Kernel

Wednesday, January 11, 2023 5:17 PM

Minimum interface required to run programs on a machine.

Key Ideas:

- Abstraction: What is the desired illusion
- Mechanism: How to create the illusion, fixed method
- Policy: Which way to use mechanism to meet a goal

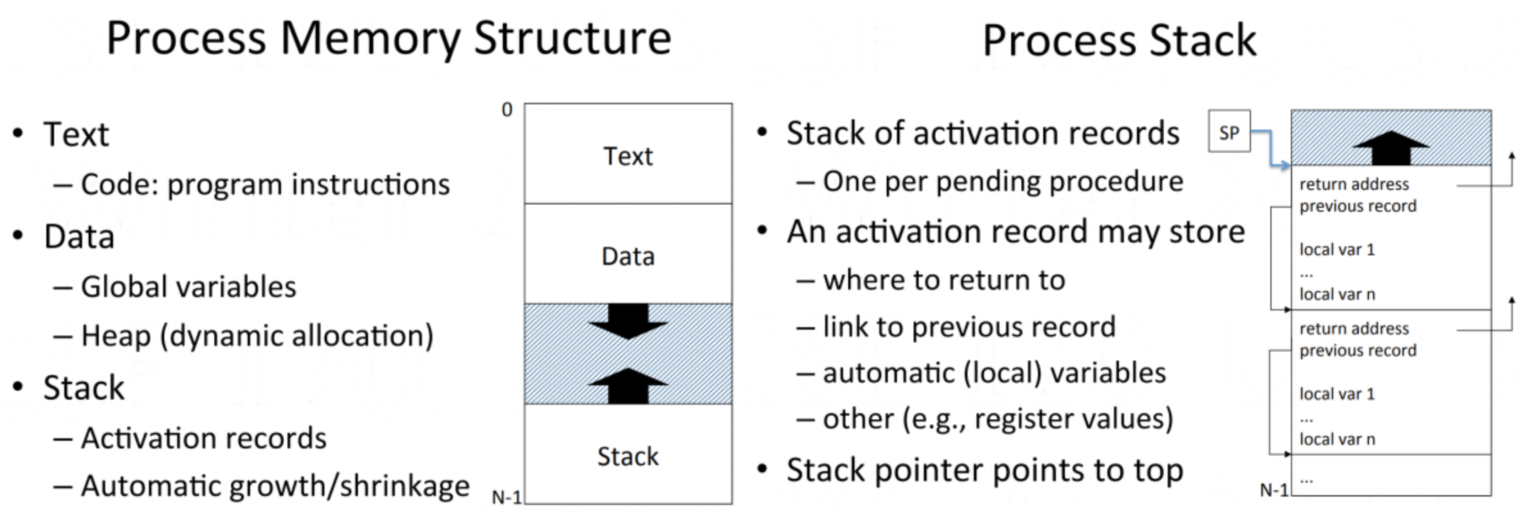
Process

Monday, January 9, 2023 2:03 PM

Def: Abstraction of a running program. Is dynamic and has state which changes.

Resources of a process:
- CPU: Executes instructions
- Memory: Stores state

Context of a process:
- CPU context: values of registers (PC, SP, FP, GP)
- Memory context: pointers to memory areas (Text, Data, Heap, Stack)
- Kernel State



Goal: Support multiple processes simultaneously

Multiprogramming, Context Switching

Wednesday, January 11, 2023 5:19 PM

Multiprogramming:

- Given a running process
  - At some point, it needs a resource, e.g., I/O device
  - Say resource is busy, process can't proceed
  - So, "voluntarily" gives up CPU to another process
- Yield (p)
  - Let process p run (voluntarily give up CPU to p)
  - Requires context switching

Context Switching:

- Allocating CPU from one process to another
  - First, save context of currently running process
  - Next, restore (load) context of next process to run
- Loading the context
  - Load general registers, stack pointer, etc.
  - Load program counter (must be last instruction!)

Simple context switching example:

- Two processes: A and B
- A calls Yield(B) to voluntarily give up CPU to B
- Save and restore registers
  - General-purpose, stack pointer, program counter
- Switch text and data
- Switch stacks
  - Note that PC is in the middle of Yield!

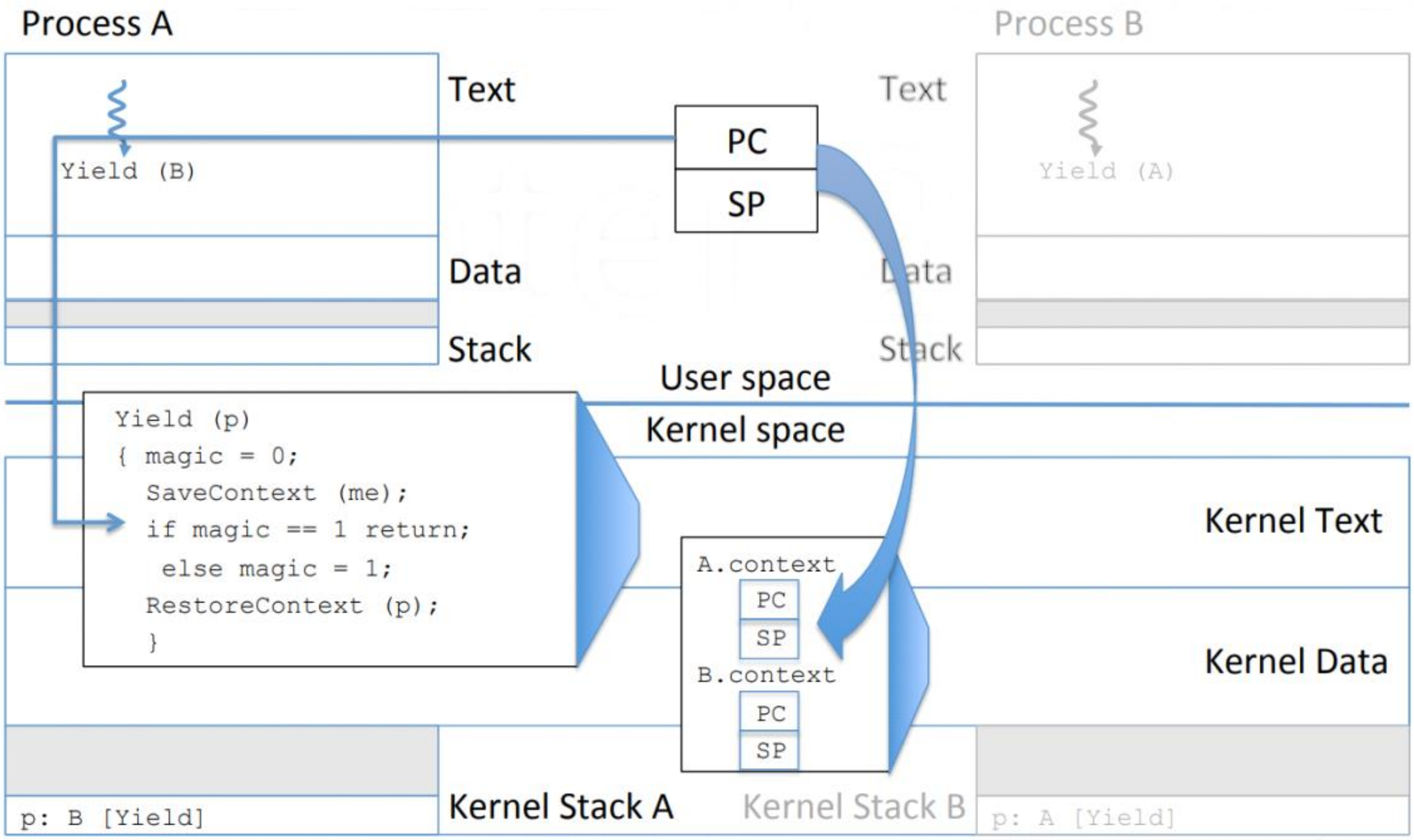
Notes:

- User context switch: syscall -> Yield() -> TRAP instruction
  - Trap instruction: indicates to CPU that it will suddenly switch to kernel space
- Kernel context switch: clock interrupt -> preemptive scheduling

Yielding via Kernel

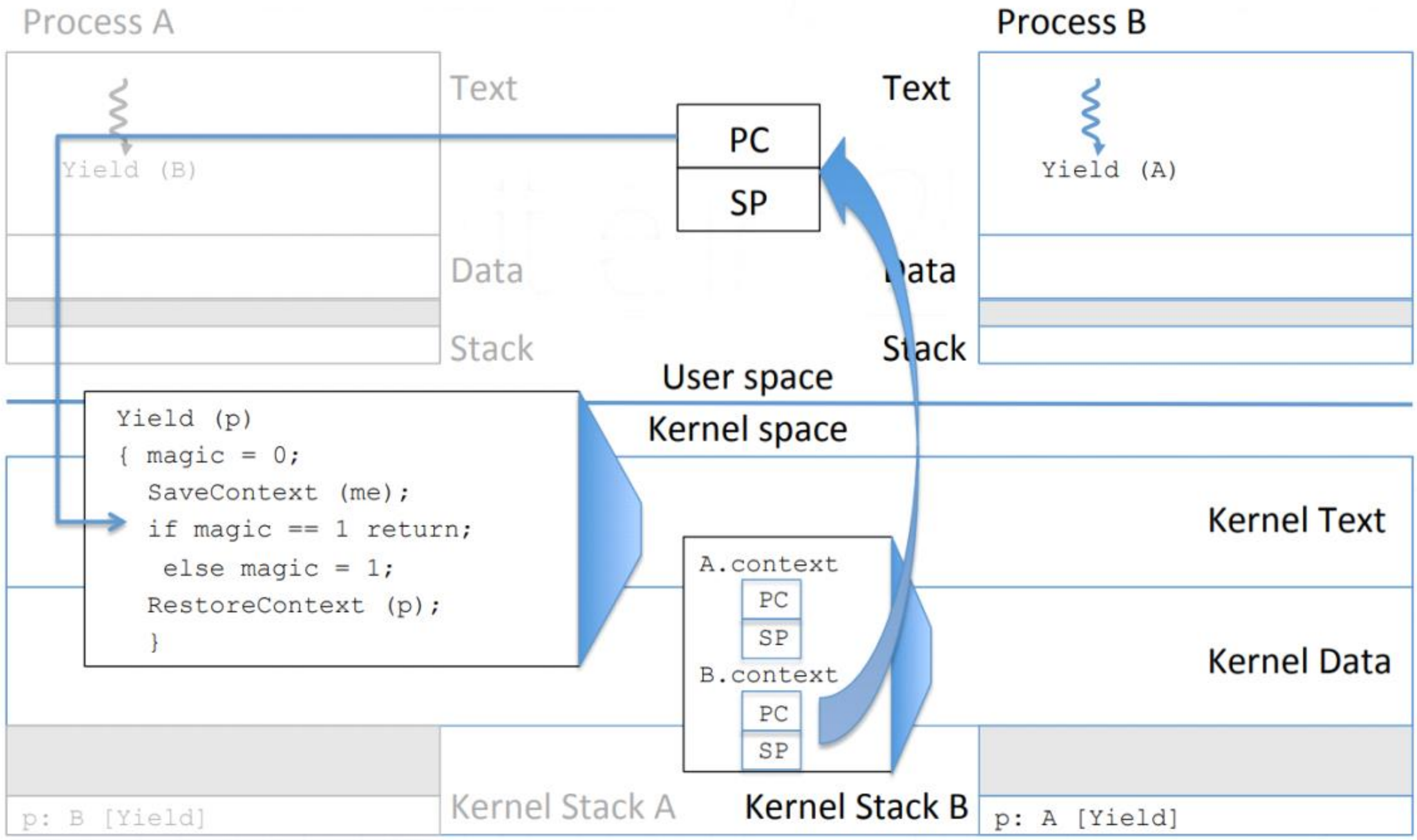
Wednesday, January 11, 2023 6:11 PM

Saving process A to Kernel Space



Restoring process B from Kernel Space

After Restoring Context of B



Timesharing, Process State Diagram

Wednesday, January 18, 2023   5:05 PM

Def: Divide CPU time into parts and allocating to processes

Idea: Create the illusion of parallel progress by rapidly switching CPU

Note: Kernel must keep track of each process' progress

Kernel Maintains List of Processes

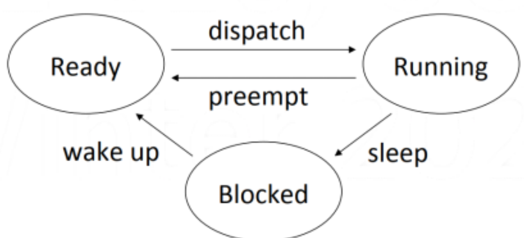
Process ID	State	Other info
1534	Ready	Saved context, …
34	Running	Memory areas used, …
487	Ready	Saved context, …
9	Blocked	Condition to unblock, …

