to write back to secondary storage(disk).
Evaluation



Optimal Page Replacement(OPT)
Idea is to replace to page that will not be needed again for the **longest period of time**
- Guaranteed **minimum page faults**
- Need future knowledge of memory reference
  - Simply not feasible, hence often used as benchmark against other algorithms

FIFO Page Replacement (Generally bad)

Evict the oldest memory page

- **Queue** OS maintains a queue of resident page numbers
  - Remove first page in queue if replacement needed
- No hardware support needed
- **Belady's Anomaly** Generally, when number of frames increases, number of page faults should decrease. However, for FIFO, number of page faults increases. This is because FIFO does not exploit temporal locality.

Least Recently Used (LRU)

Replace page that has not been used in **longest time,** exploiting temporal locality

- Close to OPT algorithm
- Difficult to implement -> Need substantial hardware support
  - **Using counter**: Each PTE has a "time-of-use" field, store the time counter value whenever page is referenced. Replace page with smallest "time-of-use". **Problem** here is that we need to search through all pages(overhead from search) and possibility of overflow of "time-of-use"
  - **Using stack**: Replace page at bottom of stack. If page is referenced, remove from stack and push on stack again. **Problem** here is that it is **not a pure stack** where entries can be removed from anywhere in stack and hard to implement in hardware.

Second Chance Page Replacement (CLOCK)

Each PTE now has a reference bit.

- Algorithm:
  1. The oldest FIFO page is selected
  2. If reference bit == 0 ➔ Page is replaced. Done.
  3. If reference bit == 1 ➔ Page is skipped (given a 2nd chance)
     - Reference bit cleared to 0
     - Effectively resets the arrival time ➔ page taken as newly loaded
     - Next FIFO page is selected, go to Step 2
- Degenerate into FIFO algorithm
  - When all pages has reference bit == 1
- Adds some notion of recency and works well in practice.

Idea here is that as pointer passes through, it clears any reference bits and the victim is the 1st page with the reference bit = 0

Frame Allocation

If there are N physical frames, and M processes, we need to distribute these N frames to M processes.

Equal Allocation

Each process gets N/M frames.

Proportional Allocation

Each process gets size(process) / size(total processes) number of frames.

Local vs Global Replacement

Local Replacement

Victim page is selected among pages of the process that causes page fault

Advantages:

- Number of frames allocated to a process remains constant -> Performance is stable between multiple runs

Disadvantages:

- If frames allocated are not enough, will hinder progress of a process
- **Thrashing:** Limited to one process, but this process will hog the I/O and degrade performances of other processes

Global Replacement

Victim page is selected among pages of all processes.

Advantages:

- Allows for self-adjustment between processes, processes that needs more frame can get from others that need less

Disadvantages:

- Inconsistent performance between runs
- Malicious processes can affect other processes
- **Cascading thrashing**: One process that thrashes will steal pages from others, resulting in other processes also thrashing

Working Set Model

We want to find the right number of frames for processes. Generally, the set of pages referenced by a process is constant in a period of time due to locality. The number of page faults is minimal until the process transits to a new locality.

- Define a working set window Δ, which is an interval of time
- $W(t, \Delta)$ is the set of active pages in the interval at time t
- We thus want to allocate enough frames for pages in $W(t, \Delta)$ to reduce page fault
- Accuracy of model depends on choice of Δ
  1. Too small: miss pages in current locality
  2. Too large: contains pages from different localities

File Systems

We use external storage to store **persistent** info

General criterias:

Self-Contained Information stored on a media is enough to describe entire organisation

Persistent beyond the lifetime of OS and processes

Efficient provides good management of free and used space, minimum overhead for bookkeeping information

File is a logical unit of information created by a process

File Metadata is the attributes associated with a file ie **Name, Identifier, Type, Size, Protection, Time,date and owner information, Table of content**

File types

- Regular files
  - Contain user information (ASCII files, binary files(executables)
- Directories
  - System files for FS structure
- Special files
  - Character/block oriented

Windows uses file extension to specify file type

UNIX uses magic number embedded at beginning of file to specify file type

File Protection

- Access Control List(ACL): List of user identity and allowed access types
- Permission bits (owner, group, universe {rwx})

File Data

Different Structures

- **Array of Bytes**: Each byte has unique offset from file start
- **Fixed length records:** Each record has a fixed size, this array of records can grow/shrink, and can jump to any using offset
- **Variable length records:** Flexible but hard to locate a record

Access Methods

Sequential Access: Data read in order, starting from beginning. Cannot skip but can rewound

Random Access: Data read in any order, either via read(offset) or seek(offset). UNIX and Windows use seek(offset).

