10 pages CS2106 ~~Midterm~~ Finals notes that will surely be sufficient for an Ez A+

**Lecture 2: Process Abstraction**

* **Process abstraction** 🡪 abstracts the execution of a program
  + Memory 🡪 text, data, stack, heap | HW 🡪 GPR, PC, SP, FP | OS context 🡪 Process ID and process State
* **Function calls** 🡪 control flow and how data is manged

Graphical user interface, text, application

Description automatically generatedGraphical user interface, text, application, email

Description automatically generated

* + Need Stack memory and frame 🡪 region to store info during func invocation
    - Stack pointer can be either last filled slot, or first avail slot (usually the case)

Graphical user interface, text, application

Description automatically generated Diagram

Description automatically generated

* + - Stack frame is added when func is invoked and removed when func call ends (main() doesn’t create!)
      * Only needs to be maintained if its going to continue execution 🡪 return factorial(N-1, N\*res);
  + Stack frame setup and teardown 🡪 handled by compiler
    - Table

      Description automatically generated Table

      Description automatically generated
      * Stack frame contains address of following statement after callee() 🡪 return PC
  + Frame pointer ($FP) 🡪 points to static location in stack frame to facilitate access of stack frame items
    - We use it with a displacement to access different stack frame items
  + Saved registers 🡪 use memory to temp hold GPR values so that it can be used for other purposes
    - Saved regs are used to restore the GPR values to how the original function state
    - Needed as callee does not know which GPR the caller is using, might accidentally overwrite
    - No need for main() as it is likely to be invoked by OS, OS will save the GPR needed
* **Dynamically allocated memory** 🡪 malloc() (in the HEAP)
  + Allocated only at runtime (cannot place in data), no definite deallocation timing (cannot in stack)
* **PID** 🡪 unique among processes
* **Process State** 🡪 indicates the execution status, handled by scheduler
  + Diagram

    Description automatically generated
    - Create 🡪 new process created
    - Admit 🡪 process ready to be scheduled for running
    - Switch: scheduled 🡪 process selected to run (need context switch)
    - Switch: release CPU 🡪 process gives up CPU voluntarily or preempted by scheduler (context switch)
    - Event wait 🡪 process requests event/resource/service which is not avail/in progress
      * E.g. system calls or waiting for I/O
    - Event occurs 🡪 process can continue
* **PCB and Table** 🡪 stores all processes and their contexts (for switching)
  + Diagram

    Description automatically generatedDiagram

    Description automatically generated
  + HW context only updated when swapped out, Mem context is a pointer, OS context contains scheduling info
* **Exceptions** 🡪 synchronous (only can happen due to program execution)
  + e.g. divide by 0, illegal memory address
  + handled by exception handler automatically
* **Interrupts** 🡪 asynchronous (caused by external events and can happen any time)
  + e.g. mouse or keyboard inputs 🡪 Handled by interrupt handler automatically

Diagram

Description automatically generatedText

Description automatically generated with medium confidence

* + Interrupts only handled when the whole instruction is complete
  + OS sets up IVT which is looked up in HW upon an interrupt
* **System calls** 🡪 provides a way of calling services in kernel (via API to OS)
  + Requires a change from user mode to kernel mode (via TRAP or syscall) and back again (after syscall)
    - There are diff syscall handlers for each syscall number which actually handles it

**Lecture 3: Process Abstraction in UNIX**

* **Unix process abstraction 🡪**

Graphical user interface

Description automatically generated with low confidenceDiagram

Description automatically generated

* **fork()** 🡪 creates a new process
  + returns PID to parent process and 0 to child process
  + Child is duplicate of current executable image (same code, address space)
    - Will copy the PC and continue exec from there + Has indpt mem space 🡪 data copied not shared
  + How to differentiate? 🡪 PID, parent PID, fork() return value
  + Fork bomb will cause out of memory
  + Steps 🡪 Create address space of child process 🡪 allocate p’ = new PID 🡪 create kernel process data structures 🡪 copy kernel environment of parent process 🡪 init child process context 🡪 copy memory regions from parent (code, data, stack Expensive!) 🡪 acquires shared resources (open files, directory) 🡪 init HW context for child process (GPR etc) 🡪 child ready (added to scheduler queue)
    - Memory copy is expensive 🡪 can just reuse the same memory region as parent except when a particular region is written to then duplicate
  + Process tree 🡪 Diagram

    Description automatically generated
* **Using forks with exec()** 🡪 can be used to execute another program using exec()
  + Will replace the current image with a new one 🡪 code and data replaced but PID same (PPID also)
    - Everything else written in code is replaced and not executed (including before and after)
  + Child process will do the task using exec() while parent still around the accept new requests
* **Process termination** 🡪 use exit(int status)
  + Normal termination = 0
  + Most system resources used by process will be released on exit
    - Except PID and status for syncing with parent (need retain for when parent calls wait())
  + Child will become a zombie after exit() 🡪 need wait from parent to remove
* **Parent/child sync** 🡪 parent can wait for child to terminate using wait(int \*status)
  + Will return the PID of the child process and stores the exit status of the terminated child process
  + The status of the exited child will be stored in &status, can use NULL if not needed
  + The standard wait() will block the parent until the first child terminates
  + cleans up the remainder of the child resources (those not removed on exit()) and kills zombies
  + other variants 🡪 waitpid() for specific child PID, waitid() for any child process to change status
* **Zombies and orphans processes** 🡪 diff terms for children
  + Zombie is when child terminates but parent did not wait() 🡪 might fill up process table
  + Orphan is when parent terminates before child 🡪 init process becomes parent of child, which will call wait() to clean up

**Lecture 4: Inter-Process Communication**

* **IPC** 🡪 for different processes to share info as memory space is independent
* **Shared memory** 🡪 implicitly communicate via reads/writes to shared variables
  + Steps 🡪 can be extended to > 2 processes using the same shared memory region
    - Process P1 creates a shared memory region M 🡪 Process P2 attaches memory region M to its own memory space 🡪 P1 and P2 can now communicate using memory region M
      * M behaves very similar to normal memory region
      * Any writes to the region can be seen by all parties that share the region
  + Pros 🡪 OS involved when M is created and attached to
    - Any reads and writes to it does not involve OS 🡪 efficient
  + Cons 🡪 Limited to single machine, needs sync
  + Race condition example 🡪 a typical counter++ is not atomic, involves read and writes
    - Graphical user interface, application, table