- All processes: unique names (IDs) and states
- Other info kernel needs for managing system
  - contents of CPU contexts
  - areas of memory being used
  - reasons for being blocked

- Running: making progress, using CPU
- Ready: able to make process, not using CPU
- Blocked: not able to make progress, can't use CPU - blocked by some resource (ie. IO)

Kernel selects a ready process and lets it run
Eventually the kernel regains control, and then selects a new process

Process State Diagram



- State transitions
  - Dispatch: allocate the CPU to a process
  - Preempt: take away CPU from process
  - Sleep: process gives up CPU to wait for event
  - Wakeup: event occurred, make process ready

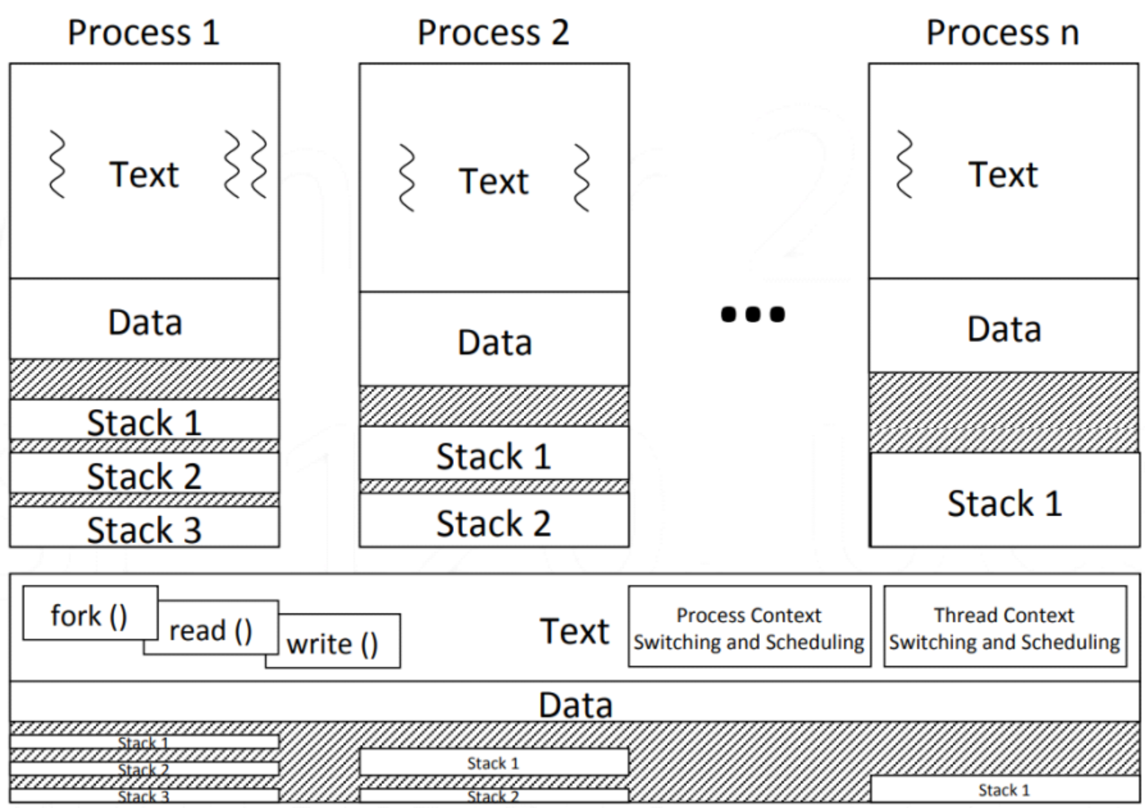
Threads

Wednesday, January 18, 2023 5:59 PM

Def: Single sequential path of execution, independent of memory

- Threads are part of a process
  - Lives in the memory of a process
  - Allows multiple threads per process
  - Threads share text and heap, but have their own stack
- Advantage to Users: unit of parallelism
- Advantage to Kernel: unit of schedulable execution

Implementation: Call ForkThread()
- Management:
  - Thread context switching
  - Thread scheduling



Idea: we can allow users to manage threads, include a thread library
- Thread calls at the user level: ForkThread(), YieldThread() …
- Supports threads on any platform, but no true parallelism

- User-level threads
  - Portability: works on any kernel
  - Efficient: thread-switching occurs in user space
  - User can decide on scheduling policy
  - But no true parallelism (without special support)
- Kernel-level threads
  - Can achieve true parallelism
  - Overhead: thread switch requires kernel call

- Hardware-level Thread
  - Actual hardware support
  - Logical CPU

Scheduling Policies

Monday, January 23, 2023 6:44 PM

Problem: Which processes get CPU and when

Def:
- Arrival time: time that process is created
- Service time: CPU time needed to complete
- Turnaround time: difference from arrival to departure

- Preemptive: kernel takes away CPU from a process through interrupts
- Starvation: process may never get CPU

Longest First: select process with the longest service time

Shortest First: select process with the shortest service time
- Provably optimal

Note: Longest/Shortest first MUST know the service time of the processes, which is not easily doable

FIFO / First Come First Serve: select processes in order of arrival
- Non-preemptive, simple, no starvation

Round Robin: Each process gets CPU in turn
- Preemptive, simple, no starvation

Shortest Process Next: select process with shortest service time
- Non-preemptive, assumes known service time, allows starvation

Multi-Level Feedback Queues:
- Priority queues: 0 (high), …, N (low)
- New processes enter queue 0
- Select from highest priority queue
- Run for T = 2^k quantums
    - Used T: move to next lower queue, FIFO –
    - Used < T: back to same queue, RR
        - Due to yield or higher priority arrival
- Periodically boost (e.g., all to highest queue)
- Preemptive, complex, possible starvation

Priority: Pick the process with highest priority
- Allows scheduling based on external criteria
- Might have starvation

Fair Share: Give CPU utilization equal to requested amount over the long run
- Each process requests some percentage CPU utilization

	1	2	3	4	5	6	7	8	9	10
A	100%	50%	33%	50%	40%	50%	43%	50%	44%	50%
B	0%	50%	33%	25%	20%	17%	14%	13%	11%	10%
C	0%	0%	33%	25%	40%	33%	43%	38%	44%	40%

- Each process requests some CPU utilization
    - Utilization: what percentage of time resource is used
    - Example of requests: A: 50%; B: 10%; C: 40%
- Select process with least action/requested ratio
- Too much overhead, requires a division for each process

Stride Scheduling:

For processes A, B, C … with requests R_A, R_B, R_C …

Calculate ***strides***: $S_A = 1/R_A$, $S_B = 1/R_B$, $S_C = 1/R_C$ …

For each process *x*, maintain ***pass*** value P_x (init 0)

