SOFTENG370 Notes 2017

Theodore Oswandi

August 25, 2017

Contents

1 Lecture 1

1.1 Generics

Operating System The software that makes the computer usable. Using modern computers without an OS is "impossible"

Examples: Windows, OSX, Linux, Unix, iOS, Android, etc...

1.2 Approaches to Understanding

Minimalist

- mostly going to be using this one
- OS contains minimum amount of software to function
- archlike

Maximalist

- All software comes with standard OS release.
- Contains many utilities and programs.
- ubuntuish

1.3 Usable vs Efficient

- make sure you make OS suited for needs
- either specialised or more general purpose
- Think of who you expect to use the system
- If creating a realtime system with potentially thousands of operations in a short amount of time, have to consider efficiency
- Same with battery life if you expect the system to be used in a mobile setting.

1.4 OS themes

Manager Model

- OS is colleciton of managers, ensuring proper use of devices.
- Managers are independent.
- look out for everything associated with computer
- tie in with hardware. Current state of HW lets OS do more/less things

Onion Model

- Onions have layers (Abstractions)
- resources contained in lower layers.
- Lower layers can't access higher level layers but other way around possible
- Very difficult to get these layers 'right'
- can use in terms of security. Very good idea

Resource Allocator Model

- similar to manager model
- emphasis on fairness and providing services

Dustbin Model

- contains middleware that not considered part of OS
- Sees OS as bits no-one wants to do

Getting Work Done Model

- Idea of it is we use computers to do something else.
- Goal for OS is to help be able to get it all done.

1.5 OS design

1.5.1 Themes

All in one

- All OS components freely interact with each other
- MS-DOR and Early Linux

Separate Layers (Onion Model)

- Simplify verificiation and debugging
- Correct design difficult to get

Modules

- All in one with modules for some features
- Linux and Windows.

Microkernels

- Client/Server model
- make OS as small as possible
- **Exokernel** puts kernel outside. OS's job only need to authenticate people to use hardware.

VMs

- Java is an example of this

1.5.2 MS-DOS

- Written to provide the most functionality in the least amount of space
- not divided into modules
- Something exokernels trying to do. Make application program access hardware directly.

1.5.3 Early Unix

- UNIX OS in 2 parts. **Kernel** and **System Programs**
- Provides:
  - File System
  - CPU sheduling
  - Memory management
  - Other OS functions
- Ken Thompson and Dennis Ritchie
- Make OS as simple as possible.
- Simple 2 letter commands.
- Ideas of pipelining and process communication

1.5.4 THE Multiprogramming System

- THE was the first to use the layered system
- Contains 6 layers:

  5 User programs
  4 Input/Output buffering
  3 Operator-Console device driver
  2 Memory Management
  1 CPU scheduling
  0 Hardware

1.5.5 WinNT and Client/Server

- WinNT still being still run
  - Win10 now has Windows Subsystem for Linux
- NT provide env subsystem to run code written for differnt OS
- NT and successors are hybrid systems. Parts are layered but some merged to improve performance.

2 Lecture 2: History of OS

- Started at mainframes.
  - Early PDAs were similar to mainframes. Had no memory protection.
- Then go to Minicomputers
- And then desktop
- And how handheld computers

Each of these stages go through cycle of:

1. No software
2. Compulers
3. Multiuser
4. Networked
5. Clustered
6. Distributed Systems
7. Multiprocessor & Fault tolerant.

2.1 Total Control

- Computers expensive in 50s. Data and programs were saved on paper tape.
- Programmers knew how the computer worked. They were very knowledgable about computers.
  - Prepared program and data cards
  - do setup
  - control computer
  - debug
- Computers did 10,000s instructions per second, but were idle a lot of the time.

2.2 Properties of old OS

- **IO polling**, since no other programs running in background, therefore just waiting on input and able to just poll.
- No file system
- No memory management or security
- OS defined by decisions made by user.
- Single program at a time

2.3 Progression: Operators & Offlining

Operators

- Goal is to reduce the time CPU was doing nothing.
- Operators now just "use" the computer. No need for programmer.
  - If something crashes, then just start the next program.
  - Batch similar jobs together, maximise usage of computer.