      Description automatically generated
    - Formula 🡪 (#instructions !) / (#programs \* # instructions order must be fixed!)
      * Note that inc-LD must always be before inc-ST and same for dec-LD and dec-ST
    - Even though there are 3 posb values, 5 is still the likeliest value as since the program is so short, it is unlikely for any interrupts + context switches to happen
    - Our goal is just to remove all “wrong” outcomes, while allowing as many good ones as possible
      * However restricting too much might also be inefficient as OS cannot optimize
  + POSIX shared memory 🡪 very similar to generic one
    - Except M must be detached from memory space after use and then destroyed by any one process (M can only be destroyed if not attached to any process)
    - Master program example 🡪 uses shm[0] to see if worker process is done

Text

Description automatically generatedText

Description automatically generated

Text

Description automatically generated

* **Message passing** 🡪 Explicit communication thru exchange of messages
  + Steps:
    - Process P1 prepares a message M and send it to Process P2 🡪 Process P2 receives the message M 🡪 Message sending and receiving are usually provided as system calls
  + Properties:
    - Msg to be sent is stored in kernel mem space
    - All send/receive operations need go thru OS 🡪 inefficient
    - Diagram

      Description automatically generated
  + Pros 🡪 Applicable beyond a single machine, portable across platforms, easier sync
  + Cons 🡪 inefficient as OS needed for every step, harder to use as data format is fixed
  + Two Different types of naming schemes 🡪 direct and indirect
    - Naming scheme (Direct Communication) 🡪 when you want to explicitly name the other party
      * send(p2, Msg) to send Msg to Process 2, receive(p1, Msg) to receive Msg from Process 1
      * one link per pair of communicating processes + need to know the identity of the other party
    - Naming scheme (Indirect Communication) 🡪 used a shared mailbox to pass messages
      * send(mb, msg) to send msg to mailbox mb, receive(mb, msg) to receive msg from mailbox mb
      * mailbox can be shared among multiple processes
  + Two different types of sync behaviors 🡪 blocking and non-blocking
    - Blocking Primitives (Synchronous)
      * send(): sender is blocked until the message is received
        + msg kept by sender until receiver ready then directly copied to its address space
      * receive(): receiver is blocked until a message has arrived
    - Non-Blocking Primitives (Asynchronous)
      * send(): sender resumes operation immediately
        + message is buffered by system 🡪 sender is never blocked

any receive() performed later will be completed immediately

buffer is under OS control 🡪 no need sync ourselves

might need to block/return with error if buffer capacity is full 🡪 not truly asynchronous

* + - * receive(): if message hasn’t arrived yet, proceeds empty-handed but doesn’t block

Diagram

Description automatically generatedDiagram

Description automatically generated

* **Pipes** 🡪 communication channels used in UNIX
  + 3 communication channels
    - 1 in 🡪 stdin (usually kb input)
    - 2 out 🡪 stdout (usually screen) + stderr (used to print out error messages)
  + Pipes as an IPC mechanism (message passing) 🡪 share the same pipe between 2 processes P and Q
    - Producer-Consumer rs: P produces bytes, Q consumes them
    - FIFO: data must be accessed in order
  + Properties 🡪 implicit sync where sender does not block unless buffer full, reader waits when empty
  + Example 🡪 pipe(int fd[])

Text

Description automatically generatedText

Description automatically generated

* **Signals** 🡪 Async IPC which can happen at any time from receiver’s POV
  + Need handle the signal using default handler or user supplied handler
  + Example 🡪
    - Text

      Description automatically generated

**Lecture 5: Process Scheduling**

* **Concurrent execution** 🡪 multiple processes progress in execution (at the same time)
  + Needs context switching when processes are interleaved
    - Chart, diagram, box and whisker chart

      Description automatically generated
* **Scheduling** 🡪 how to choose which process to run if #ready-to-run > #CPU
  + The scheduler will pick from the ready queue among the ready processes
    - Typical process behavior 🡪 seq of CPU and I/O bursts ending with I/O burst
    - Processing env 🡪 batch (no responsiveness)/ interactive (need resp)/ real time (need meet deadlines)
      * Criteria for all 🡪 must be fair (no starvation) + all utilization (no wastage of computer system)
  + When is scheduling performed 🡪
    - Non-preemptive 🡪 process in running state stays until it blocks or gives up CPU voluntarily
    - Preemptive 🡪 process given fixed time quota, after which it is suspended and another process picked
      * Process can block or give up early
  + Scheduling steps 🡪
    - • Scheduler is triggered (OS takes over) 🡪 If context switch is needed: Context of current running process is saved and placed on blocked queue / ready queue 🡪 Pick a suitable process P to run based on scheduling algorithm 🡪 Setup the context for P 🡪 Let process P run
* **Batch processing scheduling** 🡪 no user interaction, typically non-preemptive scheduling
  + Main criteria:
    - Turnaround time 🡪 Total time taken, (finish - arrival time)
      * waiting time: time spent ready for execution but waiting for CPU
      * response time: time difference between arrival and first time task receives CPU time
    - Throughput: 🡪 Number of tasks finished per unit time (Rate of task completion)
    - Makespan: 🡪The total time from start to finish to process all tasks
    - CPU utilization: 🡪 Percentage of time when CPU is working on a task
  + FCFS 🡪 tasks stored in FIFO queue based on arrival time
    - Pick first task in q to run until done/blocked 🡪 blocked task removed and placed at back of q when rdy
    - Guaranteed no starvation 🡪 number of tasks in front of X is strictly decreasing
    - Might lead to long avg wait time 🡪 YTF + drinks store example (Convoy effect)
  + SJF 🡪 task with smallest total CPU time first
    - Need know total CPUtime for task 🡪 at the start its just a constant since no past history
      * Text