Schedule: repeat every quantum

- Select process *x* with minimum pass value P_x, run
- Increment pass value by stride value: $P_x = P_x + S_x$

Optimization: use only integers for R_x, S_x and P_x

- Calculate $S_x = L/R_x$ using very large L, e.g., L = 100000

Real Time Scheduling

Monday, January 30, 2023 6:28 PM

Problem: correctness of real time systems depend on correctness of computation and timing of results
- If a result is delivered after a deadline, it is considered incorrect

Hard: Every deadline must be met otherwise something catastrophic happens, ex: nuclear power plant
- Every deadline MUST be met

Soft: Missed deadlines are ok, ex: video delivery

Periodic: Does something, then waits for the next period of time

Period (T): each periodic cycle, each process must complete before this period

CPU burst (C): every period, each process must get some amount of CPU time

Utilization = C/T

Earliest Deadline First: Schedule the process with the earliest deadline
- If earlier deadline occurs, preempt
- Works for periodic and aperiodic processes
- Achieves 100% utilization
- MUST SORT DEADLINES, can be slow
- If sum of utilization <= 100%, will meet all deadlines

Rate Monotonic Scheduling: Only for periodic processes, prioritize based on rates
- At start of period, select highest priority
- Preempt is necessary
- If all processes are finished, wait until the next deadline
- If sum(utilization) <= n*(2^(1/n) - 1) where n is the number of processes then it will pass
  - Not necessarily fail if this does not hold

Synchronization

Monday, January 30, 2023 7:33 PM

Avoid race conditions, where processes will modify the same resource at the same time

Ex: two processes modify variable money
P1: money += 100
P2: money -= 100
If both processes make a copy of money at the same time, then when they try to return, there will be ambiguity

- Identify critical sections of code
- Enforce mutual exclusion, only one process active in a critical section
  - Can achieve mutual exclusion by restricting only one process to be in its critical section at any time

Solution Requirements:
1. At most one critical section at a time
2. Can't prevent entry if no others are in theirs
3. Should eventually be able to enter
4. No assumptions about CPU speed or number

Software Lock:
- Use a "shared variable" (between processes)

```
shared int lock = OPEN;
```

P_0

```
while (lock == CLOSED);
lock = CLOSED;
< critical section >
lock = OPEN;
```

P_1

```
while (lock == CLOSED);
lock = CLOSED;
< critical section >
lock = OPEN;
```

Lock indicates if any process in critical section

- If an interrupt happens just after P1 enters the while loop, then it can enter the critical section upon resume which breaks #1, since both P1 and P0 are now in the critical section

Taking Turns:

```
shared int turn = 0;      // arbitrary set to P0
```

P_0

```
while (turn != 0);
< critical section >
turn = 1;
```

P_1

```
while (turn != 1);
< critical section >
turn = 0;
```

Alternate which process enters critical section

- If turn = 0, but P1 is running (ie context switch occurred), then P1 is prevented entry and may never enter which breaks #2 #3

State Intention

```
shared boolean intent[2] = {FALSE, FALSE};
```

P_0

```
intent[0] = TRUE;
while (intent[1]);
< critical section >
intent[0] = FALSE;
```

P_1

```
intent[1] = TRUE;
while (intent[0]);
< critical section >
intent[1] = FALSE;
```

Process states intent to enter critical section

- If P0 sets intent[0], then context switch to P1 and sets intent[1], then neither process can escape the while loop

Peterson's Solution

```
shared int turn;
shared boolean intent[2] = {FALSE, FALSE};
```

P0	P1
`intent[0] = TRUE;`	`intent[1] = TRUE;`
`turn = 1;`	`turn = 0;`
`while (intent[1] && turn==1);`	`while (intent[0] && turn==0);`
`< critical section >`	`< critical section >`
`intent[0] = FALSE;`	`intent[1] = FALSE;`

- If competition, take turns; otherwise, enter

  - Use both intention and turn variables

Disabling Interrupts

  - Need to disable interrupts for each CPU individually, might have an interrupt in the middle

TSL Instruction

Wednesday, February 1, 2023 7:11 PM

TSL mem (test-and-set lock: contents of mem)

```
do atomically (i.e., locking the memory bus)
      [ test if mem == 0 AND set mem = 1 ]
```

Operations occur without interruption

- Memory bus is locked
- Not affected by hardware interrupts

Solution: avoid interrupt issues using the TSL instruction

```
shared int lock = 0;
```

P_0	P_1
`while (! TSL(&lock));`	`while (! TSL(&lock));`
`< critical section >`	`< critical section >`
`lock = 0;`	`lock = 0;`

Shared variable solution using TSL(int *)

- tests if lock == 0 (if so, will return 1; else 0)
- before returning, sets lock to 1

Simple, works for any number of processes

Still "suffers" from busy waiting

Semaphores

Wednesday, February 1, 2023 7:20 PM

Def: Synchronization variable
- Integer values
- Can cause process to block/unblock when modifying
- Cannot test the value of semaphore

Operations:
- wait(s) - decrement; block if s < 0
- signal(s) - increment; if any blocked, unblock one

Notes:
- Only for synchronization use, cannot learn anything about another process because no information is transferred
- Still has some busy waiting within the semaphore implementation, but is relatively smaller than other solutions

Solution:

```
sem mutex = 1;          // declare and initialize
```

P_0	P_1
`wait (mutex);`	`wait (mutex);`
`< critical section >`	`< critical section >`
`signal (mutex);`	`signal (mutex);`

Use “mutex” semaphore, initialized to 1

Only one process can enter critical section

Simple, works for *n* processes

Implementation:

Semaphore s = [n, L]

– n: takes on integer values

– L: list of processes blocked on s

Operations

```
wait (sem s) {
    s.n = s.n – 1;
    if (s.n < 0) add calling process to S.L and block; }
signal (sem s) {
    s.n = s.n + 1;
    if (s.L !empty) remove/unblock a process from s.L; }
```

wait and signal MUST BE ATOMIC!!! Use a lower level mechanism to implement wait and signal.

