CS3800 Exam Review

Chapter 1: overview

interrupt
memory
processors

four components:
1) CPU
2) Main Memory
3) I/O Modules
4) System Bus

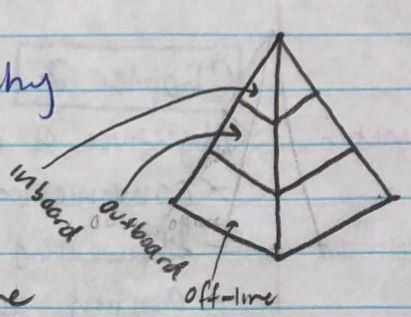
purpose of interrupts: improve processor utilization

Types of Interrupt:
1) program
2) timer
3) I/O
4) hardware failure

Sequential vs. nested

memory constraints:
- speed
- amount
- expense

★ memory hierarchy
gang down:
- decreasing $/bit
- increasing capacity
- increasing access time
- decreasing frequency of access



Principle of Locality & Memory

memory references by processor tend to cluster

Cache memory – small; fast transfer to CPU
cache size: small caches have significant performance impact
block size: unit of data exchanged b/t cache & main memory

mapping function

* when one block is read in, another one may need to be replaced
* more flexible mapping
* functions = more complex circuitry

replacement algorithms:
First In, First Out vs. LRU

I/O techniques:
1) Programmed
2) Interrupt-Driven
3) Direct Memory Access

Symmetric Multiprocessors

2+ similar processors of comparable capability

Advantages:

- scaling
- performance
- availability
- incremental growth

Multicore Computer

Chip multiprocessor → combines 2+ processors on single piece of silicon

Chapter 2: Overview

function
evolution
language
memory
architecture

Objectives of an OS:
- convenience
- efficiency
- ability to evolve

Interfaces:
- Instruction set (ISA)
- ABI
- API

Functions of an OS:
- program development
- program execution
- I/O access
- controlled file access
- system access
- error detection/response
- accounting

Why do OS evolve?

1) hardware upgrades
2) new types of hardware
3) new services
4) fixes

serial processing → simple batch systems → multiprogrammed batch systems → time sharing systems

Serial processing: no OS, direct hardware interaction
Issues: scheduling, setup time

Simple batch systems: monitors (no user direct access to processor), jobs batched together & submitted

- ★ monitor controls event sequence
- ★ reads in job and gives/returns control
- ★ resident monitor is software always in memory

"control passed to job" vs. "control returned to monitor"

↓ ↓

processor is fetching/executing instructions in user program

processor is fetching/executing instructions from monitor program

JCL: tells what compiler & data to use

User Mode	vs.	Kernel Mode
- protected memory areas		- execute privileged instructions
- no-go on certain instructions		- access protected memory
- user program execution		- monitor program execution

* batched system uniprogramming → has to wait when I/O instruction is reached

↓

Solution: multiprogramming: processor can switch to other jobs if I/O is encountered

time-sharing systems: can handle multiple interactive jobs, processor time is shared among multiple users that access system simultaneously through different terminals

major advancements in development:
- processes
- info protection/security
- system structure
- memory management
- scheduling & resource management

memory management responsibilities:
1. process isolation
2. automatic allocation & management
3. modular programming support
4. protection & access control
5. long-term storage

Key ideas:
Virtual memory
&
paging

issues with security:
1. authenticity
2. data integrity
3. confidentiality
4. availability

☆ resoure allocation policies must consider efficiency, fairness, & differential responsiveness

different architectural approaches:
- microkernel architecture
- multithreading
- symmetric multiprocessing
- distributed operating systems
- object-oriented design

microkernel: only kernel functions are address space, IPC, and basic scheduling

fault tolerance: ability of a system to continue functioning despite presence of hardware or software faults

measures of fault tolerance:
- reliability, R(t): probability of correct operation up until time t
- mean time to failure → uptime / # of uptime (& MTTR)
- availability

☆ permanent vs. temporary faults
☆ spatial vs. temporal vs. information redundancy

techniques to support fault tolerance:
- process isolation
- concurrency
- virtual machines
- checkpoints & rollbacks

☆ multiprocessor OS must provide all the functionality of a multiprogramming system plus features to accomodate multiple processors