Offlining

- Form of parallelism in early computing.
- With Big Expensive Computer BEC, but they are just waiting for IO a lot of time. Therefore want to make IO as fast as possible.
- Use smaller computers to convert slower paper to faster magnetic tape. Then that magnetic tape is used as IO for the BEC
- This is the same for output. Have another smaller cheaper computer offload the output magnetic tape from BEC to a printer.

Resident Monitor

- Keep some code in memory.
- It did the work that some operators were doing.
  - clearing memory
  - reading start of new program that needs to be loaded.
  - Can also do some of the IO routines.

Control Programs Standardise the language to communicate with the Resident Monitor. Had tags for things such as $JOB (for signifying jobs), $FTN (When fortran compiler needed), and $END (signifying end of program)

Conclusions from this

- Memory management and file system still not present. Therefore still need to reset if anything bad happens.
- Security patchy at best
- Still need IO polling
- Standard IO routines for programmers
- 2 programs in memory, but one executed
- User interface was JCL (Job Control Language)
- Output of program can be input of another.

2.4 Changes in Hardware

- **Disk drives** provide faster IO.
- Processors that you can **interrupt** also means that there is no more reliance on polling.
- IO devices and CPU concurrent execution, and use local buffer.

SPOOLING (Simultaneous Peripheral Operation On-Line) Meaning that when interrupt, contents of cards read to disk. Therefore current program interrupted.

2.5 Multiprogramming

- Putting multiple programs on at once. Need more memory to do this.
- Now also need for scheduler to manage multiple users' program needs.
  - Need to figure out how to manage stuff. Priority of jobs, how much time to allocate for these jobs, etc...

- No memory protection, so programs could overwrite other program's chunk of memory.
  - Java is an example of somehting that doesn't give you direct access to memory in JVM.
  - Memory Protection better done by hardware than having software impose limits.
- **Requirements** Limited address range and Operating modes.

Memory Protection Modes

1. User/Restricted Mode
   - Execution is done on behalf of the user.
   - User should not have access to privileged instructions
2. Kernel Mode (SU)
   - Execution done on behalf of the operating system
   - Full access to all instructions.

A **mode bit** can be used to signify what mode a certain program is running in. If something in user mode tries to access memory it is not allocated to, it will go to Kernel mode and throw exception before going back to User mode.

Why we need both We need both because:

If modes existed with relevent instructions, but full memory access; there will still be a lack of memory protection, but also no privilege instruction protection. You can put whatever code you want anywhere.

If memory access limited but no modes or privilege access; then the user will be able to modify amount of memory available for programs.

Memory Protection

- Process gets fixed area of memory that it can use
- If tries to access address out of that range then exception will be thrown.
- Base and Limit register set for each process and how much memory it can have.

2.6 Batch Systems

Memory protection and Processor modes allow you to safely put multiple programs in memory.

Features

- Jobs have their own protected memory
- Disks have file systems. Files linked to owners
- Automated Scheduling. Utilise hardware as much as possible, as operators are slow. Also allows fine tuning of how scheduler works.
- Computer consoles

Not much has changed from programmer's point of view.

3 Lecture 3: History Continued

3.1 Scheduling

- Ams to maximise use of computing machinery OS knows
- Need to know details about device and file processes. What how much resources to allocate.
- Also has to take into account timing and output size.

SOMETHING ABOUT UNIVERSITY OF AUCKLAND SYSTEM

3.2 Power to the people

- Due to hardware becoming cheaper, can have general public own personal computers
- Used to use teletypewriters, but used CRT TVs after a certain point. Editing text was difficult.
- At early 1970s, can code in similar style then you do now.

3.3 Time Sharing System

- People don't like waiting.
  - 200ms+ noticable
  - 5000ms+ unacceptible
- Difficult for scheduler to figure out how to allocate resources. People use different computer differently with differing IO demands.
- Users expect command to run as soon as you press Enter.
- Don't want to have everything run at 100%, otherwise it feels too slow.
- Security an issue for all of these people writing on terminal. Have to increase this and have authentication.

Remnants of Batch Programming

- Has way to run process at given time
- Terminal looked like cards until better graphics came

3.4 1980s computers

- Cycle starts again, started with Resident Monitor Systems.
- Simple single layer file systems
- No security, everything stored on disks. Didn't bother as it was aimed at individual users.
- Did spooling later, for printer output.
- Putting more than one program in memory, using similar system to resident monitor.
- Higher definition screens, pixel addressing for graphics.
- Cycle continues, things like time-sharing features and implementation of UNIX.