        Description automatically generated
    - Guaranteed to minimize avg wait time given fixed set of tasks
    - Might have starvation if short tasks keep coming in 🡪 long task never get the chance to run
  + SRT 🡪 preemptive ver of SJF that uses remaining time (selects job with SRT)
    - Might have starvation 🡪 new jobs can preempt currently running job
    - Provides good service for short job even when it arrives late
    - Identical to SJF if all tasks arrive at same time
* **Interactive env scheduling** 🡪 focus on response time and predictability (need preemptive scheduling)
  + Uses a timer interrupt that goes off periodically handled by OS that cannot be intercepted
    - OS scheduler is invoked every timer interrupt 🡪 usu 1 to 10ms (short = OS overhead, long = wait time)
    - Time quantum 🡪 execution duration given to process (must be multiples of timer interrupts)
      * Usually 5 to 100ms (too short means a lot of context switching = inefficient)
  + Round Robin 🡪 preemptive version of FCFS
    - Tasks run until time quantum elapsed/ gives up CPU voluntarily/ blocks
      * If blocked 🡪 placed in another q to wait then placed back in this queue, else placed in back
    - No starvation 🡪 given n tasks and quantum q, task will get CPU by (n-1)q time
    - Not really responsiveness as new tasks not given priority
  + Priority scheduling 🡪 assigns priority to task where highest priority selected first
    - 2 version 🡪 preemptive ver can preempt running process with lower priority, and non-preemptive
    - New tasks given high priority 🡪 responsive
    - Might have starvation for low priority processes (even worse for preemptive ver)
      * solve by decreasing its priority after or not consider it until next round after each time quantum
    - Priority inversion might happen when a lower priority process locks a resource that a higher prio wants
      * Solved with priority inheritance 🡪 the low priority temp inherits priority of high priority until it can unlock so high prior can use, after that priority is restored
  + MLFQ 🡪 “biases” towards I/O bound, CPU intensive get penalized in priority
    - If Priority(A) > Priority(B) 🡪 A runs, If Priority(A) == Priority(B) 🡪 A and B runs in RR
    - New job = highest priority, prio reduced when TQ finish, priority retained if give up/blocks before TQ
      * Can be abused if program quits just before TQ ends to retain high prio
    - Might have starvation as new processes have high priority 🡪 run first
    - Approx SJF 🡪 MLFQ treats new jobs as short and give it highest prio. If long, it gets preempted and its prior lower. If short, it completes quickly before its prio becomes low.
  + Lottery scheduling 🡪 give out tickets for system resources, when need choose, a random one is chosen
    - Process holding X% of tickets 🡪 wins X% of lottery held 🡪 use resource X% of the time
    - Responsiveness 🡪 newly created process can participate in next lottery
    - Flexible 🡪 can set prio by giving more tickets, set ticket pool for each resource, give tickets to children

**Lecture 6: Synchronization primitives**

* **Race condition** 🡪 when multiple processes execute concurrently interleaving AND share modifiable resource
  + Result will be non-deterministic as outcome depends on order in which shared resource is modified
  + Solution 🡪 Need synchronized access to the shared modifiable resource
* **Critical section** 🡪 the code segment with race condition, at any point only ONE process inside CS
  + Properties of good CS:
    - Mutual Exclusion (only 1) 🡪 If process Pi is executing in CS, all other processes cannot enter
    - Progress (at least 1) 🡪 If no process is in a CS, one of the waiting processes should be granted access.
    - Bounded wait (give others a chance) 🡪After process Pi request to enter CS, there exists an upper bound on the number of times other processes can enter the CS before Pi.
    - Independence 🡪 Process not executing in CS should never block other process.
  + Incorrect synchronization symptoms
    - Incorrect output/behavior 🡪 Usually due to lack of mutual exclusion
    - Deadlock 🡪 All processes blocked (no progress)
    - Livelock 🡪 Typically, processes are not in a blocked state, Processes keep changing state to avoid deadlock but make no other progress
      * Usually related to deadlock avoidance mechanisms
    - Starvation 🡪 Some processes are blocked forever
* **High-level programming lang implementations** 🡪 no HW/OS, just code

Diagram

Description automatically generated Diagram

Description automatically generated

* + Bad Attempt 1 🡪 use a shared lock that anyone can take and disable interrupts during CS
    - whole system stall (if CS buggy) + busy waiting + requires permission to disable interrupts
  + Bad Attempt 2 🡪 use a turn variable, loops while not this process’s turn, pass the turn after CS
    - P1 starves if P0 never enters CS 🡪 violates independence

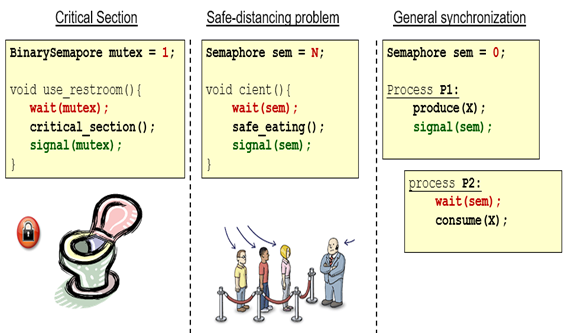
Diagram

Description automatically generated Diagram

Description automatically generated

* + Bad attempt 3 🡪 use length 2 want[], loops while other process wants, set want to 0 after CS
    - Solves independence 🡪 other process can entire CS if P0 or P1 not around
    - Has Deadlock if both want[0] and want[1] = 1
  + Good attempt 4 🡪 Attempt 2’s turn solves deadlock, attempt 3’s wait[] solves independence
    - Works as P0 blocks only if P1 both wants and its P1’s turn
    - the while loop is repeatedly tested instead of being blocked (busy waiting) + too low-lvl + not general
    - Note: want needs to be before turn else will violate mutual exclusion
* **Assembly level implementations** 🡪 use atomic instructions to prevent interleaving
  + From HLL attempt 1 🡪 we want the lock to be atomic so that its not overwritten
    - Use TestAndSet(Lock) 🡪 takes memory addr M, returns current content at M and sets &M to 1
    - Have a while loop with TestAndSet(Lock) == 1 before CS, and lock = 0 after CS
    - Works but uses busy waiting (waste of CPU) + bounded wait not guaranteed
* **Semaphore** 🡪 used to block processes and unblock it when needed
  + Semaphore S contains int value and has 2 atomic operations 🡪
    - Wait(s) 🡪 If S <= 0, blocks (go to sleep); Decrement S (if not blocked)
    - Signal(s) 🡪 Increments S; Wakes up one sleeping process if any; This operation never blocks
  + Invariant 🡪 Scurrent = SInitial + #signal(S) - #wait(S) (only waits that are complete, not blocked)
    - If semaphore is binary, its just like a critical section
  + Can still be deadlocked if used incorrectly 🡪 e.g. both P and Q set to 1 initially
    - Diagram

