Contents

Final Prep Notes

Introduction

- most fundamental system program is the operating system
- it manages the computer's resources and provides a base for application development
- Hardware → Kernel → Applications → User
- it does resource management and machine abstraction
- resources:
  - CPU
  - memory
  - disks
  - other devices
- machine abstraction:
  - processes
  - threads
  - files
  - shell
- process, files systems, device abstraction can be modeled as trees
- system calls hide the complexity of the real machine
- single-tasking, multi-tasking, single-user, multi-user are what they sounds like
- real-time operating system – should not have a time delay on the input
- distributed – separate nodes that are networked
- embedded os – specific tasks for non-pcs
- library os – single address-space machine that has the bare minimum of stuff for programs to run and they are then compiled and run directly on hardware without an os
- microkernel: bare minimum runs in kernel space, the rest in user mode
- hybrid kernel: mix of things – Windows, OS X
- monolithic kernel: everything but applications run in kernel mode – Linux

History

1945 to 1955

- plug boards, punch cards

1955 to 1965

- transistors
- assembly languages
- programming languages
- jobs, job control systems, batch systems (jobs without user input than can be run as resources permit)

1965 to 1980

- integrated circuits
- major os projects: IBM OS/360, CTSS, MULTICS
- multi threading
- time sharing
- UNIX 1970, Ken Thompson, Dennis Ritchie
- CTSS -> Multics -> UNIX
- first in asm, then later in C

1980 to 1990

- large scale integration circuits – CPUs
- microcomputers
- PCs
- DOS (apple, microsoft,...)
- GNU, BSD

1990 to present

- internet
- mobile boom
- iot

Boot Process

1. power up
2. internal firmware
   - BIOS
   - UEFI
   - custom stuff
3. boot loader
   - multiple stages
   - MBR, GPT
4. kernel
   - CPU and hardware setup
5. initialization service
   - init, upstart, systemd, launchd
6. user space programs

  - login, shell

Processses

- executable format: structure of an exe
- produced by a linker
- format header, program header, then links to code, global and static data
- common formats: PE, ELF, Mach-O
- each process has: pid, state, priority, cpu state (registers, program counter, stack pointer), virtual memory (memory map), opened files, code, data, parent id, working directory, exit status,. . .
- the kernel maintains a list of all the process descriptors
- process states: running, ready, blocked for scheduling
- context switching to give all exes an equal amount of time
- processes can be created and exit when they are done
- io devices can interrupt, the clock can interrupt

Context Switching

1. save current cpu state
2. load a CPU state from next executable
3. run that one

Threads

- lightweight process that shares some elements with its parent process
- e.g. memory address space, opened files, global data, . . .
- they tend to have a private stack, registers, state
- same data structure for processes and threads

Scheduling

- either compute-bound or IO-bound processes
- the time-bottleneck is either computation or IO
- compute-bound: long CPU bursts, little IO
- IO-bound: short CPU bursts, a lot of IO
- a scheduler can start and stop these processes to make a efficient use of the system resources
- preemptive scheduling – mostly by priority
- non-preemptive goes by what processes finish when
- fairness and balance are common goals

Batch Scheduler

- maximize throughput
- minimize turnabout
- maximize CPU utilization
- first-come first-served
- shortest job first
- shortest remaining time next

Interactive Scheduler

- minimize response time
- round-robin scheduling
- shortest process next
- lottery scheduling
- fair-share scheduling

Real-time Scheduler

- meet deadlines
- hard real time vs soft real time:
  - hard real time have deadlines that must be met
  - soft real time has a certain tolerance but still not desirable
- periodic or non-periodic events

Book Notes

Processes

- process – a program in execution; progresses in sequential fashion, i.e. no parallel execution
- made up of
  - code (text section)
  - program counter
  - registers
  - stack
  - global variables, heap
- program – passive, process – active
- processes have states
  - new – just created
  - running – in execution
  - waiting – waiting for an event to occur
  - ready – waiting for a processor
  - terminated – execution has finished
- each process has a control block that contains all the relevant information
- if there are multiple program counters, we can execute multiple parts of the program at once – threads
- in linux this is represented in `task_struct`
- the process scheduler selects which process gets CPU time, goal is to use resources as efficiently as possible, and to make the system reponsive
- you have multiple queues, some for ready processes, waiting processes
- context switch – switch from one process to another
- during a context switch, the current process state is saved, and the new process state is loaded
- processes can be created and terminated (`fork`, `execve`, `wait` and `exit`, `abort`)
- processes can be independent or cooperating, they can share memory or pass messages
- if processes cooperate, coordination is very tricky and important (e.g. race conditions)
- socket – endpoint for communication
- concatenation of IP address and port, e.g. 192.169.1.1:22
- external data representation handles the data transmission to make it simpler