Process Ordering

Wednesday, February 1, 2023 7:32 PM

Another use for semaphores:

Order How Processes Execute

```
sem cond = 0;
```

P_0	P_1
`< to be done before P1 >`	`wait (cond);`
`signal (cond);`	`< to be done after P0 >`

Cause a process to wait for another

Use semaphore indicating condition; initially 0

– the condition in this case: "P_0 has completed"

Used for ordering processes

– In contrast to mutual exclusion

Inter-Process Communication

Wednesday, February 8, 2023 6:45 PM

IPC requires mechanisms for:

- Data transfer
- Synchronization

Shared memory + semaphores

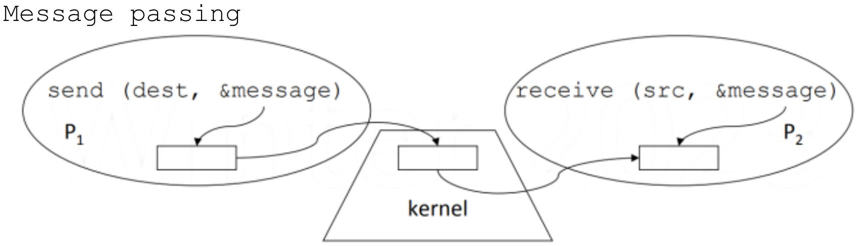
```
shared int buf[N], in = 0, out = 0;
sem filledslots = 0, emptyslots = N, pmutex=1, cmutex=1;
```

```
Producer1, 2, …                    Consumer1, 2, …
while (TRUE) {                     while (TRUE) {
    wait (emptyslots);                 wait (filledslots);
    wait (pmutex);                     wait (cmutex);
    buf[in] = Produce ();              Consume (buf[out]);
    in = (in + 1)%N;                   out = (out + 1)%N;
    signal (pmutex);                   signal (cmutex);
    signal (filledslots);              signal (emptyslots);
}                                  }
```

Monitors:

- Programming language construct for IPC
    - Variables (shared) requiring controlled access
    - Accessed via procedures (mutual exclusion)
    - Condition variables (general synchroniza6on)
        - wait (cond): block until another process signals cond
        - signal (cond): unblock a process waiting on cond
- Only one process can be active inside monitor
    - Active = running or able to run; others must wait

Message passing



- Two methods
    - send (destination, &message)
    - receive (source, &message)
- Data transfer: in to and out of kernel message buffers
- Synchronization: receive blocks to wait for message

Issues with Message Passing

Who should messages be addressed to?

– ports ("mailboxes") rather than processes

How to make process receive from anyone?

– pid = receive (*, &message)

Kernel buffering: outstanding messages

– messages sent that haven't been received yet

Good paradigm for IPC over networks

Safer than shared memory paradigms

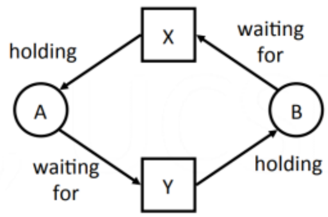
Deadlock

Monday, February 13, 2023 7:25 PM

Def: Deadlock

Set of processes are permanently blocked
- Unblocking of one relies on progress of another
- But none can make progress!

Example
- Processes A and B
- Resources X and Y
- A holding X, waiting for Y
- B holding Y, waiting for X
- Each is waiting for the other; will wait forever



Four conditions for deadlock:
1) Mutual exclusion: only one process can use a resource at a time
2) Hold and wait: process holds resource while waiting for another (ie hold memory and request more memory)
3) No preemption: can't take resource away from a process
4) Circular wait: the waiting processes form a cycle

Solutions:
- Prevention: make deadlock impossible by removing condition
  1) Mutual exclusion: Some resources may not be easily shared
  2) Hold and wait: Not all processes know the amount of resources beforehand
  3) No preemption: processes may be in the middle of using resources
  4) Circular wait:
- Avoidance: Avoid situations that lead to deadlock
  - Bankers Algorithm
    - Process claim matrix: how much of each resource a process will use at most
    - Process allocation matrix: how much of each resource a process is currently using
    - Resource availability vector: which resources are available
    - Keep system in a safe state, where there is an order of execution to escape any deadlock
- Detection and Recovery
  - Do nothing to prevent/avoid deadlocks
    - So something if/when they happen
  - Justification:
    - Deadlocks rarely happen
    - Cost of prevention or avoidance not worth it
  - Most popular approach
  - Detecting deadlocks:
    - Detect a cycle in resource requirements
  - Recovery:
    - Break the cycle
      - Terminate all deadlocked processes
      - Terminate processes one at a time
    - Potentially causes issues with resources in intermediate state (files half written)

Memory

Wednesday, February 22, 2023 7:05 PM

Where should process memories be placed?

- Memory management

How does the compiler model memory?

- Logical memory
- Segmentation

How to deal with limited physical memory?

- Virtual memory
- Paging

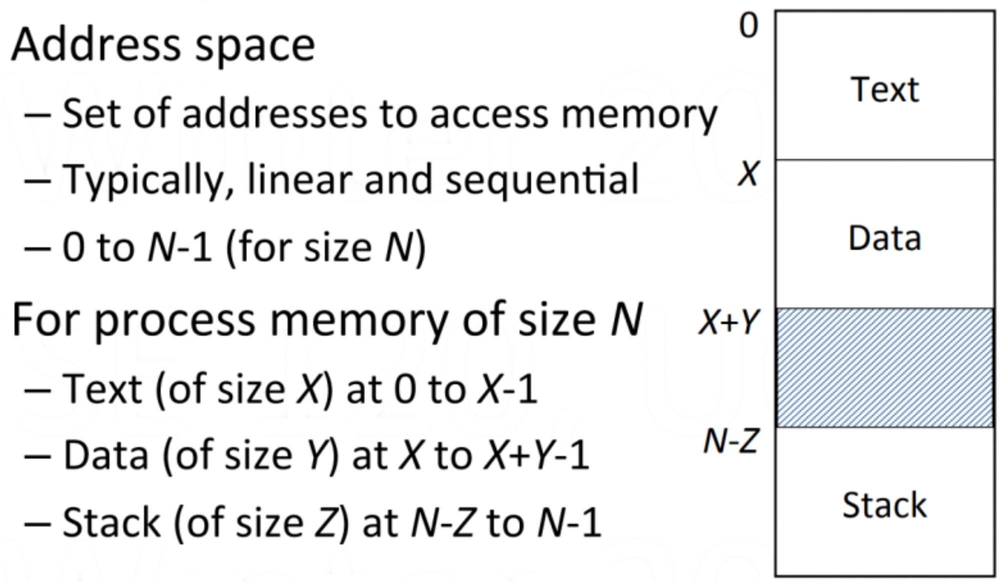
Process Memory

Wednesday, February 22, 2023 6:24 PM

Process Memory:
- Text: code of program
- Data: static variables, heap
- Stack: Automatic variables, activation records
- Shared memory regoins

Characteristics: size (fixed or variable), Permissions (r,w,x)

Address space:



Memory Management, Fragmentation

Wednesday, February 22, 2023 6:45 PM

Problem: how to allocate and free portions of memory
- Allocation occurs when: processes created or request more memory
- Free occurs when: process exits, process no longer requires memory requested

Solution:
- Physical memory starts as one empty "hole"
- Over time, areas get allocated: "blocks"
- To allocate memory - Find large enough hole
  - Allocate block within hole
  - Typically, leaves (smaller) hole
- When no longer needed, release
  - Creates a hole, coalesce with adjacent

Problem: How to select the best hole?

