Ch.1 – Introduction

•OS definition, goals

* OS has no formal definition, just is the stuff you need for a computer.
* Goal is to let the users use the computer in a way friendly to them, while still being secure to protect from attacks and errors
* Flexible, reliable, error free, efficient

•Computer system structure

•Kernel

* The one process always running on a computer (also called OS)

•Interrupts

oInterrupt vector

oPolling vs. vectored interrupt system

•Multiprocessor architectures (symmetric vs. asymmetric)

• Also known as multiprocessor systems, tightly-coupled systems

• Processors share memory and a clock; communication usually takes place through the shared memory

• Advantages include:

1. Increased throughput 2. Economy of scale 3. Increased reliability – graceful degradation or fault tolerance

• Two types:

1. Asymmetric Multiprocessing – each processor is assigned a specific task. 2. Symmetric Multiprocessing – each processor performs all tasks

• Symmetric multiprocessing (SMP)

• All processors are peers

• Kernel routines can execute on different CPUs, in parallel

• Asymmetric multiprocessing (AMP)

• Master/slave structure

• The kernel runs on a particular processor

• Other CPUs can execute user programs and OS utilities

•Dual-mode (user mode vs. kernel mode)

Ch.2 – OS Structures

Shared Memory V Message Passing

* Shared mem faster but less safe due to potential for simultaneous read/write
* Message passing more secure but slower

•System calls

oAPIs

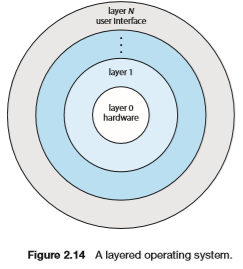
Application Programming Interface

* POSIX (UNIX systems)
* Windows
* Java

oParameter passing

•Pros and cons of the following structures:

oOS layered approach



Simplicity of construction and debugging, but very difficult to define the functionality of each layer, and performance suffers

oMicrokernels (DARWIN)

A kernel with all non-essential components stripped to create a manageable kernel. All stripped components are implemented as user-level programs

Extending OS is easier, but performance suffers

oModules

Provides many of the pros of microkernels and layering with none of cons by breaking kernel up into loadable modules.

Ch.3 – Processes

•Process definition

• Process – a program in execution

• process execution must progress in sequential fashion

• A program is a passive entity, whereas a process is an active entity with a program counter and a set of associated resources

• Each process has its own address space

• The program code, also called text section

• Stack containing temporary data

• function parameters, return addresses, local variables

• Data section containing global variables

• Heap containing memory dynamically allocated during run time

• Program counter and CPU registers are part of the process context

•States and state transitions

• As a process executes, it changes state

• new: The process is being created

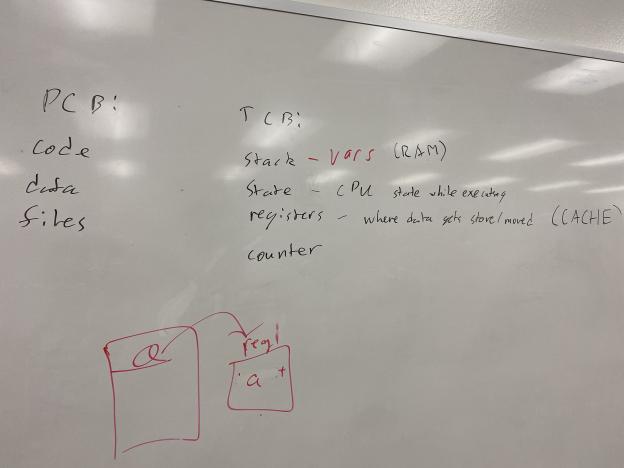
• running: Instructions are being executed

• waiting: The process is waiting for some event to occur

• ready: The process is waiting to be assigned to a processor

• terminated: The process has finished execution

•Process layout in memory

•Process control block

PCB: OS data structures to keep track of all processes

• The PCB tracks the execution state and location of each process

• The OS allocates a new PCB on the creation of each process and places it on a state queue

• The OS deallocates the PCB when the process terminates

The PCB contains:

• Process state – running, waiting, etc.