      Description automatically generated
  + Uses of Semaphore:
    - Critical section 🡪 uses a binary semaphore (toilet lock example)
    - Safe distance problem 🡪 initially set to N (number to be allowed in)
    - General sync 🡪 if we want B to run only after A, we wait(s) before B and signal(s) after A
      * can be implemented with 1 mutex and 2 lock variables

 Text

Description automatically generated

Timeline

Description automatically generated

**Lecture 7: Threads and Synchronization**

* **Threads** 🡪 alternative to processes
  + Why threads
    - Processes are expensive 🡪 requires memory during fork() and context switches (need syscalls)
    - Hard for processes to communicate with each other, need IPC
  + Relation to processes
    - A single process can have multiple threads 🡪 to the same process, just add more threads of control
    - Each thread must have their own stack, CPU registers, and ID 🡪 Text, global data, heap, resources can share
    - Process context switch need switch OS/HW/mem context
      * Thread only need switch the registers and stack (just change the FP and SP registers) 🡪 lightweight
  + Benefits
    - Economic 🡪 uses less resources than having multiple processes
    - Resource sharing 🡪 threads share most of the resources of the process, no need IPC to pass info
    - Responsiveness 🡪 multithreaded programs appear more responsive
    - Scalability 🡪 multithreaded program can take advantage of the of multiple CPUs
  + Problems
    - System call concurrency 🡪 might have parallel sys call (need guarantee correctness)
    - Process behavior 🡪 what happens if one thread forks() or exec() or exit() (OS dependent)
* **Thread Models** 🡪 User, Kernel and Hybrid Threads
  + User Threads 🡪 Each process keeps the thread table
    - Pros 🡪 threads are library calls, makes it OS independent, more configurable and flexible (scheduling policy)
    - Cons 🡪 OS not aware of threads, scheduling is at process level (if thread blocked, whole process blocks)
      * Can never exploit multi-threaded processers
  + Kernel Threads 🡪 The kernel keeps track of the process and thread table
    - Kernal might make use of threads for its own execution
    - Pros 🡪 Thread-level scheduling is possible (instead of by process)
    - Cons 🡪 threads implemented in OS (thread operation is handled as syscalls), less flexible
  + Hybrid Threads 🡪 have both user and kernel threads but controlled by kernel threads
    - OS schedule on kernel threads, user threads are bound to kernel threads (can have 1:1, many:1, 1:many)
    - Pros 🡪 flexible (can limit concurrency of any process/user)
* **POSIX threads** 🡪 common API used
  + Functions defined but not the implementations 🡪 can be implemented as user/kernel threads
    - Functions: pthreads\_create, pthread\_exit, pthread\_join (to wait for the children threads) etc
* **Classic synchronization problems**
  + Producer consumer 🡪 processes share buffer of size K, producers only can produce when not full vice versa
    - Busy-waiting solution 🡪 count = in = out = 0, mutex = S(1), canProduce = TRUE and canConsume = FALSE
      * Lots of wasted CPU usage stuck in the while loop
    - Blocking solution 🡪 in = out = 0, mutex = S(1), notFull = S(K), notEmpty = S(0)
      * Wait(notFull/notEmpty) 🡪 forces producers/consumers to sleep
      * Signal(notFull/notEmpty) 🡪 1 consumer/producer wakes up 1 producer/consumer

Text

Description automatically generated Text, whiteboard

Description automatically generated

* + - Message passing solution
      * Diagram

        Description automatically generated with medium confidence
  + Reader writers 🡪 processes share data structure D, readers read from D, writers write into D
    - Simple solution 🡪 reader tasks that arrive later than a writer task can jump queue (writer starve)
      * Graphical user interface, text, application

        Description automatically generated
    - Revolving door 🡪 have a revDoor (init to 1) that blocks all latecomers once writer waits
      * Writer: add “wait(revDoor);” at start, “signal(revDoor);” at end
      * Reader: add “wait(revDoor);” and “signal(revDoor);” at start
  + Dining philisophers 🡪 N philosophers, N chopsticks, need deadlock and starve-free way for all to eat
    - Incorrect solutions 🡪 if all pick up left at once (deadlock), if we order the philosophers (no concurrency), if a neighbour picks up a chopstick but is blocked (deadlock)
    - Possible solutions:
      * Make one right handed 🡪 R either can eat, right taken(everyone else eating), left taken (will be released)
      * Limited eater 🡪 only allow N-1 to eat at once, will not have deadlock
      * Tanenbaum 🡪 works as each philosopher can block itself and is unblocked by others (do not allow multiple processes to wait on the same semaphore)

**Lecture 8: Memory Abstraction**

* **Memory** 🡪 Assume that RAM consists of byte addressable units (physical address) that are contiguous
  + Memory usage 🡪 can grow and shrink during execution
    - The text (code), data, heap and stack (not known at compile time) all need to fit in memory
    - Transient data 🡪 only for limited duration (func call) e.g. func params, local vars
    - Persistent data 🡪 valid for the duration of the program or deallocated e.g. global vars, malloc
  + OS 🡪 allocates, manages, protects from other processes, provides syscall, mem space for internal OS use
* **Using physical address** 🡪 use the actual address that we need
  + Pros: Straightforward memory access (no mapping needed) 🡪 address fixed at compile time
  + Cons: hard to protect memory space 🡪 multiple processes can occupy same physical location (assumes start 0)
  + Solutions:
    - Address relocation 🡪 if process starts at addr 8000, add 8000 to every reference to a memory address
      * Will have increase start up time
    - Base + limit registers 🡪 each process has its own base register which is use as an offset during compile time.
      * Limit is the max posb addr it can have (check if Base + Adr < Limit)
      * Each mem access needs an addition + comparison
* **Contiguous memory** 🡪 processes must be in one piece in memory during the whole execution
  + Assumes: each process occupies contiguous mem region, physical mem is large enough to hold one process
  + Multitasking 🡪 allows multiple processes in physical mem at same time
    - When mem is full 🡪 free mem by terminating processes or swapping blocked processes to sec storage
  + Memory partitioning:
    - Fixed-size 🡪 physical mem split into fixed number of equal size, a process occupy one of them
      * Pros: easy to manage and fast to allocate (no need choose, every partition is the same)
      * Cons: partition size need to be large enough to contain the largest process 🡪 internal fragmentation (leftover space if the partition does not occupy the whole partition); Fast but slow
      * Overhead calculation 🡪 X entries \* M bytes per entry (min = max overhead)
    - Variable-size 🡪 partition size based on actual size of process, OS keeps track of free and occupied
      * OS also splits and merges when needed
      * Pros: flexible and removes internal fragmentation
      * Cons: need to maintain more info in OS, takes more time to locate suitable region, process creation/termination/swapping tend to have large number of holes 🡪 external fragmentation
      * Allocation algos: First Fit, Best Fit (smallest hole that is large enough), Worst Fit (largest hole)
        + If the hole isn’t exactly the size of process 🡪 will produce a new hole
        + Best fit best in terms of memory efficiency (no need loop thru the whole list)
        + First fit best in terms of runtime (most of the time)
      * Merging 🡪 when an occupied partition is freed, merge with adjacent hole if possible
      * Free lists 🡪 keep lists of free holes of different hole sizes
        + Partition sizes increases exponentially (in power of 2) 🡪 just take hole from list that closely matches request size (faster allocation)
        + Overhead calculation 🡪 calc the cost for a unit (start address + partition size + status + node ptr)