Xerox created GUI elements for Office use. Then Apple engineers used ideas to create their Mac.

Features

1. Virtual memory
2. Multiprogramming
3. Complex file system
4. Networking
5. Multi-user

3.5 1980s Networking

Security, Transparency and Protocols/Standardisation create new problems.
Network OS: File sharing, communication scheme, running independent to other machines on network.
Distributed OS: Sharing processing power and resources of lots of computers to make it look like only single system.

3.6 Multiprocessor Systems

Heat is an issue, kind of a soft cap on processor frequency. Therefore can add more cores instead of trying to make each core faster.

Tightly Coupled System Processors sharing memory and clock. Communication through this shared memory. Most computers are now this.

Parallel Systems Mean increased throughput and cheaper way to increase performance. WIth increased reliability and rate of degradation.

Symmetric Multiprocessing: All core running same OS, most modern systems run this way

Asymmetric Multiprocess Different cores allocated to different jobs/section. Used in very large systems.

3.7 Realtime System

Timing constraints very important.

Hard real-time

- must run within time, or failure happens
- Has to be specifically designed to be hard realtime
- Nuclear plants, air traffic control

Soft real-time

- Doesn't matter too much, more lax.
- Most OS handle soft realtime
- Phone system, multimedia

3.8 Pocket Computer & Smartphones

- Started as PDA/Pocket computers.
- Went through cycle again. Started as resident monitors.
  - Due to hardware limitations, so have to start at the basic level again.
- Battery life and power consumption very important factors.

PalmOS Operating system that PalmPDAs ran on.
Small memory with slow processor.
Efficiency very important factor, to just get passable performance.

Android Popular operating system for current smartphones.
Linux based, application programming in Java.
Google trying to build their own kernel to replace Linux (Fuchsia)

iOS Operating systems that mobile Apple products run on.
Based on OSX (Their desktop OS)
virtual memory and paging for code but not data as writing to flash degrades it.

4 Assignment Notes

- Use standard UNIX symbols to control the threads
- setupstacktransfer()
  - **siguser1** represents the user's signal. Let you send stuff to yourself similar to interrupt, but done in software and not hardware
  - **sigaction** is a struct that holds information. Kind of like an object is global due to process having to be able to get to it at any time

  - Has a separate, special stack for that singal handler to use.
  - **man pages** are really important for this assignment.
  - If want to get all man pages relevant to signal then use $man - ksignal$

- Threads need their own stack

  Running independedntly of each other and calling their own functions so to guarantee proper functionng it is best for them to have their own stack
- **&setupaction** address of instructions for the signal handler
- **thread1()** contains code that will be executed in the thread
- threadfuncts is array of names of functions that should be called for all threads If add more then you need to add to the array
- In task 2; 3 threads but 2 of them running the same logic from thread2()
- Information about thread structure found in littleThread.h
- static variables aren't allocated on the stack. And preserve value throughout multiple function calls
- MISSED UP TO LIKE 35min in
- **sigaltstack** lets you use that special alternate stack for different threads

  have malloc some memory and will use it

  When you call associateStack() when making new process you make a new alternate stack
- **kill(getpid(), SIGUSR1);**

  KILL is system call to get signals. Set it up but haven't associated it with anything it yet

  KILL sending pid of process you want to send it to. Send signal to yourself (try to kill yourself).
- make local copy of thread in fuction and set it to READY.
- C doesn't have exception handling. Therefore if error happens in a stack then need the ability to jump to part of memory to give error.
  - **setjmp**: Take snapshot of where you are. Registers of processor (PC will contain this). Can also be used to "freeze" state of a given thread if need to be suspended.
  - **longjmp**: Jump back to state where setjmp called. Can be used to "unfreeze" an already suspended thread to resume it. [Line34 in OSA.c]
  - Copy stack information/register information and when jump back then recopy it back to "jump back to where you were"
  - variable states preserved if stored on the stack
  - if setjmp return 0 then returned directly, or nonzero if from longjmp. Will be used later for forking to create new processes, to check if from parent or child
- **Switches** Pass it your current thread and the thread you want to go to.
- Only one thread running at a time, other ones will be READY due to only using a single processor.
- can get this assignment to work without understanding it