Threads and Concurrency

- threads can be used to split certain responsibilities off from the main program
- thread creation is lightweight compared to process creation
- threads can increase efficiency and simplify code
- threads
  - increase responsiveness, e.g. for UI, when a process is blocked
  - tend to share memory, making communication easier
  - are more economic, creation and switching is a lot cheaper
  - are scalable, allowing programs to take advantage of multicore architectures
- parallelism – more than one task performed at the same time
- concurrency – more than one task making progress at the same time
- two types of threads – user level threads (POSIX Pthreads, Java, etc) and kernel threads (windows, linux, iOS, etc)
- user threads and kernel threads can either be one-to-one, many-to-many, or many-to-one (user to kernel), or a mix of the two
- with more threads being able to be used, programmers can hardly do all the checking – compilers or run-time libraries are there to help
- we can have
  - thread pools – pool of many threads where they await work, simple, possibly slightly faster
  - fork-join – tasks are forked and then joined again
  - openmp – compiler directives for C, C++, FORTRAN, creates a parallel reagion
  - grand central dispatch – blocks are assinged to queues and then executed when possible
  - intel threading building blocks – template for parallel C++
- threading has issues like semantics of syscalls, signal handling, thread cancellation, thread-local storage, scheduler activations

Scheduling

- goal: maximum CPU utilization
- generally we have a CPU/IO cycle – CPU burst, then I/O burst
- distribution of CPU bursts is the main concern
- CPU scheduler selects from the processes in the ready queue and allocates them to a CPU core
- the scheduler is involved when a process
  1. switches from running to waiting
  2. switches from running to ready state
  3. switches from waiting to ready
  4. terminates
- for 2. and 3. the scheduler can make optimizing decisions
- scheduling for 1. and 4. is nonpreemptive, for 2. and 3. it can be preemptive
- nonpreemptive scheduling – a process keeps the CPU until it terminates or switches to waiting
- basically all modern operating systems use preemptive scheduling
- this can result in race conditions is data is shared between multiple processes
- the dispatcher switches the context depending on the scheduler's decision
- scheduling criteria
  1. CPU utilization – as much as possible
  2. throughput – highest number of completed processes per time unit
  3. turnaround time – time it takes to execute a particular process

4. waiting time – time a process spent in the waiting queue
5. response time – time from request submission to first response

- optimization criteria: maximize 1. and 2., minimize 3., 4,. and 5.

First-Come, First-Serve (FCFS) Scheduling

- processes are scheduled in the order that they arrive in
- very simple, but hardly optimal

Shortest-Job-First (SJF) Scheduling

- order processes by required CPU time, then run them from shortest to longest
- this results in the minimum average time for a given set of processes
- knowing the required time is the issue here

Round Robin (RR) Scheduling

- each process gets a small amount of time (time quantum q), 10-100 ms
- after q, the process is preempted and added to the end of the ready queue
- this means that no process waits more than $(n-1)q$ time for n processes
- the turnaround is worse than SJF, but the response time is way better
- q should be chosen such that 80% of CPU bursts are shorter than it

Priority Scheduling

- processes are assigned a priority rating
- then the process with the highest priority is executed first
- starvation can occur when low-priority processes never execute
- aging can help – priority increases as the process waits longer
- can be mixed with RR, processes with the same priority are run in RR

Multilevel Queue Scheduling

- we have multiple execution queues based on a processes priority
- priority decreases from real-time processes, system processes, interactive processes, batch processes

Multilevel Feedback Queue

- processes can move between queues (e.g. aging)
- multiple queues with different q and maybe also systems

Thread Scheduling

- when threads are supported, those get schedules instead of processes

Multiprocessor Scheduling

- scheduling becomes a lot more complex when multiple processing units are available
- each processor can be self-scheduling
- software threads are ordered by the OS, then the CPU preprocessors decide which of those threads to run
- tasks are then also distributed between differently loaded cores