Direct Access: Basically random access but with records(fixed-length)

File Operations

Create:	New file is created with no data
Open:	Performed before further operations To prepare the necessary information for file operations later
Read:	Read data from file, usually starting from current position
Write:	Write data to file, usually starting from current position
Repositioning:	Also known as seek Move the current position to a new location No actual Read/Write is performed
Truncate:	Removes data between specified position to end of file

Representation of Open File

1. File pointer (keep track of current position within file)
2. File descriptor (unique identifier of file)
3. Disk Location (Actual file location on disk)
4. Open count/Reference count (how many process has this file opened)
    a. Remove entry in table iff open count == 0

Three Table Approach for File Information

1. **Per-process open-file table**
    a. Stores file descriptor table
    b. Each entry points to the system-wide open file table entries
    c. If one process opens a file then forks, there will be two fds pointing to the same system-wide open file entry. Thus, share same offset.
2. **System-wide open-file table**
    a. Keeps track of all open files in system
    b. Each entry points to a V-node entry
    c. If one process opens a file then forks, only 1 entry here.
    d. Stores **file offset**, **reference count**
3. **System-wide V-node(virtual node) table**
    a. To link with file on physical drive
    b. Contains information about physical location of file

Directory

Helps user group files, and helps system keep track of files

Single Level all files contained in root directory

Tree-Structured Since directories can contain directories, they are like trees or subtrees. Files are leaves.

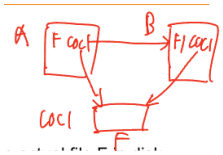
- **Absolute pathname**: Directory names followed from root to final file location
- **Relative pathname**: Directory names followed from current working directory(CWD)

Directed Acyclic Graph(DAG) We are able to "skip levels", ie share a file/directory such that it appears in multiple directories but refer to same copy of actual content.

Hard Link (Cannot use for directories)

A and B has **separate pointers** point to he actual file F in the disk.

UNIX command: ln



Advantages:

- Low overhead, only pointers added in directory

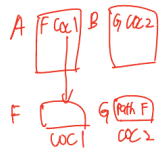
Disadvantages:

- Problem during deletion. -> Only delete file if reference count == 0

Soft Link (Can use for directories)

Usage of a special link file, G, that contains the path name of file F. When G is accessed: Find out where is F, then access F.

UNIX command: ln -s



Advantages:

- Simple deletion

If symbolic link deleted, only G is deleted, not F.

If F is deleted, G remains but will not be working(dangling link)

Disadvantages:

- Larger overhead: Special link file takes up actual disk space

General Graph not desirable, because it is

- **Hard to traverse**: Need to prevent infinite loops.
- **Hard to determine when to remove a file/directory**

But possible in UNIX, using soft link(symbolic link)

File System Implementations

File Information, data and allocation

In the process of allocating file data, a good file system must keep track of logical blocks, allow efficient access, and ensure disk space is utilized effectively.

Contiguous Allocation

Allocate consecutive disk blocks to a file. However, random allocation to start block due to possible snapshot of memory

- Easy to keep track
- Fast access (only need to seek to first block)
- **External fragmentation**
- **File size** need to be specified in advance

Linked List

Maintain disk blocks as a linked list. Each disk block stores next disk block number, alongside file data. File information will store first and last disk number for the file.

- **No external fragmentation**
- **Slow access**: O(n) time since need to traverse through list
- Less usable space: Part of disk block used for pointer to next block
- Less reliable: Fails if one pointer is incorrect

Linked List V2.0

Same as linked list, but now all pointers are stored in a **File Allocation Table(FAT)** that is stored in memory(RAM)

- **Faster access**: Linked list traversal now in memory
- **Takes up space**: FAT tracks all disk blocks. This number can be huge when disk is large, consuming valuable memory space(RAM)

Indexed Allocation

Use one block per file to store an array of indices containing disk block address for that index. ie. IndexBlock[N] == Nth block address

- Lesser memory overhead(only index block needs to be in memory)
- Fast direct access
- Limited maximum file size (Max number of blocks == number of index block entries)
- Overhead due to index block

Free Space Management

Methods to know which blocks are free for allocation

Ideas:

- **Allocate**: Need to remove free disk block from free space list. Need to do this when file is created or enlarged(appended)
- **Free**: Add free disk block to free space list. Need to do this when file is deleted or truncated

1. **Bitmap**
    a. Value of 1 means free, 0 means occupied
    b. **Provides a good set of manipulations**
    c. Problem: Need to keep in memory for efficiency reason(overhead)
2. **Linked List**
    a. Linked list of blocks used to contain indices of free blocks
    b. Easy to locate free block, and only first pointer is needed in memory

c. High overhead - But can mitigate by using free blocks to store this information

Directory Structure

Goal here is to provide a mapping from file name to file information. Sub-directories are stored as file entry with a special type.

Linear List

Directory contains a list of entries, one entry per file. Entry contains file name(and possibly other metadata) and file information or pointer to file information.

- **Requires linear search**: O(n) lookup -> terrible
- Solution: Cache the latest few searches

Hash Table

Directory contains a size N hash table. We hash the file name from 0 to N-1, and use chained collision resolution.

- Fast lookup -> O(1) constant time lookup
- **Limited size for hash table**
- Depends on good hash function

File Information(FAT vs EXT2 file system)

File Allocation Table(FAT)

Cached in memory(RAM), one entry per data block/cluster.

FAT Entries

- **FREE** for unused block
- Block number for next block
- EOF code (ie NULL)
- BAD (unusable, block corrupted)

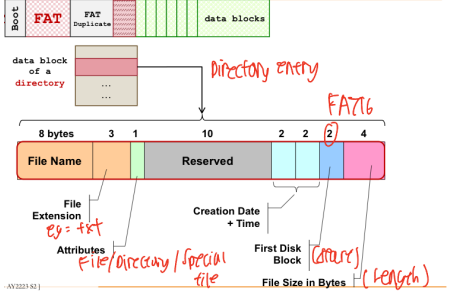
Directory

A special type of file. Each file or subdirectory here is represented by a directory entry

Directory Entry

Fixed-size 32 bytes per entry

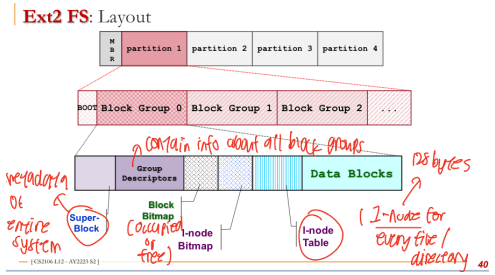
- Name + Extension (11 chars, 11 bytes total)
- Attributes (1 byte)
- Reserved (10 bytes)
- Creation date (2 bytes) and Time (2 bytes)
  - Year range 1980 to 2107
  - Accuracy of second is +- 2sec
- First Disk Block Index
  - 2 bytes for FAT16, 4 bytes for FAT32
- File size(Length): 4 bytes



Tracing Process for FAT

1. *Get required directory table from disk blocks (likely in cache)*
2. *Get required directory entry and find first disk block number*
3. *Use FAT to find remaining disk block numbers, until EOF encountered*
4. *Use the numbers to do disk access on file data blocks. If subdirectory, repeat step 1.*

Linux Ext2



I-node structure

Each I-node is 128 bytes.

Each file/directory has 1 I-node.

Data Block Pointers (Total 15)

First 12 pointers -> Direct blocks

13th block -> Single indirect block

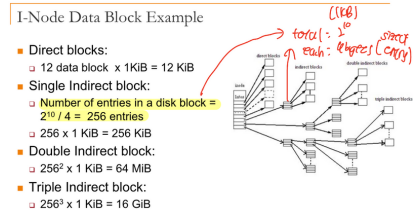
14th block -> Double indirect block

15th block -> Triple indirect block

Multi-level data blocks

Allows for fast access to small file, flexibility in handling huge files.

Assuming we have 4 bytes block address(size of page entry), 1KiB disk block



Directory Structure

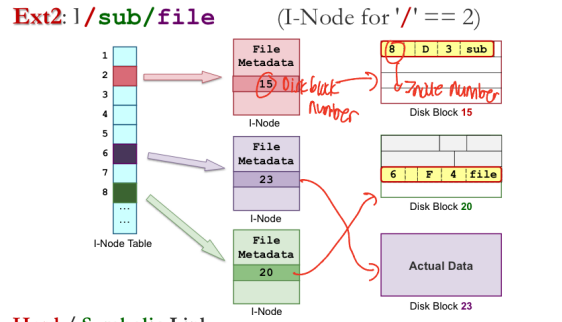
Directory is just another file. Within this directory, contains a linked list of directory entries for file/subdirectories information within this directory.