5 Lecture 4: Virtual Machines

./ used to signify that it isn't an internal command

MISSED TO VIRTUALISATION

Virtualisation if running on hardware then want to be as close to 90% performance as possible. Preferably 95-98 but not always possible

Design of IBM vm make each user feel like they have own cpu minidisk = lets user feel like they have access to whole drive problem is you don't want actual kernel mode to be accessable to all guests solution is each user has their own virtual kernel mode, but this kernel mode actually runs on the user level. Priviledged instructions actually needed to be passed down as not all things kernel does need that mode

Hypervisor Types Allocating resources to VM - like actual CPU cores - or chunks of memory allocated for it Can have "nested" vms

Type 1 Special purpose OS have support for bunch of tools to make using it easier

Type 2 Ones that you install yourself. (virtualbox, parallels) Run applications on host

Problems trap and emulate couldn't be run on x86 up to a point.

Hardware virtualisation x86 Most OS only use level 0 (kernel mode) and level 3 (user mode) Problem with VM in real machine, then you need to keep track about process and registers. Have to keep track of this for all processes. Hardware system lets you change processor for one VM to another

each VM page tables for their own processors used to have nested page table system. VMs create their own virtual page tables and some will exist in real memory

Solutions Binary translation Look at instructions before execution, problem instructions get translated to be safer to be run in kernel mode

These translations are similar Only translated code is run

OS level virutalisation If lots of machines running same OS, then can use containers that make it seem like they are all separate. Useful for servers Simpler than VMs as they are sharing same copy of OS

More Styles paravirtualisation - XEN modify source code of OS you want to run increase efficiency to allow calls to be made straight to VMM instead of process

Application Virtualisation WINE Want to run something made for an OS on another OS Makes the application think like its running on intended OS

Windows Subsystem for Linux Not really virtual machines If app makes linux kernel call, kernel figures it out and sends it to subsystem Tied into kernel level, applicaiton doesn't really know about it. It just functions as normal and kernel does all of the work.

C and OS implementations

Week 2 friday MISSED 10 MINUTES OF LECTURE

Direct access to memory: address.c Whenever you run the program, the stack address space is different This is for security ASLR address space layout randomisation. Stack, heap and libraries put in different addresses. Helps add level of security

Accessing Registers Can choose to store something in a register Use keyword 'register' prefacing variable type on initalise Can't get address of register, so if set to register OS may put it out of register into memory if you try get address of variable

Volatile Another keyword prefacing variable type Don't do any clever tricks When you don't know if variable value will have changed due to non-local reason due to things like interrupt.

Whenever you use this variable, you have to go back to memory and check its value again as it may have changed.

Memory Management No memory management Static memory allocated at runtime, no malloc. But hard to get rid of them

Dynamic memory Garbage collection doesn't inherently exist. As it is unpredictable

Allocating stack space can be done by calling 'free' Free knows how much memory to free up since malloc uses a little bit more space just above for length of bit of stuff stored

Inline assembly Example code is 32bit OS dependant Can put assembly language directly in C code

Running commands from C program system() lets you put string of command you want to use

Alternatives (languages for OS) C++ similar to C but with object stuff too Windows has C kernel with some C++ and C# Objective C MacOS written with ObjC, but trying to move to Swift Java Can't exclusively use java, need stuff with other stuff as well Assembly old school if you need even more fine tuning

More assignment stuff Part 1 create a lot of threads, link them together (linked list) circular linked list (doubly) keep going around cycle of threads until all finished executing. Only 2 threads given, but should be able to do with n threads Part 2 Add thread.yield() This will call transition system like in part 1. Stop current thread (not finished) and pausing itself to allow another thread to run. Part 3 Interrupt the thread with external source use set.itimer, send signal to processor to signify event happening. (timer has run out, every 20ms) Tells current thread to pause externally and start next thread.

