crowdsourced guide

// disclaimer: currently studying for cs162 midterm and making a study guide for myself, thought I'd share. So far this is very incomplete/ will update as I write more//

** Chapter 2 The Kernel Abstraction **

Section 3 Types of mode transfer

User to Kernel mode: interrupts, system calls, processor exceptions.

Kernel to User mode: new process, resume after an interrupt, processor exception, system call; switch to different process, user-level upcall.

Section 4 Implementing safe mode transfer

At minimum, the common sequence must provide: limited entry points into kernel; atomic changes to processor state (mode, stack, program counter, and memory protection are all changed at the same time); transparent, restartable execution

Section 5 Putting it all together: x86 mode transfer

Mask interrupts=hardware,

Save three key values (stack pointer - esp, execution flags eflags, instruction pointer eip)=hardware,

Swtich onto the kernel interrupt stack=hardware,

Push the three key values onto the stack=hardware,

Optionally save the error code=hardware,

Invoke the interrupt handler=hardware -> handler software

Section 6 Implementing secure system calls

Locate system call arguments, validate parameters, copy before check, copy back results

Section 7 Starting a new process

Allocate memory for PCB, allocate memory for the process itself, copy the program text into memory allocated for it, initialize stack (user-level), initialize stack(kernel-level: for syscalls, exceptions, and interrupts)

To start running: copy arguments into user memory, transfer control to user mode.

Section 8 Implementing upcalls

We call virtualized exceptions and interrupts upcalls.

In UNIX: signals, in Windows: asynchronous events.

Several uses for upcalls: pre-emptive user-level threads, asynchronous I/O notification, interprocess communication, user-level exception handling, user-level resource allocation.

Similarities with hardware interrupts: types of signals, signal handling, signal stack, signal masking, processor state

Section 9 Case study: booting an operating system call

Boot ROM (read-only memory which stores boot instructions)

the boot program is called BIOS (basic input-output system)

In physical memory: BIOS, boot loader instructions and data, OS kernel instructions and data, login app instructions and data

Section 10 Case study: Virtual machines

host operating system the operating system providing the virtual machine abstraction

guest operating system the operating system running inside the virtual machine

When the host kernel starts the virtual machine, the guest kernel acts as though it is being booted

Section 11 Summary and future directions

Hardware mechanisms for operating systems:

• privilege levels,

• privileged instructions

• memory translation

• processor exceptions

• timer interrupts

• device interrupts

• interprocessor interrupts

• interrupts masking

• system calls

• return from interrupt

• boot ROM

Chapter 3 The Programming Interface

Section 4 Case Study: interprocess communication

• Producer-consumer

• Client-server

• File System

Chapter 4 Concurrency and Threads:

Section 4 Implementing Kernel Threads

Thread Context Switch

saves current running thread’s registers to the TCB and stack, then loads the second thread’s registers

• Voluntary: thread_yield/thread_join-thread_exit suspend execution and start new thread (disable interrupts while switching)

• Involuntary: interrupt of processor exception could invoke and involuntary context switch. Hardware saves current register state and executes the handler’s code.

Chapter 6 Multiobject Synchronization:

Section 5 Deadlock

Deadlock is a cycle of waiting among a set of threads, where each threads waits for some other thread in the cycle to take some action. Examples: mutually recursive locking, nested waiting;

Conditions (preventing any of the following will eliminate the deadlock) :

• Bounded resources

• No preemption

• Wait while holding

• Circular waiting

Banker’s algorithm for avoiding deadlock: a thread states its maximum resource requirements when it begins a task; but it then acquires and releases resources incrementally as the task runs. The runtime system delays granting some requests to ensure that the system never deadlocks.

• In a safe state, for any possible sequence of resource requests, there is at least one safe sequence of processing the requests that eventually succeeds

• In an unsafe state there is at least one sequence of future resource requests that least to deadlock no matter what processing order is tried

• In a deadlocked state, the system has at least one deadlock

We can realize this idea by tracking: the current allocation of each resource to each thread; the maximum allocation possible for each thread, and the current set of available, unallocated resources.

Chapter 7 Scheduling:

Section 1 Uniprocessor scheduling

First in First out(FIFO):

simplest scheduling algorithm; drawback - bad average waiting time

Shortest Job First(SJF):

shortest (remaining time). Optimizes average waiting time; however is impossible to implement and is not ‘fair’. Trade-off between average response time vs. variance in response time (SJF maximizes the variance, minimizes average response time)

Round Robin:

Compromise between FIFO and SJF. Each process runs for a time-quantum of time, after which switches to the next one. Yields really bad average response time

Max-Min fairness:

max-min fairness iteratively maximizes the minimum time allocated to a single process until all resources are assigned

Multi-level feedback Queue(MFQ)

• Responsiveness

• Low overhead

• Starvation-freedom

• Background tasks

• Fairness

Basic idea: priority levels (higher priority thread will always execute before lower-priority thread); time quanta (increasing with lowering priority level); adjust priority: if runs out of time -> dump a level down, if requests I/O -> bump level up. To ensure min/max a thread with achieved goal of min/max values doesn’t run until every thread with not achived min/max values has run.

Chapter 8 Address Translation:

Address translation is conversion from the address a process thinks it is referencing to the physical location of that memory cell.

• Process isolation

• Interprocess communication

• Shared code segments

• Program initialization

• Efficient dynamic memory allocation

• Cache management

• Program debugging

• Efficient I/O

• Memory Mapped Files

• Virtual memory

• Checkpointing and restart

• Persistent data structures

• Process migration

• Information flow control

• Distributed shared memory

Section 1 Address translation concept:

Goals: memory protection, memory sharing, flexible memory placement, sparse addresses, runtime lookup efficiency, compact translation tables, portability.

Section 2 Towards flexible address translation

Segmented memory - isntead of using physically contiguous chunks of memory, seemingly continuous memory of the process is in fact mapped to segments across various adresses. Segmentation fault occurs from trying to access an uninitialized memory segment; or one not belonging to the current process.

Segmented memory, paged memory, multilevel translation: paged segmentation, multilevel paging, multi-level paged segmentation;

Chapter 9 Caching and Virtual Memory:

Section 5 Replacement policies

• Random

• FIFO (can be worst possible when repeatedly accessing a large object which doesnt fit in cache as a whole. Forces cache misses which could have been avoided)

• MIN Optimal Cache Replacement (whichever block is used farthest in the future, thus unattainable)

• LRU (Least Recently Used)

• LFU (Least Frequently Used)

Belady’s anomaly: Adding space to cache memory can hurt the cache hit rate. (FIFO)