Min is just the cost per unit (assume whole partition is free)

Max is number of partitions \* cost per unit

* + - * Buddy system 🡪 implementation of free list
        + upon request, free block is split into half repeatedly to meet request size (two halves are siblings)
        + When buddy blocks are both free 🡪 merge
        + Algo:

Graphical user interface, text, application

Description automatically generated Graphical user interface, text, application

Description automatically generated

* + - * + How to find buddies 🡪 two blocks B and C are buddy of size 2^S, if

The lowest S bits (bits 0 .. S-1) of B and C are identical (leftmost bits are same)

Bit S (i.e., S+1st bit) of B and C is different

Diagram

Description automatically generated

* + - * Bitmap 🡪 alternative system to use an array of bits to store partition info (1 if used, 0 ow)
        + check if there are consecutive 0s = to size of process
        + Use bitwise OR for allocation 🡪 OR with 1 at the bits to allocate, 0 everywhere else
        + Use bitwise AND for dellocation 🡪 1 at bits to keep, 0 everywhere else (incl bits to dealloc)
        + Overhead calculation 🡪 each bit in map represents smallest allocatable unit (max = min overhead)

Just take size of memory/size of smallest allocatable unit

**Lecture 9: Disjoint Memory Schemes**

* **Paging** 🡪 now memory space can be split into disjoint physical ranges
  + Physical memory is split into fixed sized regions 🡪 physical frames (logical memory also split)
    - Process can be loaded to any avail memory frames (physical is disjoint, logically contiguous)
    - Frame size (Page size) should be power of 2 and the physical and logical frame size should be the same
  + Table

    Description automatically generatedNeed page table 🡪 to map the physical to logical pages
    - Physical Address = Frame\_number x sizeof(physical\_frame) + offset from beginning of physical frame
      * Offset needed to find specific address within frame
  + Given: page size of 2n (in bytes), m bits logical address 🡪 page size is the offset
    - The most significant (m-n) bits of LA is used to find frame number f, offset is remaining n bits of LA
    - PA = f\*2n + o (f is physical frame number in page table)
  + Properties
    - No external fragmentation 🡪 every single free frame used
    - Small internal fragmentation 🡪 at max only 1 page per process not fully utilized (last page)
    - Separation of logical and physical space 🡪 allows flexibility and simple translation
  + Implementation
    - Pure software 🡪 OS stores page table info in PCB (context switch needed, everything else static not needed)
      * Issues 🡪 need 2 mem access for every memory reference (read page table + actl memory item)
        + RMB: code also need to be fetched as its stored in memory (2 mem access for each LoC also)
    - TLB🡪 hardware optimization (multiple per CPU core) to cache page table entries
      * Saves on accessing memory on TLB-hit 🡪 just retrieve frame number then access physical memory
      * Still need 2 memory access on TLB-miss 🡪 access page table (update TLB) + get physical frame
      * Calculations 🡪 P(TLB hit) x latency(TLB hit) + P(TLB miss) x latency(TLB miss)
        + Latency for hit is (TLB\_accesstime + mem\_accesstime), latency for miss is (TLB\_at + 2\*mem\_at)
        + Number of page entries = total logical memory address / page size (in bytes)
      * When context switch 🡪 TLB is flushed (ensure correctness + security)
        + Thus new processes will have many TLB misses to fill the TLB
  + Protection 🡪 paging can be used to provide memory protection (for each page table entry)
    - Access rights bits (wrx bits) 🡪 ensures code is only rx, data only wr etc
    - Valid bit 🡪 OS-set to indicate whether the page is valid to access by the process
  + Page sharing 🡪 allow several processes to share the same physical memory frame
    - Shared code page (library code) 🡪 the page table of the other process will point to the same page
    - Copy-on-write 🡪 at the start, page table is identical, only change when it is written to
      * Permission bits also copy, except write bits as we don’t want to overwrite
* **Segmentation scheme** 🡪 split logical memory into text, data, heap and stack since they have different properties
  + Why segment 🡪 allow regions to grow/shrink with gaps in between, allows easy check if in-range access
  + Memory segments are mapped into contiguous physical partitions of the same size (disjoint between segments)
    - Will have external fragmentation but has more efficient bookkeeping
    - Each segment has a name (heap, stack etc) and a limit (indicates the exact range of the segment)
    - Memory references are specified by segment name + offset (e.g. “heap” + 234)
  + Logical address translation 🡪 each segment has a base addr and limit/size (stored in segment table in CPU)
    - Logical address <SegID, Offset>: SegID is used to look up <Base, Limit>
    - Physical Address PA = Base + Offset (Offset < Limit for valid access else will have seg fault)
* **Segmentation with paging** 🡪 each segment is multiple pages with page table instead of contiguous memory region
  + Segment can grow/shrink by allocating new page then adding to page table
  + Segment table now contains the page table base address and page limit (as well as permissions)
  + Now to access need <segment\_id, logical\_addr> 🡪 use 1st number for the segment table, then 2nd like in paging
  + No internal or external fragmentation