Processes Instance of program execution Thing OS uses as construct to control work

Two parts Resources/Task/Job files open and using windows on screen restrictions on process Code that's running what process is actually doing these days have threads for multiple streams of instructions

Thread sequence of instructions executing without interruption this does happen, but not from thread's point of view. Thread can't tell if it has been paused or not Can run multiple threads but share resources

Typical uses split work accroess processors/cores thread for user response, another for some computation task GUI threads and process threads Server applications, have threads for clients. Server preallocates set of threads for handling requests

Thread implementations user level OS sees one thread per process

advantages work if os doesn't support threads easier to make, no system calls application specific control switching is easier (some have register files for threads)

System level operating system knows about it controlled by system calls System knows about state of thread as well. Therefore will schedule based on their state

advantages Threads treated separately If multiprocessor, then can schedule different threads on differnt processors thread blocking in kernel doesn't stop all thread on same process for example if doing read on file, usually code will wait for result and therefore block can allocate cpu to do something in the meantime in this case

Jacketing Check "will I block" before doing something that may block check to see if data already exist in memory. If already there cna just get it without having to block. if have to get it, then let processor do something else while it tries to get data

Best of Both worlds Solaris had both system and user level threads before ver9 Uses one to one mapping of user level to kernel level threads. Mapping of single lightweight process to kernel threads. lightweight processes If something on user level thread makes blocking call, other threads on that lightweight process gets to do its thing system makes its own kernel thread and new lightweight process to allow this to happen Windows 7 threads Since Win7 then have user mode scheduling. This also tries to get the best of both world Linux threads Used to not have threads, everything put on one thread Clone call makes a new process. Shares memory, open files and signal handlers Saw them as processes and not threads, so scheduled them Can't signal whole thread, therefore since cloned you aren't sending it to all of them and only the one you specify Killing threads dangerous, due to them sharing memory, then if killed then blocking may cause memory to be in inconsistent state as lock has not been released yet In POSIX, don't actually kill threads. You tell it to cancel itself instead, telling it to die at some point Threads are written in such as way that before it makes a blocking system call, it does some tidying so cancellations can happen Cloned threads can't block if other clone made blocking system call

Week 3 Wednesday More on threads and processes Part 3 assignment numthreads constant will be correct can initialise arrays with that size if you want

Lecture 6 Process Control Blocks Things os should know about process BIG LIST GOES HERE process state turns out to be thread state priority used by scheduler owner - security considerations process generally on one processor (306 core moving cost) process group - proceessors working together memory and resource considerations see if process result can be piped to another process, or that it is waiting for result of this process

UNIX process parts can be scattered as parts somewhere else process structure some of information of process held here

user structure not instant access to this in user space some of information of process held here

In UNIX, text = code

WIndows NT split it up into lots of things in ANOTHER BIG LIST TO COPY YAY

MISSED SLIDE 3 TO END OF LECTURE EMPHASIS ON FORKS

6 Lecture 7

6.1 Runnable

- On one core, only one thread/process at the same time. (Exception SMT)

- Other processes/threads may be ready to run, or already running

6.2 Multitasking

Pre-emptive Multitasking

- OS uses some kind of criteria to determine how large of a time slice that task
- The more you call yield and switch processes, the more time is wasted and less actual work is done by CPU

  Cognitive multitasking Threads know that may have thread.yield() called on it and therefore are coded in a way such that when yield() is called, issues are less likely to occur
- Advantages
  - Control
  - Predictability

  Disadvantages
  - Critical Sessions
  - Efficiency

Co-operative Multitasking

- Two main ways to approach
  1. Process yields right to run
  2. System stop process when system call made
- Doesn't mean task won't run and complete in one go.
- Old UNIX (before 2.6) didn't allow pre-emptive calls when making system calls
  - pre-emptive multitasking always at user level
  - hasn't always been preempntive at system level
  - Actually used to be cooperative in the past
  - Unix was written simpler in the past, expected it to be simple with blocking calls made.

6.3 Context Switch

- Change from one process running to another on same processor, or to handle an interrupt
- Has to save the process state before this can occur
- Context changes as process executes
- Context contains:
  1. Registers
  2. Memory (dynamic elements like call stack)
  3. Files & Resources
  4. Caches

6.4 Returning to Running

State Transition

- Store process properties so it can begin again where it left off
- Page table to be updated if changing processes
- Environment must be restored
- If changing threads on same process then may can just restore registers
- If system has multiple register sets then could thread change with 1 instruction

6.5 OTHER STATES

6.5.1 Waiting

Waiting To stop unnecessary resource consumption Status changed from runnng to waiting
Suspended Different form of waiting

Java Always had Threads from the start Threads have generally been user level Although Thread.suspend existed that froze thread on system level Thread.resume() to restore it Issue was some resources are tied to one process, and therefore gets a lock Therefore if frozen then other threads can't access it Threads.stop() kill thread and force it to release locks that it may have But may cause data to be left in inconsistent state