First fit: select the first hole that fits the block
- Simple and fast

Next fit: select the next available hole that fits the block
- Simpler and faster

Best fit: selects the smallest hole that fits the block
- Must check every hole (slow)
- Leaves very small fragments

Worst fit: selects the largest hole
- Must check every hole (slow)
- Leaves very large fragments

Problem: fragmentation, where lots of small holes are scattered everywhere
- Internal fragmentation: unused space <u>within</u> allocated block, cannot be allocated to others
- External fragmentation: unused space <u>outside</u> any blocks, can be allocated

Compaction: Reallocate processes so that a larger holes can be created
- Simple but very time consuming

Subblock: Break block into smaller sub blocks and fit into smaller holes, filling fragments
- Easier to fit and faster but complex

50% Rule, Unused Memory Rule

Monday, February 27, 2023 6:39 PM

Def: 50% rule
- Holes = 1/2 * Blocks

Note holes are always external fragmentation

Def: Unused Memory Rule:
- f = k / (k+2)
- k = h/b - average hole to block size
- As k -> infinity, then f -> 1
- As k -> 0, then f -> 0

Buddy System

Monday, February 27, 2023 6:56 PM

Problem: variable size allocations cause external fragmentation

Idea: have a few preselected sizes
- One size: inflexible, may be too small or large
- A good variety of sizes: flexible but more complex

Solution: Buddy system

Partition into power-of-2 size chunks

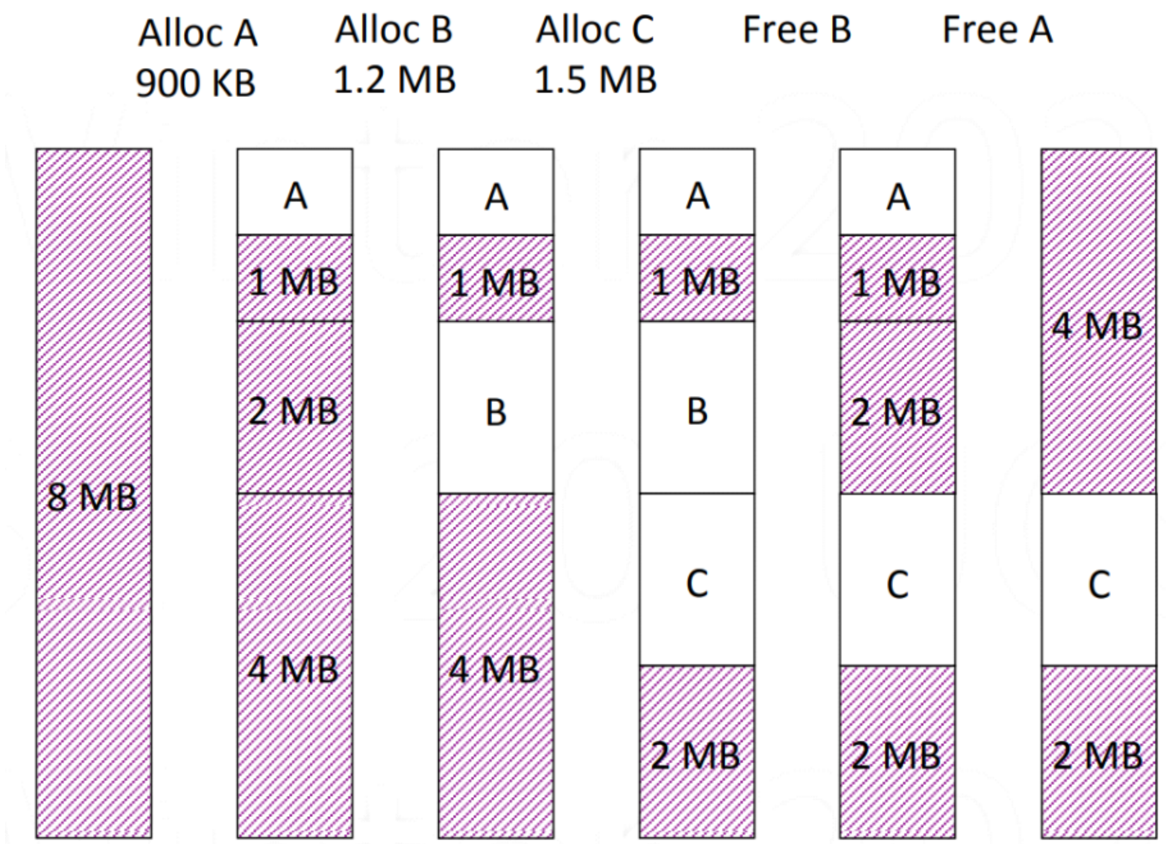
Alloc: given request for size r

```
find chunk of size ≥ r (else return failure)
while (r ≤ sizeof(chunk)/2)
  divide chunk into 2 buddies (each 1/2 size)
allocate the chunk
```

Free: free the chunk and coalesce with buddy

```
free the chunk
while (buddy is also free)
  coalesce
```

Ex:



Note: use a binary tree to store the allocations

Logical Memory

Wednesday, March 1, 2023 6:31 PM

Logical memory: a process' memory as referenced by a process
- Allocated without regard to physical memory

Problems with sharing memory:
- Addressing: unknown where process will be allocated
- Protection: prevent process from modifying another
- Space: how to distribute finite memory to many processes

Address Space: Set of addresses for memory, usually linear
- Typically kernel occupies the lowest address

Local addresses: assumes separate memory starting at 0
- Compiler generated
- Independent of location in physical memory

Converting logical to physical addresses:

Software: Compiler sets the offset at compile time

Hardware:
- Addressing: Base register filled with start address, added to logical address on access
- Protection: use a bound register to ensure process does not go out of bounds

Organizing Physical Address Space:
- Segmented: divide into segments of different sizes
  - Segment translate table: remembers the starting address of each segment
    - V: valid bit
    - Base: segment location
    - Bound: segment size
    - Perm: permissions
  - Add offset + base to find the physical address
  - Also hold entries for bounds and permission
  - **One segment table per process stored in kernel**
- Paged: Partition into pages of fixed size
  - Keep table mapping pages in logical to pages in physical, one per process
    - V: valid bit
    - Demand paging bits
    - Frame: page location
  - Convert top n bits of logical to top n bits of physical and keep the offset