**Lecture 10: Virtual Memory Management**

* **Virtual memory** 🡪 when logical space > physical space, need virtual memory (but treated as logical memory)
  + Basic idea 🡪 split the logical address space into small chunks (an extension of paging scheme)
    - Store in physical mem for more commonly accessed, store in sec storage for less accessed
  + Extended paging scheme 🡪 need a resident bit to tell if it is in physical mem or sec storage
    - CPU can only access memory pages 🡪 page fault otherwise (OS need to bring the page into mem)
      * If page not in memory, page fault (hardware) raised then OS takes control (software enter kernel mode) 🡪 choose page to be replaced (global/local using replacement algo) 🡪 write page if dirty 🡪 locates and loads page into physical mem (set resident bit to 1) + update page table 🡪 return from trap
      * If process is already in TLB 🡪 means that it is already in memory (won’t page fault)
      * Fetching is CPU free, while being fetched, process is blocked (takes very long time relatively)
  + Problems 🡪 latency of sec storage is much much slower so we want to page fault as little as posb
    - Thrashing 🡪 if memory access results in page fault most of the time
  + Locality 🡪 Most time is spent on relatively small part of code + accesses only relatively small part of data
    - Temporal locality 🡪 memory address used now is likely to be used again, just need load once
    - Spatial locality 🡪 memory addresses close to the address that is used now is likely to be used soon
      * Page contains many consec locations that are likely to be accessed in near future
  + Demand paging 🡪 start with no pages in memory, only allocate when there is page fault
    - Pros: fast startup time for new process (only need the essentials), small memory footprint
    - Cons: process slow at start due to page faults, page faults might cascade into other processes (thrashing)
* **Page table structure** 🡪 how to handle large logical memory space efficiently
  + Direct paging 🡪 just having 1 bit for each virtual memory address for 64-bit VA is way to big (must be contiguous)
  + 2-level paging 🡪 page the page table (sub page tables can be stored in swap if not needed)
    - Split page table into smaller page tables 🡪 Each with a page table number
    - If the original page table has 2P entries:
      * With 2M smaller page tables, M bits is needed to uniquely identify one page table
      * Each smaller page table contains 2P-M entries or page size/PTE entries size 🡪(branching factor)
    - To keep track of the smaller page table 🡪 single page directory is needed
      * Page directory contains 2M indices to locate each of the smaller page table
    - Overhead calculation: add these up together
      * page directory 🡪 #entries \* PTE size; page tables 🡪 #used page tables \* #entries/table \* PTE size
    - Number of levels 🡪 (#VA bits - #offset bits (page table size) ) / (branching factor)
    - Advantages 🡪 enables page table to grow beyond size of frame (no need contiguous) + can have empty entries (no need allocate page table if not needed saves space)
    - Problems 🡪 now need 2 memory accesses (for directory + page table) just to get the frame number
      * TLB can be used as it bypasses the 2 memory accesses but TLB misses have higher latency
      * MMU caches used with TLB to cache page directory entries (not PTE)

Diagram

Description automatically generatedDiagram

Description automatically generated

* + - * + Only need to store the root of the P4 table in the register
      * Invalid entries are pruned + not allocated 🡪 saves a lot of space but need more traversal
    - Calculation: take note of the malloc size, number of pages, frame size
      * Number of entries in last level 🡪 calc how many pages used: e.g. 1GB / 4KB pages = 2^30/2^12 = 2^18
        + Then we need the branching factor: page size/PTE entries size e.g. 4KB / 8 Bytes = 2^9
        + Next, we calc how many frames used: number of pages used/branching factor = 2^18/2^9 = 2^9
        + Lastly, check if it is page aligned 🡪 read the starting address and see if bits 12 to 21 are all 0: if not need add 1 to number of frames used for all the above levels
      * Number of entries in second last level 🡪 ceiling(# last level pages / branching factor)
      * In third level, it shld be either 1 entry or need +1:
        + How to know if need +1? 🡪 one entry might be able to hold 1GB and we only want to allocate 0.5GB, so normally can fit in one entry
        + But when we add 0.5GB to 0x1FFFFFFFF0000, it will be 0x200000….. which means it cross the boundary 🡪 think of it as any address to this level can at max fill up the last 4 bits, then ath else it will spill over to a new entry (cross the boundary)
  + Inverted page tables 🡪 keep a physical mapping of physical frame to <pid, page#> (which process using frames)
    - Normally, there will be M page tables (1 per process), but only N valid entries 🡪 lots of overhead
    - pid = process id , page# = logical page number in the corresponding the process
    - page# is not unique among processes, pid + page# can uniquely identify a memory page
    - entries are ordered by frame number 🡪 to lookup, need search whole table
      * Pros: one table for all the processes 🡪 space saving
      * Cons: slow translation
    - In practice, inverted tables often used to check who are all the sharers of the physical frame X
    - Diagram