Waiting in UNIX WCHAN can contain numbers, represents address in kernel Uses a queue to create a queue for processing Queue associated with hash value or kernel addresss

HERE GOES SOME PROCESS OF HOW IT ALL WORKS

Finishing Resouces used by process need to be accounted for Shared resources usage lowers due to process finishing Make sure tidying up is done, if not done already Don't rely on this, should do this yourself

"When you log out, you want all your processes to finish too" Create a cascading effect, one process shutting down causes other ones associated with it to shut down too

Reasons to Stop Normal Stop must call exit routine does all required tidyup

Forced Stop Only want some processes to be able to kill specific processes. Parents can kill children Children can "generally" kill parents since same owner

UNIX stopping Has 'zombie states' Process that is finished, until parent checks exit status This is a return state/value of a process Used so next process/processes can find out how child finishes and continue execution based on this result If parent is around and child finishes, child becomes a zombie

If parent never calls wait if parent finishes then zombie is freed

Another FSM

Info from Linux Process Table NI = nice value can be positive a negative used to change priorities the lower the number, the higher the priority negative numbers are super priority Only SU can change nice values Normal users can only change nice values to positive values RSS = resident set size memory allocated to it TT = teletype TIME = how long process has been running for CMD = actual command that was executed

7 Lecture 8

7.1 Scheduling Processes/Threads

- **CPU burst time**: time takes for thread running to have to wait for some reason
- Basically, the majority of threads stop after processing for not very long time
- Therefore if we stop them frequently it doesn't make too much of a difference as they are probably waiting anyway

7.2 Levels of Scheduling

Batch Systems

1. Very long term scheduler
    - outside OS, more admin level
    - STUFF
2. Long term scheduler
    - Have multiple queues
    - STUFF
3. medium term scheduler
    - Still programmer dependent how its done
    - STUFF

4. short term scheduler
    - Will mainly look at this one
5. Dispatcher
    - Does the switching from thread/process to another

7.3 Scheduling Algorithms

7.3.1 FCFS - First Come First Served

- No wasting time by determining how to allocate
- Use average waiting time and the CPU burst times for processes
- Produces Gantt chart looking thing
- Weight times are when the processes start

7.3.2 Round Robin

- Pre-emptive version of FCFS
    - Still don't let them run to completetion
    - Use of pre-empting them and time slices
- Hard to determine what size time slices to allocate
- Some processes are CPU intensive and require longer time slice
    - But if let these processes do its thing, user may feel slowdown.
    - Interactive processes affected by this
- If short time slice then good in terms of interaction as its jumps around to lots of processes.
    - However, CPU intensive tasks take longer to complete
- Still doesn't have concept of priority
- If task takes shorter time than time slice, instantly schedule another task as to not waste CPU cycles.
- Average wait time reduced due to forced time slices
- Making time slices smaller reduces the average wait time

7.3.3 Minimising Average Wait Time

- Need to know how long CPU bursts are

Shortest Job First

- Gets minimum average waiting time
- But don't always know all CPU burst times
- Therefore use an estimation algorithm. Basing it off previous CPU bursts to estimate how long subsequent bursts will approximately be.

Pre-emnptive SJF

- Uses arrival time and burst time
- Short it not because of CLK interrupt, but because another processor came in with a shorter CPU burst time.
- Use remaining CPU burst time remaining if trying to determine if you are going to stop and schedule another process
    - If a process has CPUburst = 7
    - Something with CPUburst=4 comes at time=2
    - Compare 5 (7-2) with incoming CPUburst=4
    - Therefore will stop original processor and run new one since $5 > 4q$
- If has 2 options with same weight, then up to programmer to choose. Theoretically similar, but in reality will have some weighting choosing one over the other

7.4 Handling Priorities

Explicit Priorities

- If have very low priority, then there is chance that some priorities will never actually run **Starvation**
- SOMETHING GOES HERE

Variable Priorities

- Processes get higher priorities the longer they've existsed (aging)
- Solves the starvation problem

7.5 Multiple Queues

- Multiple queues exist for things that require different time slices and CPU cycles
- Kind of a heirarchy of these processors
- Still assumes single processor