Monday, March 6, 2023 6:39 PM

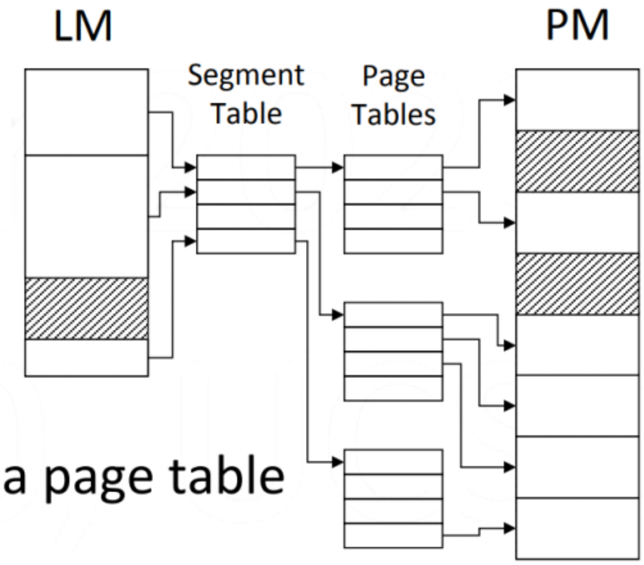
Combining Segmentation and Paging:

Logical memory
- composed of segments

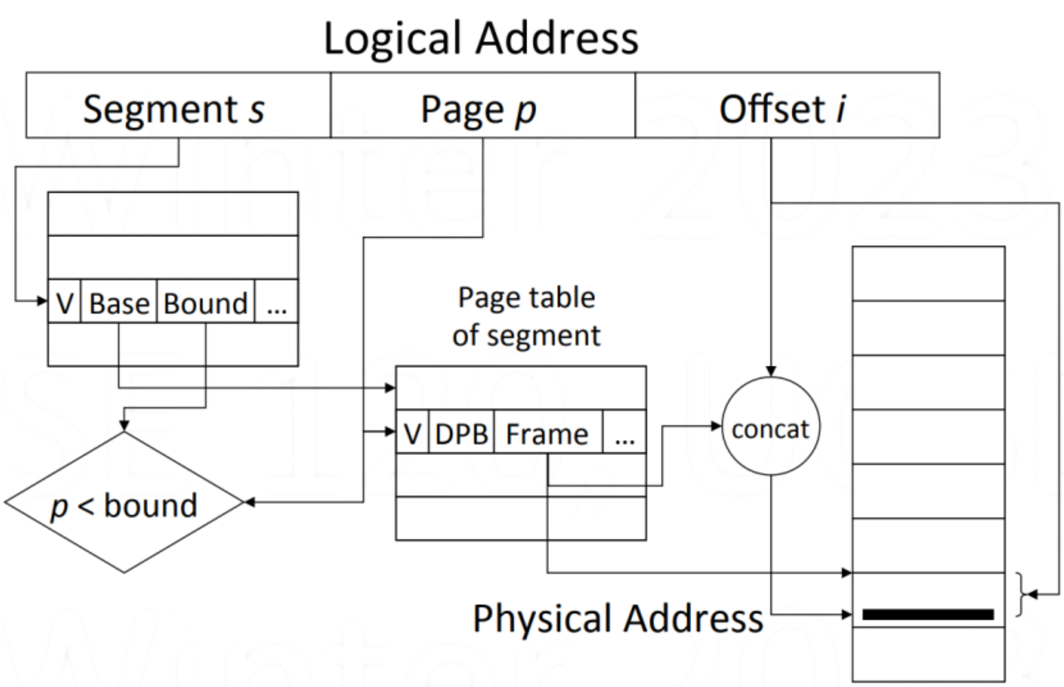
Each segment
- composed of pages

Segment table
- Maps each segment to a page table

Page tables
- Maps each page to physical page frames

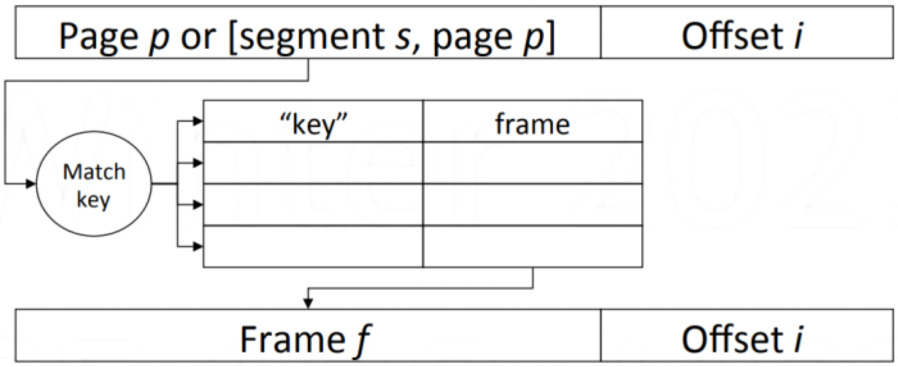


Address translation:
- Logical address: [segment s, page p, offset i]
- Do various checks: s < STSR, valid == 1, p < bound, permissions
- Use s to index segment table to get page table
- Use p to index page table to get frame f
- Physical address = concatenate (f, i)



```
Cost: Each lookup is a memory access
  - Keep commonly accessed pages in fast memory
  - Leverage space locality
```

TLB: Translation Look-aside Buffer



Fast memory keeps most recent translations

If key matches, get frame number

else wait for normal translation (in parallel)

Virtual Memory

Monday, March 6, 2023 7:14 PM

Def: Virtual memory is a logical memory except not all memory may be store in physical memory
- Keep most of process memory is kept in disk, which is larger and cheaper
- Unit of memory is segment or page

Idea: Treat physical memory as a cache of commonly used segments or pages
- If a page access is not in memory, throw a page fault

Page fault handling: TRAP into kernel
- Find page on disk (kept in kernel data structure)
- Read page into free frame (may need to replace)
- Record frame number into page entry table
- Set valid bit and other fields
- Retry instruction

Problem: Disk is slow, 5 - 6 orders of magnitude slower
- Ensure page faults are rare

