Ch.1 - Introduction

• An OS is a program that acts as an intermediary between a user of a computer and the computer hardware
• Goals: Execute user programs, make the comp. system easy to use, utilize hardware efficiently
• Computer system:
    Hardware ↔ OS ↔ Applications ↔ Users (↔ = 'uses')
• OS is:
    ◦ Resource allocator: decides between conflicting requests for efficient and fair resource use
    ◦ Control program: controls execution of programs to prevent errors and improper use of computer

The OS is composed of:
    Kernel, system programs, and middleware.

System programs: ships with the operating system, but not part of the kernel
Middleware: a set of software frameworks that provide additional services to application developers

Application programs: all programs not associated with the operating system

• Kernel: the one program running at all times on the computer
• Bootstrap program: loaded at power-up or reboot
    ◦ Stored in ROM or EPROM (known as firmware), Initializes all aspects of system, loads OS kernel and starts execution

• I/O and CPU can execute concurrently
• Device controllers inform CPU that it is finished its operation by causing an interrupt
    ◦ Interrupt transfers control to the interrupt service routine generally, through the interrupt vector, which contains the addresses of all the service routines
    ◦ Incoming interrupts are disabled while another interrupt is being processed
    ◦ Trap is a software generated interrupt caused by error or user request
    ◦ OS determines which type of interrupt has occurred by polling or the vectored interrupt system

• System call: request to the operating system to allow user to wait for I/O completion
• Device-status table: contains entry for each I/O device indicating its type, address, and state
    ◦ OS indexes into the I/O device table to determine device status and to modify the table entry to include interrupt

• Storage structure:
◦ Main memory – random access, volatile

◦ Secondary storage – extension of main memory that provides large non-volatile storage
◦ Disk – divided into tracks which are subdivided into sectors. Disk controller determines logical interaction between the device and the computer.

• Caching – copying information into faster storage system
• Multiprocessor Systems: Increased throughput, economy of scale, increased reliability
    ◦ Can be asymmetric or symmetric
    ◦ Clustered systems – Linked multiprocessor systems

Asymmetric Multiprocessing – each processor is assigned a specific task.
Symmetric Multiprocessing – each processor performs all tasks

• Multiprogramming – Provides efficiency via job scheduling
    ◦ When OS has to wait (ex: for I/O), switches to another job

• Timesharing – CPU switches jobs so frequently that each user can interact with each job while it is running (interactive computing)

Interrupt driven
    Hardware interrupt by one of the I/O devices
    Software interrupt (**exception** or **trap):**

• Dual-mode operation allows OS to protect itself and other system components – User mode and kernel mode
    ◦ Some instructions are only executable in kernel mode, these are privileged

• Single-threaded processes have one program counter
• Multi-threaded processes have one PC (program counter) per thread

• Protection – mechanism for controlling access of processes or users to resources defined by the OS
• Security – defense of a system against attacks
• User IDs (UID), one per user, and Group IDs, determine which users and groups of users have which privileges

Ch.2 – OS Structures
• User Interface (UI) – Can be Command-Line (CLI) or Graphics User Interface (GUI) or Batch
    ◦ These allow for the user to interact with the system services via system calls (typically written in C/C++)

• Other system services that a helpful to the user include:

program execution, I/O operations, file-system manipulation, communications, and error detection
• Services that exist to ensure efficient OS operation are: resource allocation, accounting, protection and security
• Most system calls are accessed by Application Program Interface (API) such as Win32, POSIX, Java
• Usually there is a number associated with each system call
    ◦ System call interface maintains a table indexed according to these numbers

• Parameters may need to be passed to the OS during a system call, may be done by:
    ◦ Passing in registers, address of parameter stored in a block, pushed onto the stack by the program and popped off by the OS
    ◦ Block and stack methods do not limit the number or length of parameters being passed

• Process control system calls include:
end, abort, load, execute, create/terminate process, wait, allocate/free memory

• File management system calls include:
create/delete file, open/close file, read, write, get/set attributes

• Device management system calls:
request/release device, read, write, logically attach/detach devices

• Information maintenance system calls:
get/set time, get/set system data, get/set process/file/device attributes

• Communications system calls:
create/delete communication connection, send/receive, transfer status information

• OS Layered approach:
    ◦ The operating system is divided into a number of layers (levels), each built on top of lower layers. The bottom layer (layer 0), is the hardware; the highest (layer N) is the user interface
    ◦ With modularity, layers are selected such that each uses functions (operations) and services of only lower-level layers

• Virtual machine: uses layered approach, treats hardware and the OS kernel as though they were all hardware.
    ◦ Host creates the illusion that a process has its own processor and own virtual memory
    ◦ Each guest provided with a 'virtual' copy of the underlying computer