7.6 UNIX processor Scheduling

- Every process has priority associated with it
- Priorities are recalculated every second
- Larger number means worse priority. Lower numbers go first
- Can **Nice** a process, adds priority for process (nicer to everybody else)
  - Ordinary users can only nice their own processes, thereby delaying their processing
- Aging exists, priorities get worse the longer they run
  - Worst level exists, so this doesn't continue forever
  - For every process at worst level, are scheduled in round robin
- The longer a process spends waiting, the lower its priority level becomes and therefore higher chance of being executed

7.7 Old Linux Process Scheduling

- Used two process scheduling algorithms
  1. Time sharing algorithm for most processes
  2. Realtime algorithm for absolute priorities hold over fairness
- Processes have different scheduling classes that determine which algorithm to apply
- Uses **prioritised credit based algorithm** for time sharing
  - Process with most credits go first
  - If process running on clock tick, it loses a credit
  - If process hits 0 credits then another process chosen
  - Therefore the more you wait the more credits you get

7.8 Linux Real-time Scheduling

- Linux uses both **FIFO**)First in first out) and **Round-Robin** scheduling.
- In both situations, processes have priority + scheduling class
- Scheduler does process with most priority
  - If equal priority then choose one that has been waiting the longest
  - FIFO processes run until exit or blocked, no pre-empting
- In Round-robin, processes pre-empted after a while and moved to end of queue.
  - Allows

New Linux Processing

8 Lecture 9

MISSED SLIDE 1 & 2

Periodic process

- common that period and deadline are the same
- Deadlines and period may change depending on the workload of the system

Sporadic Processes aperiodic process things can happen at the same if ∞ evens can occur at the same time then need to figure out how to allocate it

Cycling Executives Handle periodic processes Prescheduled - know information before power machine, so can schedule Can't pre-empt because schedule already generated Hard to maintain

CE Schedule MAJOR SCHEDULE MINOR CYCLE

8.1 Scheduling with Priorities

Lets you do important tasks first CATCH UP

8.2 Priority Allocation

Fixed

- **Rate monotomics RM,** shorter period means higher priority
- Least compute time LCT, similar to Shortest Job First

Dynamic

- CPU burst times used/useful
- **Shortest completion time**
  - Simnilar to SJF
  - Uses pre-emption, but requires good information about execution time requirement.
  - Schedule, and compare the time required to finish computation of process at every cycle
- **Earliest Deadline**, process with closest deadline goes first
  - Does this every cycle, and compares all processes that want to be sceduled and their respective deadlines
  - Add don't cares/idle times for when process is complete before deadline. Counts as ∞, allowing another process to run
- **Least Slack Time** (deadline - compute time) gets highest priority
  - If no slack time left, then must schedule now.
  - Slack time doesn't change if it gets process. Due to fact that its deadline gets closer, but its computation has gone for another cycle, cancelling each other out.

8.3 Theory

- Static priorities, RM is optimal policy
- Dynamic priorities, EDF(Earliest Deadline) and LST(Least Slack) are optimal
- Only really works for single processors.
  - Required more sophisticated processes, to allocate multiple processors

9 Lecture 10

9.1 Prolbem of Concurrency

- Sharing resources is a problem
- Multiple threads/processes trying to access it at the same time

- Some resouces can only be safely accessed by a single thread at a time
  - Reading from resources that are being written to.
  - Writing to a file simultaneously
- **Race Condition**: Where order of thread execution produces different results

9.1.1 Example of contention

- Don't have control over thread execution once threads have started executing
- have to `-lpthread` on linux when compiling C programs to use thread library
- `counter++` seems innocent, but actually has a window for error.
  - Due to concurrency, there is contention if multiple threads are the one that calls the function that causes a reaturn
  - Did not count to 10 a lot of the time due to contention happening a lot of times

9.2 Critical Sections

- **Mutual Exclusion:** Area of code that it expects only 1 thread active at given time
- Need to lock thread when critical session going on
- Need to have a way to make sure threads aren't waiting forever (Starvation)
- Starvation can be caused by deadlocks of indefinite postponement

9.3 Software Solutions

- Both can get lock at same time if multithreaded
- Thing dying prematurely
  - OS needs to keep track of when things are alive/dead
  - Always an issue and should have something in place to deal with it, if it were to happen
- Polling puts unnecessary strain on system
- **Spin lock** - something waitng just doing nothing.
- **Busy wait** - Waiting when processor doing something, which is something you don't want to happen
- Suspended thread after seeing that thread is unlocked may lock it at another time, in which another process has already got a lock