Page Replacement: What page to replace with a new page?
- FIFO: replace the page that is the oldest
    - Simple: use frame ordering
    - Does not perform well, oldest page may be the most popular
- OPT: select page to be used furthest in the future
    - Optimal but requires future knowledge
    - Establishes best case
- LRU: select page that was least recently used
    - Predict future based on past
    - Costly, need to time stamp each page access, find least
- Clock algorithm:
    - add reference bit associated with each frame
    - when frame is filled set bit to 0 by OS
    - if frame is accessed set bit to 1 by hardware
    - Arrange all frames in a circle
    - Pointer to next frame to consider replacing
        - If ref = 0, replace this frame
        - Else set bit to 0
        - Advance clock hand and repeat until a frame is found

Which is better?:

OPT $\geq$ LRU $\gtrsim$ Clock $\gtrsim$ FIFO

Resident Set: process' pages in memory
- Local: limit frame selection to requesting process
- Global: select and frame from any process

Working Set: what are the most important pages

Working set: $W(t, \Delta)$
- Pages referenced during last delta (process time)

Process given frames to hold working set

Add/remove pages according to $W(t, \Delta)$

If working set doesn't fit, swap process out

Files and File System

Monday, March 13, 2023 6:32 PM

File: logical unit of storage, container of data
- Accessed by <name, region within file>

Goals:
- Archival storage: keep forever including previous versions
- Support various storage technologies
- Best achieve / balance: performance, reliability, security

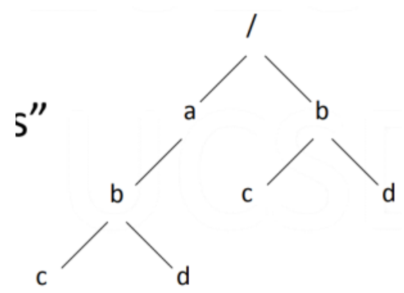
File System: a structured collection of files
- Access control - who is allowed access?
- Name Space - how is the name of the file structured (path)
- Persistent storage - how is the data physically stored

  Abstraction:
  - Objects are data, programs, for system or users
  - Objects referenced by name, to be read/written
  - Persistent - remains "forever"
  - Large - "unlimited" size
  - Sharing and control access
  - Security: protecting information

  Objects:
  - Anything that can be accessed by name
  - Can be read or written
  - Can be protected
  - Can be shared
  - Can be locked
  - IE: IO devices (disk, keyboard, display), Process memory

Hierarchical Namespace, File Model

Monday, March 13, 2023 6:44 PM



Name space organized as a tree
- Name has components, branches start from root
- No size restrictions
- Intuitive for users

IE: UNIX "path names"
- Absolute: /a/b/c
- Relative b/c relative to /a
- Not strictly a tree: links

File attributes:
- Type (user or system)
- Times: creation, accessed, modified
- Sizes: current size, maximum size
- Permissions

File Operations:
- Create, delete
- Prepare for access: open, close, mmap
- Access: read, write
- Search: move to location
- Attributes: get, set (permissions)
- Mutual exclusion: lock, unlock
- Name management: rename

Read/Write Model, Memory Mapped Model, Access Model

Monday, March 13, 2023 7:03 PM

Read/Write Model: read/write COPY of file in memory

fd= open(filename, mode) : opens the file, returns the file descriptor
nr = read(fd, data_buffer, data_size) : read from the file and store to the buffer, returns the actual amount read
nw = write(fd, buf, size) : write to the file from the buffer, returns the
close(fd) : close the file

Memory Mapped Model:

addr = mmap(fd, NULL, n) : loads the fd into array
addr[index] …

Problem: how is the file actually updated?

Access Model: How are files shared to varying degrees

- Who can access control?
- What operations are allowed

UNIX: r/w/x permissions for owner, group and everyone

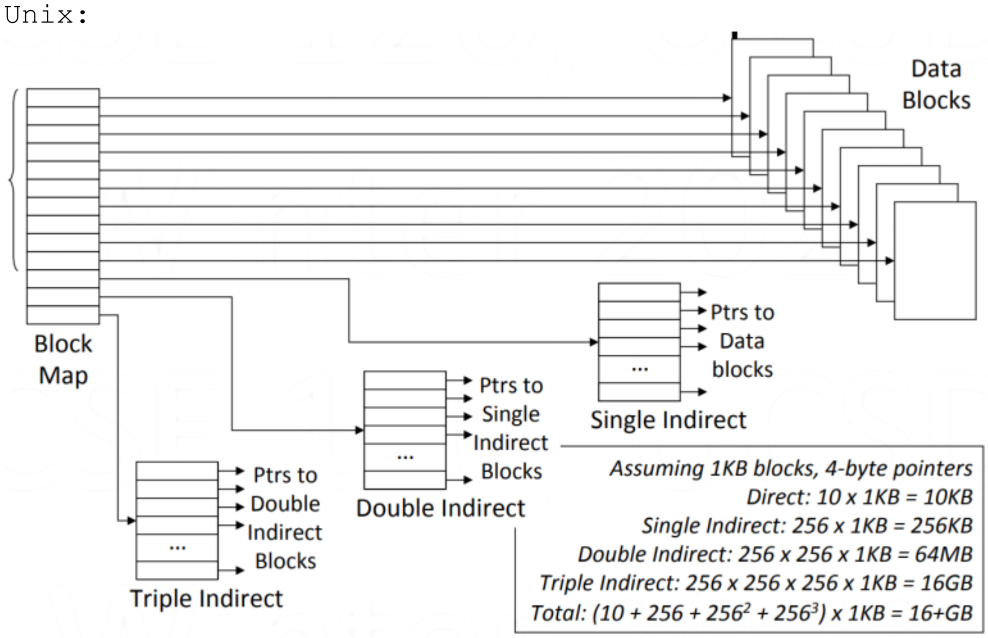
Storage Abstraction

Monday, March 13, 2023 7:30 PM

Idea: Use blocks to hide complexity of device
- Model storage as array of blocks of data
- Randomly addressable by block number
- Typical block size: 1kB, 4kB - 64kB

Simple interface:
- read(block_num, mem_addr)
- write(block_num, mem_addr)

Disk regions:
- File System Metadata -
  - Information about the file system
  - Files metadata in use, free entries
  - Data blocks in use, free entries
- File Metadata: file control blocks
  - Information about a file
    - Attributes: type of file, size, permissions
  - References to data blocks
    - Contiguous blocks: pointer to the first block and size of sequence
    - Groups of contiguous blocks: store multiple sequences of contiguous blocks
    - Non-contiguous blocks: each block individually named
  - Unix:



- Keeping track of free blocks:
  - Free block map: pointer to free block and size of free span
  - Doubly linked list
  - Bit map: set bit to 1 if block is occupied
- Data Blocks - file contents