Chapter 3: processes

structure
process = program code + set of data associated with that code
↳ elements of a process:
- identifier
- state
- priority
- program counter
- memory pointers
- context data
- I/O status info
- accounting information

states

process models

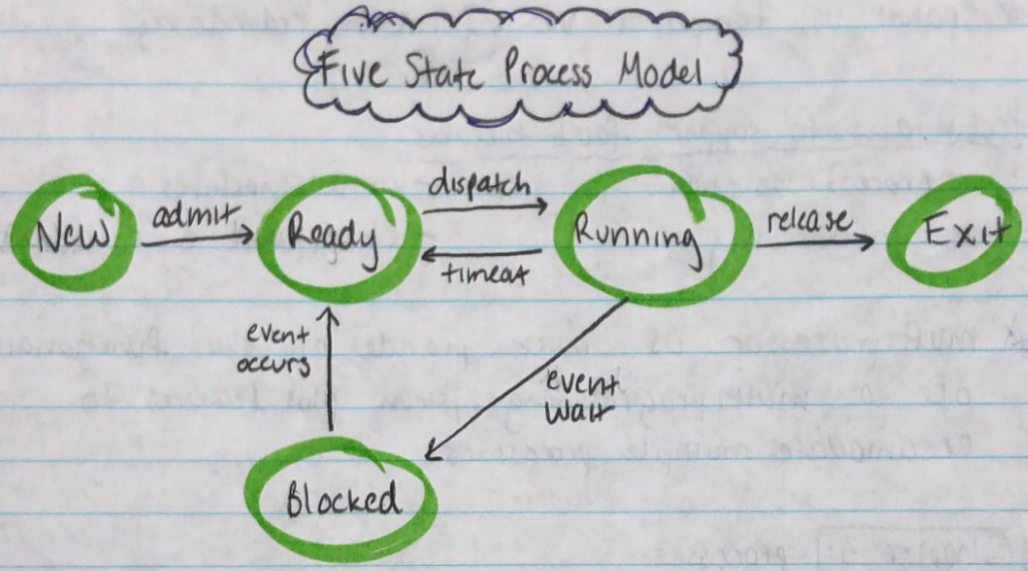
memory
↳ contained in process control block

devices

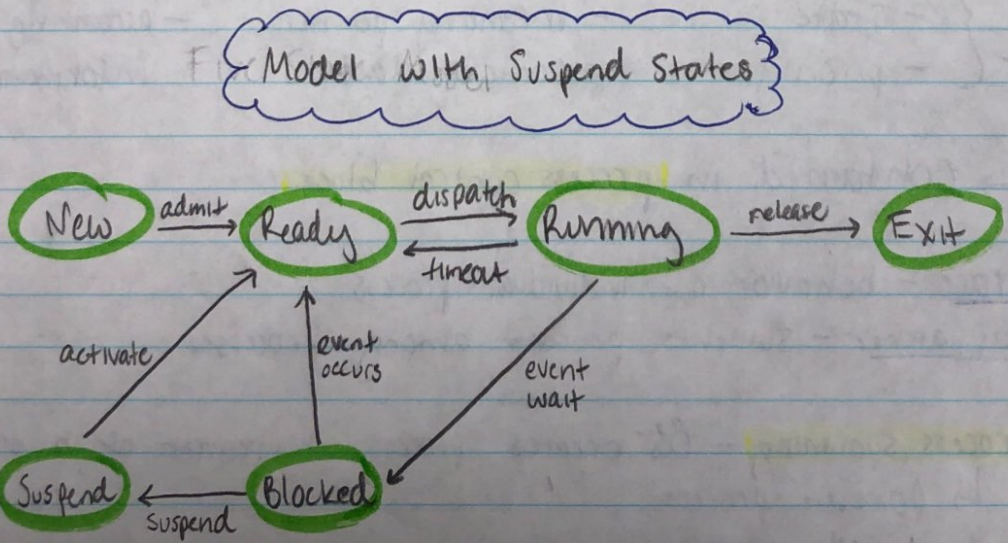
trace - behavior of individual process
dispatcher - switches processor among processes

process spawning - OS creates process at request of a process
↳ parent process
↳ child process

- reasons for process creation:
  - new batch job
  - interactive logon
  - created by OS to provide service
  - spawned by existing process
- reasons for process termination:
  - many



Swapping - involves moving part or all of a process from main memory to disk
↳ happens when none of the processes in main memory are ready



reasons for suspend states:

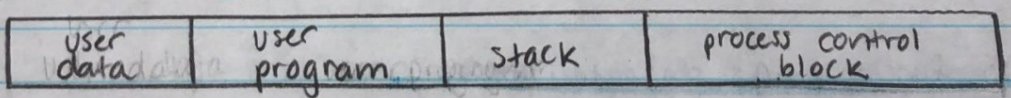
- swapping
- other OS reason
- parent process request
- interactive user request
- timing

☆ memory tables - keep track of both main (real) & secondary (virtual) memory
↳ include:
- allocation of main memory to processes
- allocation of secondary memory to processes
- protection attributes of memory blocks
- info to manage virtual memory

☆ I/O tables - OS uses to manage I/O devices & channels of system

☆ file tables - info about file existence, secondary memory location, current states, & other attributes

process image:



most important data structure in an OS:

process control block

Contains all of the information about a process that is needed by an OS

to interrupt a process:

1. interrupt - independent of process
2. trap - handles errors/exceptions, generated w/i currently running p.
3. supervisor call

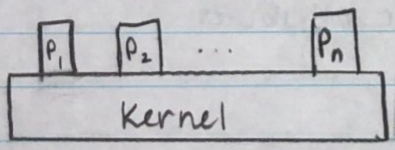
Creation of a Process:

1. assign new pid to process
2. allocate space
3. initialize PCB
4. set linkages
5. create or expand other data structures

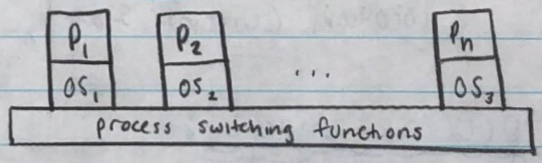
Full Process Switch

1. save processor context
2. update PCB of current running process
3. move PCB to correct queue
4. select another process
5. update PCB of selected process
6. update memory management data structures
7. restore processor context

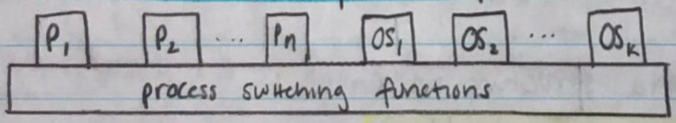
Separate kernel:



OS functions w/i user processes:



OS functions as separate processes:

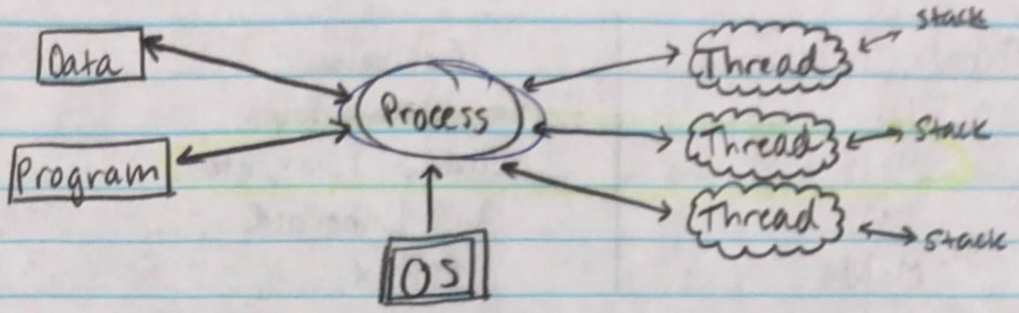


Chapter 4 threading

structure
states
types

single-threaded approaches:
- 1P : 1T
- MP : 1T

multithreaded approaches:
- 1P : MT
- MP : MT



Components of a thread:
- an execution state
- execution stack
- saved thread context (when not running)
- some static storage for local variables
- access to the memory & resources of its process

Benefits of Threads:
- takes less time to create than a process
- less time to terminate
- switching takes less time
- enhance communication efficiency

★ suspension of a process includes suspending all of its threads
★ termination of a process terminates all its threads

Ready Running Blocked

User-level threads: all thread management done by application

Kernel-level threads: all thread management is done by kernel

★ also combined approaches

Threads : Processes	Example
1:1	Unix implementations
M:1	Windows, Linux, etc
1:M	Ra, Limerald
M:N	TRIX