- NUMA-awareness will assign threads closest to where the memory they want to use is
- soft and hard scheduling can happen (deadlines are moveable or not)

Earliest Deadline First (EDF) Scheduling

- the process that has the earliest deadline is executed first

POSIX Real-Time Scheduling

- allows managing of real time threads
- either uses FCFS or RR for the scheduling of equal-priority tasks

Virtual Memory

- code needs to be in memory to be executed, but the entire program rarely needs to be in memory
- if only a part of the program is loaded, memory size is less of an issue, programs use less memory while running, more programs running at the same time reduced the need for IO operations and loading programs
- virtual memory – user only sees logical memory, physical memory is abstracted
- you only need part of the program in memory to execute it
- logical address space can be way larger than physical address space
- address space can be share between processes
- process creation is more efficient
- more concurrency, less IO to swap memory
- virtual address space – logical view of how processes are stored in memory
  - usually start at address 0
  - physical memory organized in page frames
  - MMU maps logical memory to physical memory
- virtual memory can be implemented using either demand paging or demand segmentation
- generally code and data at beginning of memory, heap grows up from there, stack starts at end and grows down
- system resources can be shared by mapping them into virtual memory
- demand paging
  - can bring entire process into memory on load or just what is needed
  - less IO, faster response
  - can have lazy swapper – never does anything until it is necessary
  - memory is requested, if it is not there, a page fault happens and the required memory has to be fetched
- a whole bunch on how to organize pages and how to handle them
- Buddy and Slab allocators

Mass Storage

- HDD – spinning magnetic platters
- NVM – nonvolatile memory: SSD, USB, etc
- address mapping of disks is as a large 1D array of logical blocks, is the smallest unit of transfer
- a bunch of stuff on drive and storage management

File System Internals

- computers have storage devices, those are split into partitions, which hold volumes
- each volume is formatted into a file system
- many options
- boot partition contains information and pointers about where to find the kernel
- root partition contains the OS
- at mount time the consistency of the data is checked
- file sharing between users is a possibility
- permission and protections must be in place to regulate this process
- virtual file system (VFS) on UNIX provides a good abstraction between the OS calls and the actual implementation, see the functions `open`, `close`, `read`, `write`, `mmap`
- for remote file systems, DNS and other naming services provide the information required to use the distributed computing stuff

File System Interface

- file – contiguous logical address space
- many different types: numeric, character, binary, program
- contents are defined by the file's creator
- has attributes: name, ID, type, location, size, time, date, access
- file operations: create, write and read at pointer location, reposition (seek), delete, truncate, open (find file and move to memory), close (move file back to memory)
- open files: list of opened files, file pointer by process, access rights, disk location
- locking mediates who can do what at a certain time
- directories are nodes that contain information on the files in them
- we generally deal with general-purpose fs, but special-purpose ones exist
- directories allow a quick, efficient, comfortable organization and location of files
- single-level directories are nor that great, there is no differentiation
- tree-structured dirs are better, they have more organization, paths, same names on different paths without conflict
- links in file systems can lead to connections that are not part of a standard graph – acyclic directories
- one can guarantee not having cycles by only linking to files and not directories
- file systems must be mounted before they can be used
- access rights can be handled using user IDs and group IDs
- file permissions are set as RWX, for owner, group, public

File System Implementation

- file systems tend to be layered: apps, logical file system, file-organization modules, basic file system, IO control, divices
- file control blocks (all metadata) are called `inodes` in UNIX
- layering reduces complexity and redundancy, but also adds overhead
- the mount table stores file system mounts, mount points, file system types
- directories can be implemented either as a linear list (simple), or as a hash table (faster, but more complex)
- allocation can be contiguous (files are stored in a set of successive blocks), finding space is difficult, fragmentation occurs
- allocation can also be linked, basically a linked list for the blocks of each file, slower, more complicated, but no fragmentation, finding a file is more time consuming

- allocation may furthermore be indexed, meaning there is a table for each file that contains the indices of the blocks of that file relative to a starting index
- UNIX FS has a combined scheme of indices to indices etc
- the fs also has a list of free blocks that can be used for allocation
- this list can also be a linked list, which makes it simpler to access the blocks
- HDDs simply overwrite free blocks and are OK with that, NVM cannot do that so it needs TRIMing – free blocks need to be properly erased
- frequently used hard disk info can be cached or mapped into memory for faster access