      Description automatically generatedSize of page tables 🡪 Total frames = size of physical memory/page size
      * Calculation: Page Table Size = total frames \* (PTE size)
    - To translate logical address to physical 🡪 e.g. given 16 bytes frame size and 0110 1011 logical address, split into offset and page number
      * Offset = 4 (2^4 = 16) = 1011; page number = 0110
      * Then look at IPT and find the one with the correct page number (and PID) 🡪 get i (e.g. 2 in this case)
      * Physical address will be 0010 1011
* **Page replacement algos** (local replacement) 🡪 which page to replace due to limited number of resident mem pages
  + Needed when there is no free physical memory frame during page fault 🡪 need evict a mem page
    - Evicted page can be clean or dirty (need write back) 🡪 need swap in and out to sec storage (2 I/O)
    - Need know which process the page belongs 🡪 either use inverted or scan through all processes’ page table
      * Need change the resident bit
  + Calculation: Time\_access = (1-p) \* Time\_memory + p \* Time\_pagefault
    - Since T\_pagefault >> Time\_mem 🡪 need reduce p to keep T\_access reasonable
  + Algos:
    - Optimal page replacement (theoretical) 🡪 replace page that will not be needed for longest period of time
    - FIFO 🡪 evict the oldest memory page
      * Need maintain queue of resident page number 🡪 simple to implement without hardware support
      * FIFO does not exploit temporal locality for large number frames (bad performance in practice)
    - LRU 🡪 evict the page that has not been used in the longest time
      * Exploits temporal locality as we expect page to be reused in short time window by mirroring the past
      * Hard to implement, either use a counter (stores time) or stack (3223)
    - Clock replacement 🡪 keep reference bit, set at 1 if accessed, 0 if not accessed
      * Upon first pass, if bit = 1, set to 0 🡪 evict upon first encounter of 0
* **Frame allocation policies** 🡪 how to distribute limited physical memory frames among processes
  + Only needed if the number of frames < number of processes (if not won’t have problems)
  + Simple approaches 🡪 equal allocation and proportional allocation
    - Proportional is not good as most of the time might not need all the frames (might be working on small set)
  + Local replacement 🡪 Victim page are selected among pages of the process that causes page fault
    - Pros: number of frames allocated to process is constant 🡪 performance is stable between runs
    - Cons: if frames allocated not enough 🡪 hinder progress of a process
  + Global replacement 🡪 Process P can take a frame from Process Q by evicting Q's frame during replacement
    - Pros: allows processes to get more frames from others that need less
    - Cons: badly behaved processes can affect others 🡪 frame allocation depends on all processes (less isolation)
  + Thrashing 🡪 when there are insufficient physical frames (need to have enough but not too much that its wasted)
    - In global replacement, thrashing process steals pages from other processes 🡪 cause them to thrash (cascade)
    - In local replacement, thrashing limited to one process 🡪 but can take up I/O bandwidth and affect others
  + Working set 🡪 The set of pages referenced by a process is relatively constant in a period of time
    - E.g. during a function, references are likely on local vars, params, code
    - When while in working set 🡪 page faults are rare
    - When transitioning to new working set 🡪 tend to have a lot of page faults
    - Need set a fixed time interval and find the active pages in the time interval 🡪 then allocate enough pages
      * If interval too small 🡪 may miss pages in current WS; too large 🡪 may contain pages outside WS

**Lecture 11: File System Introduction**

* **File system** 🡪 abstraction, high level resource management, protection and sharing between processes and users
  + Must be self-contained (plug and play), persistent and efficient
  + Uses disks (HDD) with variable disk I/O, addressed by disk sector, contains non-volatile, explicit data (need open)
* **File** 🡪 Contains data (structured info) and metadata (additional info)
  + logical unit of info created by process (is smallest unit of storage to store stuff permanently)
  + Metadata: name, identifier, type, size, protection, time-date-owner info, table of content
    - File name: human readable reference to the file
      * length of file name, case sensitivity, special symbols, extension
    - File type: to determine operations available
      * Types: regular files (ASCII, binary files), directories, special files (character/block)
        + ASCII 🡪 displayed as is (e.g. text files, code)
        + Binary files 🡪 need be processed by specific program (executable, java file, mp4, pdf)
      * Unix has magic number (at beginning of file), windows has file extension (part of file name)
    - Operations: rename, change attributes, read attributes
    - File protection: controls access (rwx, append, delete, list)
      * 9 permission bits: (owner/group/universe) 🡪 e.g. chmod 777 means 111/111/111 = rwxrwxrwx
      * Access control list: permission for specific users, groups, has permission “upperbound”
  + File data
    - Structure: array of bytes, fixed length (use offset), variable length records (flexible but harder to locate)
    - Access methods:
      * sequential 🡪 cannot skip, read in order (used in magnetic tapes)
      * random access 🡪 read(offset) to read, seek(offset) moves file pointer
      * direct access 🡪 for fixed length records
    - Operations: create, open, read, write, reposition (seek), truncate (remove data between pos to end of file)
      * Usually done via syscalls 🡪 provides protection, concurrent and efficient access
      * Need open() before any further operations 🡪 gives the fd which is used in syscalls
        + open() enters the file into OS table of currently open files and places the file pointer at the beginning of the file
        + Close() removes the file from the table of open files.
      * Each process has its own per-process open-file table (keep track of open files) 🡪 system-wide open-file table (keep track of all open files in sys) 🡪 system-wide v-node table (link file with physical drive)

Diagram, engineering drawing

Description automatically generated Diagram

Description automatically generated

* + - * + File open same file 🡪 same offset (parent/child) changing one will affect other OR different offset
* **Directory** 🡪 for logical grouping of files (user) and keeping track of files (system)
  + Ways to structure directory
    - Single level (for cd drives etc)
    - Tree structured 🡪 dir can be recursively placed in other dir
      * Absolute pathname: Directory names followed from root of tree + final file
        + i.e. the Path from root directory to the file
      * Relative Pathname: Directory names followed from the current working directory (CWD)
        + CWD can be set explicitly or implicitly changed by moving into a new directory under shell prompt
    - DAG 🡪 file can “appear” to be in multiple directories (with diff pathname) but refer to same file
      * Hard link (only for files) 🡪 A and B has separate pointers point to the actual file F in disk
        + Pros: low overhead (only ptrs); changes made to hard link will affect (e.g. change permission)
        + Cons: deletion problem (if one deletes while other still using) 🡪 use reference count to fix
    - General graph: can have cycles (hard to traverse), hard to determine when to remove file/dir
      * Symbolic link 🡪 special link file G containing path name of F, when G accessed, find F then access F
        + Pros: simple deletion: if delete G, F remains; if delete F, G remains (but unusable)
        + Cons: overhead: need creation of link file
  + Also has permission bits 🡪 independent from file permission
    - read (can read this list): affects ls, auto-completion
    - write (can you change this list): affects create, rename, delete file/subdir
    - exec (can you use this directory as working dir): affects cd 🡪 allows access to file without read access in dir