Chapter 5 concurrency

mutual exclusion

semaphores

deadlock

monitors

message passing

atomic operation - indivisible, guarantees isolation from concurrent processes

critical section - accessing shared resources where other processes should not interfere

deadlock - processes can't proceed due to waiting on one another

livelock - processes continually change states in response to other processes, preventing any useful work

mutual exclusion - requirement that when one process is accessing a critical section, no other one can

race condition - multiple processes reading/writing a shared data item where the final condition depends on order of execution

starvation - runnable process overlooked indefinitely

mutual exclusion support

hardware:
- uniprocessor interrupt disabling
- multiprocessor
- compare & swap

} easy, easy to verify; but "busy-waiting", starvation & deadlock possible

Semaphore

three operations:

1. initialized to nonnegative integer value
2. semWait → decrements
3. semSignal → increments

general semaphore:

```
semWait(semaphore s) {
    s.count--;
    if(s.count < 0)
        block & queue process
}
```

```
semSignal(semaphore s) {
    s.count++;
    if(s.count <= 0)
        remove & ready process
}
```

binary semaphore:

```
semWait(b_semaphore s) {
    if(s.value = one)
        s.value = zero
    else
        block & queue process
}
```

```
semSignal(b_semaphore s) {
    if(s.queue is empty)
        s.value = one
    else
        remove & ready process
}
```

mutex - binary semaphore where only the host process can release

* strong semaphore (longest released first) vs. weak semaphore (no specified order)

** code traces **

ex: producers/consumers, dining philosophers, readers/writers

Conditions for Deadlock

* mutual exclusion - only one process may use a resource at a time
* hold-and-wait - a process may hold allocated resources while waiting assignment for others
* no pre-emption - no resource can be forcibly removed from a process holding it
* circular wait - process holds resource that is needed by next resource in the chain

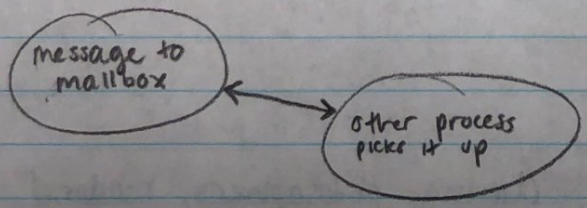
monitor - programming language construct that provides equivalent functionality to that of semaphores

↳ it procedures + initialization sequence + local data

- local data variables are accessible only by the monitor's procedures & not by any external procedure
- process enters monitor by invoking one of its procedures
- only one process executing in monitor at a time

message passing

* send primitive - includes specific identifier of the destination process
* receive primitive:
  - require that process explicitly designate a sending process
  - implicit addressing



approaches
prevention
avoidance
detection
recovery

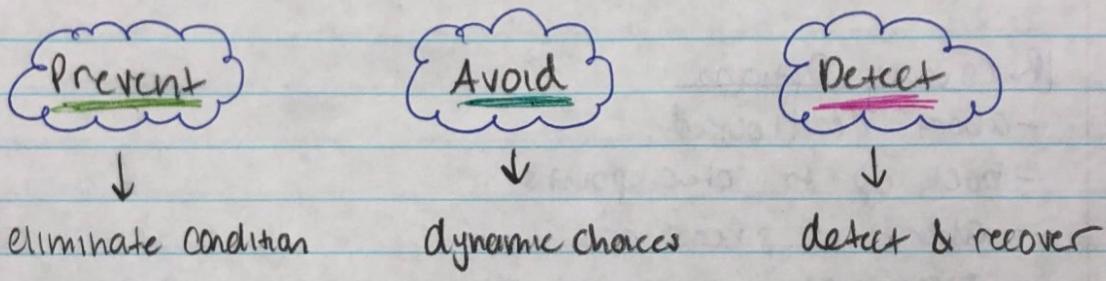
Chapter 6 deadlock

- permanent & no efficient solution
- ★ each process in the set is blocked awaiting an event that can only be triggered by another blocked process in the set

reusable resources vs. **consumable resources**

- can safely be used by one process at a time & isn't depleted
- can be created & destroyed
  - ↳ interrupts, signals, messages, info

★ deadlock conditions don't always result in deadlock



direct deadlock prevention → prevent occurence of circular wait

indirect deadlock prevention → prevent occurence of one of the three necessary conditions

resource allocation denial vs. process initiation denial

- resource allocation denial → don't give more resources
- process initiation denial → don't even start the process

deadlock avoidance

RACA

- resource vector
- claim matrix
- available vector
- allocation matrix

banker's algorithm → resource allocation denial
↳ safe vs. unsafe states

avoidance advantages	vs.	restrictions
-less restrictive than deadlock prevention		- independent processes - know resources in advance