• Application failures can generate core dump file capturing memory of the process
• Operating system failure can generate crash dump file containing kernel memory

System programs provide a convenient environment for program development and

execution.
They can be divided into:

1. File manipulation
2. Status information sometimes stored in a File modification
3. Programming language support
4. Program loading and execution
5. Communications
6. Background services
7. Application programs

Ch.3 – Processes

Process/job – a program in execution

• Process contains a program counter, stack, and data section.
  ◦ Text section: program code itself
  ◦ Stack: temporary data (function parameters, return addresses, local variables)
  ◦ Data section: global variables
  ◦ Heap: contains memory dynamically allocated during run-time

• Process Control Block (PCB): contains information associated with each process: process state, PC, CPU registers, scheduling information, accounting information, I/O status information

Process state:

1. **new**: The process is being created
2. **running**: Instructions are being executed
3. **waiting**: The process is waiting for some event to occur
4. **ready**: The process is waiting to be assigned to a processor
5. **terminated**: The process has finished execution

• Types of processes:
  ◦ I/O Bound: spends more time doing I/O than computations, many short CPU bursts
  ◦ CPU Bound: spends more time doing computations, few very long CPU bursts

• When CPU switches to another process, the system must save the state of the old process (to PCB) and load the saved state (from PCB) for the new process via a context switch
  ◦ Time of a context switch is dependent on hardware

Short-term scheduler (or **CPU scheduler**) – selects which process should be executed next and allocates CPU

Sometimes the only scheduler in a system

Short-term scheduler is invoked frequently (milliseconds) ⇒ (must be fast)

Long-term scheduler (or **job scheduler**) – selects which processes should be brought into the ready queue

Long-term scheduler is invoked infrequently (seconds, minutes) ⇒ (may be slow)

The long-term scheduler controls the **degree of multiprogramming**

• Parent processes create children processes (form a tree)
◦ PID allows for process management
◦ Parents and children can share all/some/none resources
◦ Parents can execute concurrently with children or wait until children terminate
◦ fork() system call creates new process ▪ exec() system call used after a fork to replace the processes' memory space with a new program

A zombie process: no parent waiting (did not invoke wait()) process
An orphan process: parent terminated without invoking wait(), leaves the child process as an orphan process
• Cooperating processes need interprocess communication (IPC):
shared memory or message passing

Advantages of process cooperation
1. Information sharing
2. Computation speed-up
3. Modularity
4. Convenience

• Message passing may be blocking or non-blocking
◦ Blocking is considered synchronous
▪ Blocking send has the sender block until the message is received
▪ Blocking receive has the receiver block until a message is available

◦ Non-blocking is considered asynchronous
▪ Non-blocking send has the sender send the message and continue
▪ Non-blocking receive has the receiver receive a valid message or null
Rendezvous: both send and receive are blocking

What is a process?
Process states
new/ready/running/waiting/terminated
How does OS represent a process? PCB
Task-struct
Process schedulers:
long-term, short-term, and medium-term
context switch
Scheduling queues
job queue, ready queue, device queues
Interprocess communication methods:
shared memory/message passing
Process creation and termination
fork(), exit(), abort(), wait(), execlp(), execvp()
zombie and orphans
Understand process tree

Client-server communication methods:

sockets/rpc/pipes

Two kinds of pipes:

Ordinary / Named pipes

Ch.4 – Threads

• Threads are fundamental unit of CPU utilization that forms the basis of multi-threaded computer systems

• Process creation is heavy-weight while thread creation is light-weight

◦ Can simplify code and increase efficiency

• Kernels are generally multi-threaded

• Multi-threading models include: Many-to-One, One-to-One, Many-to-Many

◦ Many-to-One: Many user-level threads mapped to single kernel thread

◦ One-to-One: Each user-level thread maps to kernel thread

◦ Many-to-Many: Many user-level threads mapped to many kernel threads

• Thread library provides programmer with API for creating and managing threads

• Issues include: thread cancellation, signal handling (synchronous/asynchronous), handling thread-specific data, and scheduler activations.

◦ Cancellation:

▪ Asynchronous cancellation terminates the target thread immediately

▪ Deferred cancellation allows the target thread to periodically check if it should be canceled

◦ Signal handler processes signals generated by a particular event, delivered to a process, handled by default signal handler or user supplied signal handler.

◦ Scheduler activations provide upcalls – a communication mechanism from the kernel to the thread library.

▪ Allows application to maintain the correct number of kernel threads

Thread-local storage (**TLS**) allows each thread to have its own copy of data. TLS is unique to each thread, and similar to static data.

What is a thread? A thread is a flow of control within a process.

The benefits of multithreading:

increased responsiveness to the user, resource sharing within the process, economy, and scalability factors, such as more efficient use of multiple processing cores.

User-level threads vs kernel threads

Three models relate user and kernel threads: many-to – one, one-to-one, many-to-many.