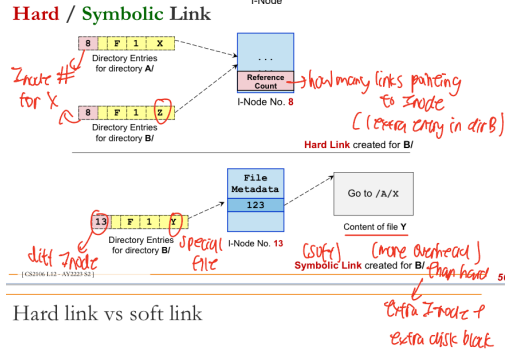
Each **directory entry** contains:

- I-node number for this entry
- Size of this entry so we can locate next entry
- Length of file/subdirectory name
- Type (File / subdirectory)
- File/Subdirectory name (up to 255 chars)

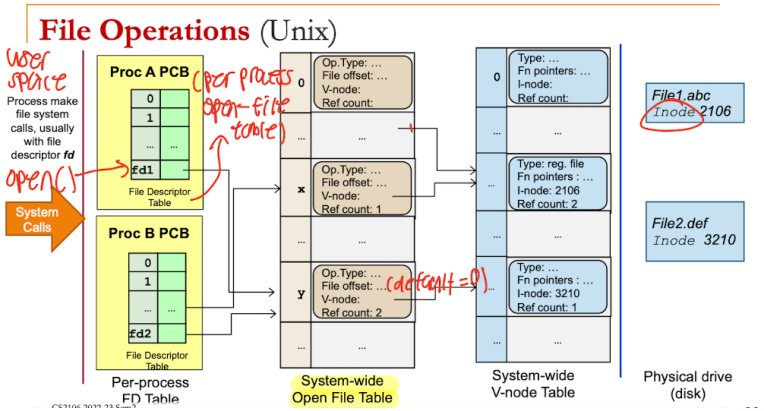
Tracing Process for Ext2

1. *Get root directory I-node. Generally fixed, eg 2*
2. *If next path is a subdirectory*
   a. *Locate directory entry in current directory*
   b. *Get I-node number, then read I-node, go to disk block and find name and read next I-node number*
   c. *Set current directory to next pathname*
   d. *Repeat step 2*
3. *Else it is a File*
   a. *Locate directory entry in current directory*
   b. *Retrieve I-node number, then read it and go to disk block to obtain data*

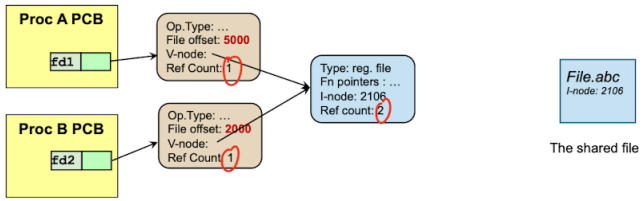




Hard link vs soft link

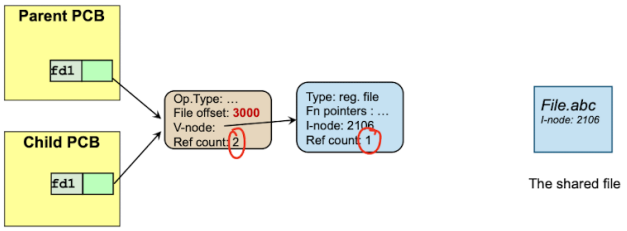


- A file is opened twice from two processes
  - I/O can occur at independent offsets



Process Sharing File in Unix: Case 2

- Two processes using the same open file table entry
  - Only one offset ➔ I/O changes the offset for the other process



2. [Page table structure] [Adapted from AY1920S2 Final – Page Table Structure] The Linux machines in the SoC cluster used in the labs have 64-bit processors, but the virtual addresses are 48-bit long (the highest 16 bits are not used). The physical frame size is 4KiB. The system uses a hierarchical direct page table structure, with each table/directory entry occupying 64 bits (8 bytes) regardless of the level in the hierarchy.