R A R A

Deadlock Detection

- resource vector
- request matrix
- allocation matrix
- available vector

★ ignore processes not asking for more resources
★ clear processes with the available resources

Recovery Options

- abort deadlocked
- back up to checkpoints
- abort in succession
- successively preempt resources

★ OR: integrate different strategies for different situations

Chapter 7 memory management

requirements:

- relocation
- protection
- sharing
- logical organization
- physical organization

fixed
dynamic
placement
addressing
paging

logical vs. physical organization

Segmentation → linear

primary & secondary

fixed partitioning → internal fragmentation

vs.

dynamic partitioning → external fragmentation

could be solved by compaction, but that's costly & a waste of time

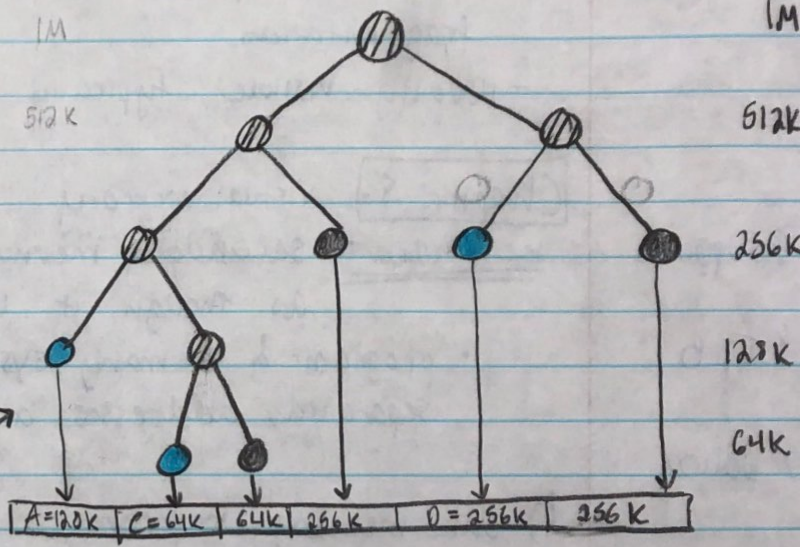
Placement Algorithms

* best-fit
* first-fit
* next-fit

Buddy System

- fixed + dynamic partitioning
- space available for allocation is treated as single block

tree diagram



non-leaf node
leaf node for unallocated block
leaf node for allocated block

logical addressing - reference to a memory location independent of the current assignment of data to memory

relative addressing - address is expressed as a location relative to some known point

physical or absolute - actual location in main memory

Paging - partition memory into equal fixed-size chunks that are relatively small
- pages - process chunks
- frames - available chunks of memory

Segmentation - subdivided into segments that may vary in length but have a maximum length

segment number + offset

☆ similar to dynamic but eliminates internal fragmentation
- usually visible, typically an organization convenience

Chapter 8 Virtual memory

Key idea: secondary memory can be addressed as though it was primary memory
- program & memory system addressing are separated, addresses are translated and mapped

- support
- TLB
- policy

1) all memory references are logical that are translated into physical at runtime
2) process may be broken up into pieces

resident set - portion of process in main memory

thrashing - too much loading & unloading; more time spent swapping than executing

- principle of locality helps avoid thrashing

Support needed:

- hardware must support paging & segmentation
- OS must be able to manage page/segment movement between main & secondary

* modified bit signals need to write to secondary

translation lookaside buffer:

special high-speed cache to avoid doubling memory access time

* be aware of pros/cons of different page sizes

fetch policy

demand vs. prepaging → ineffective if extra pages aren't referenced
↓
many page faults at beginning

placement policy – where in real memory does this go?

replacement policy – what gets replaced when something new is brought in

frame locking – page in that frame may not be replaced

- "Optimal" → wtf
- LRU → not been referenced in a while
- FIFO → circular buffer
- Clock → "use" bit

page buffering - page isn't lost but assigned to free page list or modified page list

☆ resident set management → how many pages to bring in? OS must decide

fixed-allocation - process only has certain # of frames

variable-allocation - # of frames can vary

Global or local replacement

Global → all unlocked pages

local → only among resident pages of process that caused page fault

Cleaning Policy

demand cleaning vs. precleaning

demand cleaning → only when selected for replacement

precleaning → batches

load control - # of processes resident in main