Most modern operating systems provide kernel support for threads.

Thread libraries provide API for creating and managing threads.

Implicit threading: thread pools, OpenMP, and Grand Central Dispatch.

Issues:

the semantics of the fork() and exec() system calls.
signal handling, thread cancellation, thread-local storage, and scheduler activations.
How to calculate speedup using Amdahl's law

$$speedup \leq \frac{1}{S + \frac{(1-S)}{N}}$$

Ch.5 – CPU Scheduling
• Process execution consists of a cycle of CPU execution and I/O wait
• CPU scheduling decisions take place when a process:
  ◦ Switches from running to waiting (nonpreemptive)
  ◦ Switches from running to ready (preemptive)
  ◦ Switches from waiting to ready (preemptive)
  ◦ Terminates (nonpreemptive)

• The dispatcher module gives control of the CPU to the process selected by the short-term scheduler
  ◦ Dispatch latency- the time it takes for the dispatcher to stop one process and start another

• Scheduling algorithms are chosen based on optimization criteria (ex: throughput, turnaround time, etc.)
  ◦ FCFS, SJF, Shortest-Remaining-Time-First (preemptive SJF), Round Robin, Priority

You should know how to calculate waiting time, average waiting time, throughput time and average throughput time for the above scheduling algorithms.

• Determining length of next CPU burst: Exponential Averaging:

1. t_n = actual length of n^{th} CPU burst
2. τ_{n+1} = predicted value for the next CPU burst
3. $\alpha, 0 \leq \alpha \leq 1$
4. Define:

$$\tau_{n=1} = \alpha\, t_n + (1-\alpha)\tau_n.$$

(commonly α set to 1/2)

• Priority Scheduling can result in starvation, which can be solved by aging a process (as time progresses, increase the priority)
• In Round Robin, small time quantums can result in large amounts of context switches
  ◦ Time quantum should be chosen so that 80% of processes have shorter burst times than the time quantum

• Multilevel Queues and Multilevel Feedback Queues have multiple process queues that have different priority levels

- In the Feedback queue, priority is not fixed → Processes can be promoted and demoted to different queues
- Feedback queues can have different scheduling algorithms at different levels

What is the preemptive scheduling algorithm?

Preemptive scheduling allows a process to be interrupted in the midst of its execution, taking the CPU away and allocating it to another process.

• Multiprocessor Scheduling is done in several different ways:

- Asymmetric multiprocessing: only one processor accesses system data structures → no need to data share
- Symmetric multiprocessing: each processor is self-scheduling (currently the most common method)
- Processor affinity: a process running on one processor is more likely to continue to run on the same processor (so that the processor's memory still contains data specific to that specific process)

• Little's Formula can help determine average wait time per process in any scheduling algorithm:

- $n = \lambda \times W$
- n = avg queue length; W = avg waiting time in queue; λ = average arrival rate into queue

• Simulations are programmed models of a computer system with variable clocks ◦ Used to gather statistics indicating algorithm performance

- Running simulations is more accurate than queuing models (like Little's Law)
- Although more accurate, high cost and high risk

Real-time CPU scheduling

1. CPU scheduling is the task of selecting a waiting process from the ready queue and allocating the CPU to it. The CPU is allocated to the selected process by the dispatcher.
2. Scheduling algorithms:
    a. FCFS(First-Come,First-Served)
    b. SJF(Shortest-Job-First)
    c. RR(Round-Robin)
    d. Priority-Based
    e. Multilevel Queue
    f. Multilevel Feedback Queue

The FCFS algorithm is nonpreemptive; the RR algorithm is preemptive.
The SJF and priority algorithms may be either preemptive or nonpreemptive.
Starvation problem => aging (solution)

Multilevel queue algorithms allow different algorithms to be used for different classes of processes. The most common model includes a foreground interactive queue that uses RR scheduling and a background batch queue that uses FCFS scheduling.

Multilevel feedback queues allow processes to move from one queue to another.

1. Thread Scheduling
    a. Process-contention scope
    b. System-contention scope
2. Multiprocessor scheduling

Typically, each processor maintains its own private queue of processes (or threads), all of which are available to run. Additional issues related to multiprocessor scheduling include processor affinity, load balancing, and multicore processing.

1. Real-time CPU Scheduling
    a. Priority-Based
    b. Rate-Monotonic
    c. Earliest-Deadline-First
    d. Proportional Share
2. The POSIX Pthread API provides various features for scheduling real-time threads.

Explain the difference between preemptive and non-preemptive scheduling.

Preemptive scheduling allows a process to be interrupted in the midst of its execution, taking the CPU away and allocating it to another process. Non-preemptive scheduling ensures that a process relinquishes control of the CPU only when it finishes with its current CPU burst.