Assume Process *P* runs the following simple program on one of the lab machines:

```
1.  #include <stdio.h>
2.  #include <stdlib.h>
3.  #define ONE_GIG 1<<30
4.  #define NUM_PAGES 1<< 18
5.  void main(){
6.    char* array = malloc(ONE_GIG);
7.    int i;
8.    for (i=0; i<NUM_PAGES; i++)
        array[i<<12]='a';
9.    printf("0x%lX\n", array);
10.   // point of interest
11.   free(array);
12. }
```

At line 9, the program prints out the value of variable **`array`** in hexadecimal:

Consider the point in time when the process execution reaches the point of interest (line 10 in the listing). Answer the following questions:

a. If we use a single-level page table, how much memory space is needed just to store the page table for process *P*?
b. How many levels are there in the page-table structure for process *P*?

c. How many **entries in the root page directory** are holding information related to process *P*'s **dynamically allocated data**?
d. How many **physical frames** are holding information related to process *P*'s dynamically allocated data in the **second level** of the hierarchical page-table structure (next after the root)?
e. How many **physical frames** are holding information related to process *P*'s dynamically allocated data in the **penultimate level of the page-table structure** (i.e., the level before the last one).
f. How many **physical frames** are holding information related to process *P*'s dynamically allocated data in the **last level of the page-table structure?**

ANS:

a. **The maximum memory space size for *P* is 2^{48} bytes.**

Splitting this memory space into 4KiB page gives us $2^{48} / 2^{12} = 2^{36}$ pages
So, a single level page table has 2^{36} PTEs, each at 8 bytes => page table is 2^{36} x 8 bytes = 2^{39} bytes (512 GiB!)

b. **If we are keeping the "page directory" at each level to a single page, the branching factor is**

$2^{12} / 8$ bytes = $2^9 = 512$

i.e. each directory can points to 512 next level directories. [Note: we assume that a memory pointer is the same size as a PTE for simplicity].

This tell us that we use 9-bit of the virtual address for each level. Since the last 12 bits of the virtual address is used as the page offset. So, we are left with 48-12 = 36 bit of address:

ceiling(36 bit address / 9-bit per level) = 4 levels.

Note that this is considering the entire memory space for process *P*, i.e. not restricted to just the dynamically allocated space used by "`array`".

With the above information, we can now look at the address of array closer. The virtual address has 5 components:

Root Level Idx	Second Level Idx	Penultimate Level Idx	Last Level Idx	Page Offset
9 bits	9 bits	9 bits	9 bits	12 bits

Give address of array = `0x7F9B55226010`

Root Level Idx									Second Level Idx									Pen.Level Idx									Last Level Idx									Page Offset 12bits												
0	1	1	1	1	1	1	1	1	0	0	1	1	0	1	1	0	1	0	1	0	1	0	1	0	0	1	0	0	0	1	1	0	0	1	0	1	1	0	0	0	0	0	0	0	0	0

Component	Effective Value
Page Offset	$0x010 = 16_{10}$
Last Level Idx	$0x26 = 38_{10}$
Pen.Level Idx	$0xA9 = 169_{10}$
Second Level Idx	$0x6D = 109_{10}$
Root Level Idx	$0xFF = 255_{10}$

c. **`Array` span across $2^{18} + 1$ pages (1 GiB data / 4 KiB page = $2^{30} / 2^{12} = 2^{18}$ pages + 1 due to page misalignment).**

Since we have a branching factor of 2^9 (see (b)), we will need only 1 entry after 2 levels. So there is only 1 entry needed at the root level (highest level).

d. **1 page (only 2 entries required!)**
e. **2 pages for 2nd level page table entries**
ceiling(513 last level page table pages / 2^9) = 2 pages

From the component value table, you can see that the first last level index is at position 38, i.e. the first (512-38) = 474 entries is in the first page and the rest (38+1) entries spill over to the next page.

f. **It would seems that we have 2^{18} last level PTEs for the `array`: allocated 1GiB ➔ split into 4KiB pages = $2^{30} / 2^{12} = 2^{18}$ pages. However, the printed address shows that the `array` is unfortunately not allocated at the page boundary (start of a page should have 0 at the 12 least significant bits of the virtual address). From the components value table, you can see that the first byte of array is +16 offset from the top of the page.**

This gives us 2^{18}+1 pages in total.

We need $2^{18} / 2^9 + 1 = 2^9 + 1 = 513$ frames to hold ($2^{18} + 1$) PTE entries. The last frame will be holding just 1 PTE.

Discussion is in reverse order, from (f) to (c)