• Program counter – location of instruction to next execute

• CPU registers – contents of all process-centric registers

• CPU scheduling information – priorities, scheduling queue pointers

• Memory-management information – memory allocated to the process

• Accounting information – CPU used, clock time elapsed since start, time limits

• I/O status information – I/O devices allocated to process, list of open files

•Types of Process (CPU bound, I/O bound)

•Context switching

•Long-term and short-term scheduling

• 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

• Processes can be described as either:

• I/O-bound process – spends more time doing I/O than computations, many short CPU bursts

• CPU-bound process – spends more time doing computations; few very long CPU bursts

• Long-term scheduler strives for good process mix

•Process creation and termination

ofork(), exec(), and wait() system calls

• fork() system call creates new process

• exec() system call used after a fork() to replace the process’ memory space with a new program

• Each process has a process identifier (pid)

• The parent executes fork() system call to spawn a child

• The child process has a separate copy of the parent’s address space

• Both the parent and the child continue execution at the instruction following the fork() system call

• The return value for the fork() system call is

• Zero value for the new (child) process

• Non-zero pid for the parent process

• Typically, a process can execute a system call like exec() to load a binary file into memory

Termination

• Process executes last statement and then asks the operating system to delete it using the exit() system call.

• Returns status data from child to parent (via wait())

• Process’ resources are deallocated by operating system

• Parent may terminate the execution of children processes using the abort() system call. Some reasons for doing so:

• Child has exceeded allocated resources

• Task assigned to child is no longer required

• The parent is exiting and the operating systems does not allow a child to continue if its parent terminates

• Some operating systems do not allow child to exists if its parent has terminated. If a process terminates, then all its children must also be terminated.

• cascading termination. All children, grandchildren, etc. are terminated.

• The termination is initiated by the operating system.

• The parent process may wait for termination of a child process by using the wait()system call. The call returns status information and the pid of the terminated process

pid = wait(&status);

• If no parent waiting (did not invoke wait()) process is a zombie

• If parent terminated without invoking wait , process is an orphan

•Inter-process communication

oShared memory V Message passing (see OS Structures)

Ch.4 – Threads

•Thread definition

A basic unit of CPU utilization. Has an ID, counter, register, and state

See PCB for more

•Multithreading models

oMany-to-one

oOne-to-one

oMany-to-many

•User-level and kernel-level threads

User level are faster to create, context switch, and manage, but Kernel-level are required for system calls

•Thread libraries

Provide programmer with an API to create and manage threads

* POSIX Pthreads, Windows, Java

•Implicit threading

* Transfer control of threading to run-time libraries
* Thread pools
* limited pool of threads
* OpenMP
* A set of compiler directives to support parallel programming in shared memory environments
* Grand Central Dispatch
* Tech developed by apple. Uses a dispatch queue to handle parallel process execution

•Threading issues

ofork() and exec()

• Semantics of fork() and exec() system calls

• Signal handling

• Synchronous and asynchronous

• Thread cancellation of target thread

• Asynchronous or deferred

• Thread-local storage

• Scheduler Activations

• Does fork()duplicate only the calling thread or all threads?

• Some UNIXes have two versions of fork

• exec() usually works as normal – replace the running process including all threads

Ch.5 – CPU Scheduling

•CPU scheduling criteria

• CPU utilization – keep the CPU as busy as possible

• Throughput – # of processes that complete their execution per time unit

• Turnaround time – amount of time to execute a particular process

• Waiting time – amount of time a process has been waiting in the ready queue

• Response time – amount of time it takes from when a request was submitted until the first response is produced, not output (for time- sharing environment)

• Meeting the deadline (real-time systems)

•CPU scheduler and dispatcher

• Dispatcher mod ule gives control of the CPU to the process selected by the short-term scheduler; this involves:

• switching context

• switching to user mode

• jumping to the proper location in the user program to restart that program

• Dispatch latency – time it takes for the dispatcher to stop one process and start another running

•Preemptive vs. Non-preemptive scheduling

• Under non-preemptive scheduling, each running process keeps the CPU until it completes or it switches to the waiting (blocked) state

• Under preemptive scheduling, a running process may be forced to release the CPU even though it is neither completed nor blocked

• In time-sharing systems, when the running process reaches the end of its time quantum (slice)

• In general, whenever there is a change in the ready queue

•Scheduling algorithms

Waiting Time

• Waiting time definition

Twaiting = Tstart – Tarrival

• Average waiting time = Sum(Twaiting)/ #processes

• For now, we assume

• Average waiting time is the performance measure

• Only one CPU burst (in milliseconds) per process

• Only CPU, No I/O

• Once started, each process runs to completion

FCFS:

P1: Burst Time: 24, P2: 3, P3: 3, Order arrived: 1,2,3: Wait Time: p1=0, p2 24, p3 3 = 27

Average wait time: (0 + 24 + 27) / 3 = 17

• Advantage: simple

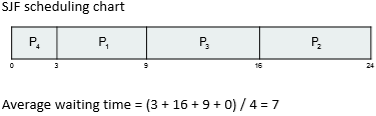
• Disadvantages:

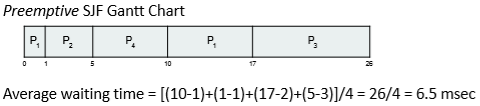
• convoy effect - short process behind long process

• may lead to poor overlap of I/O and CPU since CPU-bound processes will force I/O bound processes to wait for the CPU, leaving the I/O devices idle

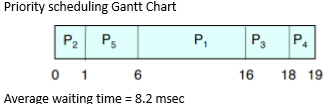
Convoy Effect

SJF example: p1 6, p2 8, p3 7, p4 3burst time



SRTF Ex: Process, Arival, Burst: p1 0,8 p2 1,4 p3 2,9, p4 3,5

RR: process, BurstTime: p1 24, p2 3, p3 3 TimeQuantum 4) Typically, higher average turnaround than SJF, but better response | q should be large compared to context switch time | q usually 10ms to 100ms, context switch < 10 usec 

Priority: Process, BTime, Priority: p1 10 3, p2 1 1, p3 2 4, p4 1 5, p5 5 2

• A priority number (integer) is associated with each process

• The CPU is allocated to the process with the highest priority (smallest integer ≡ highest priority)

• Preemptive

• Non-preemptive

• SJF is priority scheduling where priority is the inverse of predicted next CPU burst time

• Problem ≡ Starvation – low priority processes may never execute

• Solution ≡ Aging – as time progresses increase the priority of the process

•Time quantum

oCPU bound vs. I/O bound

•Multilevel Queues and Multilevel Feedback Queues

• Ready queue is partitioned into separate queues, e.g., • foreground (interactive) • background (batch)

• Process permanently in a given queue

• Each queue has its own scheduling algorithm: • foreground – RR • background – FCFS

• Scheduling must be done between the queues: • Fixed priority scheduling; (i.e., serve all from foreground then from background). Possibility of starvation. • Time slice – each queue gets a certain amount of CPU time which it can schedule amongst its processes; i.e., 80% to foreground in RR • 20% to background in FCFS

Multilevel Feedback Queue

• A process can move between the various queues; aging can be implemented this way

• Multilevel-feedback-queue scheduler defined by the following parameters:

• number of queues

• scheduling algorithms for each queue

• method used to determine when to upgrade a process

• method used to determine when to demote a process

• method used to determine which queue a process will enter when that process needs service

•Linux O(1) and CFS scheduler

• Prior to kernel version 2.5, ran variation of standard UNIX scheduling algorithm

• Version 2.5 moved to constant order O(1) scheduling time

• Preemptive, priority based

• Two priority ranges: time-sharing and real-time

• Real-time range from 0 to 99 and nice value from 100 to 140

• Map into global priority with numerically lower values indicating higher priority

• Higher priority gets larger q

• Task run-able as long as time left in time slice (active)

• If no time left (expired), not run-able until all other tasks use their slices

• All run-able tasks tracked in per-CPU runqueue data structure

• Two priority arrays (active, expired)

• Tasks indexed by priority

• When no more active, arrays are exchanged

• Worked well, but poor response times for interactive processes

• Completely Fair Scheduler (CFS)

• Scheduling classes

• Each has specific priority

• Scheduler picks highest priority task in highest scheduling class

• Rather than quantum based on fixed time allotments, based on proportion of CPU time

• 2 scheduling classes included, others can be added

1. default 2. real-time

• Quantum calculated based on nice value from -20 to +19

• Lower value is higher priority

• Calculates target latency – interval of time during which task should run at least once

• Target latency can increase if say number of active tasks increases

• CFS scheduler maintains per task virtual run time in variable vruntime

• Associated with decay factor based on priority of task – lower priority is higher decay rate

• Normal default priority yields virtual run time = actual run time

• To decide next task to run, scheduler picks task with lowest virtual run time

Extra Topics from the Chapters:

Trap/Execution: Trap is a type of synchronous interrupt. Caused by an exceptional condition and the trap switches to kernel mode.