Diagram

Description automatically generated

**Lecture 12: File System Implementation**

* **Disk organization:**
  + MBR at sector 0, followed by one or more partition (each can contain independent file system)
    - Each partition contains OS boot block, partition details, directory structure, files info and data
* **Implementing File** 🡪 File info and data in a partition
  + Files 🡪 collection of logical blocks (same size)
    - If file size != multiple of logical blocks 🡪 internal fragmentation in last block (not rly big issue)
  + Contiguous file block allocation 🡪 maintain starting address and length for each file
    - Pros: simple to keep track; fastest access (seq access just read, random access just use offset)
    - Cons: external fragmentation (worst disk utilization); file size need specified in advanced
  + Linked list 🡪 file info stores start and end block numbers; each disk block stores data + next block ptr
    - Pros: no ext fragmentation (best disk capacity utilization)
    - Cons: random access slowest (need traverse LL); overhead from storing ptr; less reliable (if ptr fails)
  + Linked list v2 🡪 use FAT table in memory that stores all block pointers in single table; just keep track of start
    - One entry in FAT for one disk block; Use a negative value to indicate last block
    - Pros: fast random access (can traverse the LL in memory then seek disk once)
    - Cons: FAT table can be huge (has to keep track of all disk blocks in partition); memory overhead
  + Indexed allocation 🡪 uses disk blocks to store index blocks for each file, might need indirect blocks for large files
    - Index block is array of disk block addresses 🡪 IndexBlock[ N ] == Nth Block address
    - Pros: lesser mem overhead (just need open index block of opened file); fast random access
    - Cons: limited max file size (max = index block entries); index block takes up logical block (overhead)
  + Indexed allocation variations
    - Linked scheme 🡪Keep a linked list of index nodes
      * Cons: expensive since traversal cost
    - Multilevel index: 🡪 Similar idea as multi-level paging (slowest seq access)
      * Can be generalized to any number of levels
    - Combined scheme: 🡪Combination of direct indexing and multi-level index scheme
      * Fast access for small files, but can still handle large files
* **Free space management** 🡪 part of partition details, need to know which disk block is free
  + Allocate: Remove free disk block from free space list 🡪 needed when file is created or enlarged (appended)
  + Free: Add free disk block to free space list 🡪 needed when file is deleted or truncated
  + Bitmap 🡪 each disk block 1 bit (1 means free, 0 means occupied); free space = number of 1s \* block capacity
    - Pros: can use bitwise manipulation easily
    - Cons: need keep in memory (overhead)
  + Linked list of disk blocks 🡪 each disk block contains a number of free disk block numbers OR ptr to next free
    - Pros: Easy to locate free block 🡪 Only first pointer is needed in mem (other blocks can cached for efficiency
    - Cons: High overhead (uses disk blocks to store meta info)
* **Implementing directory** 🡪 keep track of files in directory + map file name to file info
  + File need to be open with syscall to locate file using pathname + file name before use
    - To get fd 🡪 given path name need recursively search dir along path until reach file
  + Linear list 🡪 each dir has a list where each entry in list stores file name + file info (or ptr to it)
    - To search: need linear search 🡪 can cache last few searches to improve perf
  + Hash table 🡪 each dir contains hash table
    - To search: hash file name + lookup (chained collision resolution used 🡪 at worst need check whole chain)
    - Pros: fast lookup (hash)
    - Cons: limited size hash table; need good hash func
* **MS FAT system** 
  + File data are allocated to:
    - A number of data blocks / data block clusters (cluster = better seq access but more ext frag)
    - Allocation info is kept as a linked list 🡪 all data block pointers kept separately in the FAT
      * Only partition detail is part of the FAT 🡪 dir struct, files info and data are all in data blocks
  + File Allocation Table (FAT) 🡪 FAT 16 means 2^16 data blocks = 2^16 entries in FAT
    - One entry per data block/cluster; store disk block information (Free? Next block (if occupied)? Damaged?)
    - OS will cache in RAM to facilitate linked list traversal
    - Each entry can address 16bits of disk block 🡪 size of each entry = 16 bit
      * Contains either: FREE (unused), Block Number (of next block), EOF (null ptr), BAD (error)
  + Directory 🡪 represented as special type of file
    - Root directory stored in special location; other dir stored in data blocks
    - Directory entry 🡪 fixed 32 byte per entry (each file/subdir is an entry)
      * Name (8 bytes) + ext (3 bytes) 🡪 no need the dot “.” , it is automatic
      * Attributes (1 byte)
      * creation date + time (2 bytes each): limited to epoch date
      * first disk block (depends on FAT ver, FAT16 will be 2 bytes) + file size (4 bytes)
  + To access: need look at FAT table to see if the file/directory occupies more than one disk block
    - If it is EOF, then it only occupies one, if not then need follow the chain in the FAT
* **Ext2** 🡪 each partition contains block groups which
  + Diagram

    Description automatically generatedDiagram

    Description automatically generated
    - I-nodes (128 bytes each) 🡪 1 entry for every file & directory
    - Superblock 🡪 describes whole file system
      * Total I-Nodes number, I-Nodes per group, Total disk blocks, Disk Blocks per group (for redundancy)
    - Group descriptors 🡪 describes each of the block group
      * Number of free disk blocks, free I-nodes, Location of the bitmaps (for redundancy)
    - Bitmaps 🡪 1 = occupied, 0 = free
    - I-node handles file info only, bitmap stores the partition details and data blocks stores file data + dir struct
  + I node (stored in disk) 🡪 first few bytes are metadata
    - Data block pointers:
      * first 12 pointers points to disk blocks with actual data
      * 13th ptr points to disk block that stores direct ptrs (single indirect block) and 14th/15th is double/triple
    - Pros: fast access to small file; able to handle large files
    - Calculation (1KB disk block, 4 bytes block address):
      * 12 \* 1KB = 12KB direct blocks
      * Number of entries in disk block = 1KB (210)/4 = 256 entries
        + Storage capacity for Nth indirect block = 256N total entries \* 1KB
  + Dir structure 🡪 within the i-node there is a disk block number for the directory entry
    - LL of directory entries (variable size) containing: i-node number for this + size of this entry (for offset calc to next entry) + length of the file/subdir name (includes dot) + file/subdir name + type (file/subdir)
  + How to access file 🡪 need jump between I node/ disk block
    - Let CurDir = "/" 🡪 Root directory usually has a fixed I-Node number (e.g. 2)
      * Locate + access directory + read the actual I-Node (2 disk accesses)
    - Look at the next part in pathname:
      * If it is a directory, e.g. "sub/" 🡪 Locate + access the directory CurDir
        + Retrieve I-Node number, then read the actual I-Node
        + CurDir = next part in pathname
        + Goto Step 2.
      * Else it is a file 🡪 Locate the directory entry in CurDir
        + Retrieve I-Node number, then read the actual I-Node
  + Diff from v-node 🡪 v-node is stored in RAM, contains all the i-node info of file which are opened
  + Hard link 🡪 same i-node number (so will access same data); only creates one entry (low overhead)
  + Symbolic link 🡪 creates a special file + entry with different i-node where the path to original is stored

A picture containing text, person

Description automatically generated