- Main problem is multiple threads that sees that `locked = false`, and trying to set it to `true`. Due to small vulnerable time gap

9.3.1 Peterson's Solution

- `flag = [false, false]` is a shared variable
- Getting lock
  - Set your own flag[self] to true, and set turn to another thread, and wait until they either they let you or finish doing process on object
  - Setting flag = true means that you want to access it.
- Wait when its other thread's turn and they wanna use the file.
- Not feasable in real life due to instruction re-ordering
- Instruction Re-ordering
  - Both compilers and processors do some optimisation at runtime/compile time.
  - Users don't have control over this

9.3.2 Bakery Algorithm

- Each thread given number indicating when it can request a lock.
- Numbers aren't unique so need to use another form of identifier to distinguish different processors with the same allocated number

9.3.3 Interrupt Priority Level

- Can be done by increasing interrupt priority level so other processes can't pre-empt
- Before you do a check on a lock, turn interrupt off so you can't be pre-empted.
- Only turn off interrupt for certain sections of code so that pre-empting will not be possible for that part only and not block other processes.

- Disadvantages:
  - Doesn't work efficiently with multiple processors present
  - A message must be sent to all other processors to let them know of the level change. May cause processors to wait
  - Not all processors at priority level need to be stopped so waste of resources

9.4 Using Hardware - Test and Set

Atomic, create instruction that cannot be divided and must run to completion So the locks that it obtains are guaranteed to not be interrupted. Testing the value, and setting it in one indivisible instruction.

```
while test_and_set(locked)
```

- This removes the gap that is prone to be pre-empted.

This doesn't solve the issue of a busy wait, nor does it make it fair.

Getting out of spin DIDN"T LISTEN YAY

Priority Inversion If low priority process has lock, as long as it can complete and pass the resource on Just need to make sure that the lower priority process can do its task. But sometimes can't due to interrupts and waiting and stuff Don't want lower priority to be stuck in ready state, while it still hasn't been scheduled.

Possible solution temporarily increase priority of something with low priority to higher priority only while it uses the resource. MORE THINGS

Placing in a queue Once new process realises that resource is in use, it suspends, and reschedules itself.

Putting process to sleep then the lock may be set to false so process waiting forever

Put another lock around the code to make sure that locking and unlocking can be done without interrupts

Busy waits solves this issue. OS puts busy waits on lots of little bits of code, but problem with busy waits is it may be waiting for a while.

Semaphores Solution to concurrency problem Basically counter with atomic operations 2 functions, V(S), and P(S). Original value, set to let n number of things NOT SURE AE

P() is kind of a lock If process sees S 0 or less then will wait V() is called when finish May set S ¿ 0 which may let another process use the resource

Implementing Semaphores

UP TO PRODUCER CONSUMER PROBLEM (14) Want consumer to block so that the buffer will have a value stored in it

10 Lecture 11

Reader/Writer problem Readers aren't a problem, as they don't change values As you add writers, then it makes it potentially inconsistent

Only 1 writer at at time in critical section, once it is gone then you can let readers go do their own thing. If we have multiple readers, we want them all in there at once with no writers

Writer prefered writer before reader prioritises reading of updated values read most recent data

Priority problem again, if writers keep coming then readers will wait forever

Reader Prefered Opposite problem to other one, writers may wait forever

both of these may end up indefinite postponemenet

No preference Use queue, neithe writer or reader has no priority

Getting program correct (2) had exclusive access semaphore, set to 1 so that only one thing can access it.

If producer goes twice then overwrites the value in the buffer of original, so no 1:1 mapping of producers to consumers

If consumers gets there before producer then the `number_deposited` is set to 0, producer sees this too and then waits forver and enters a deadlock

Bad Programmers

Have to make sure that if you have a lock in a section of your code, to put something in to unlock it too.

Can use things in OS to help programmers

Can possibly just call unlock if you see another process has a lock on an object and then be allowed to access it. Throws away any safety created if programmer has malicious intents

Monitors Object that allows at most one thread executing inside of it. Similar to old school kernel allowing one process to run in it

As long as you're running in the monitor then you don't have to worry about concurrency issues as the monitor only allows one thing to run at a time.

Slower due to monitor only doing one thing at a time, and therefore potentially getting a big backlog of tasks to do

Conditiion variables Queue you put thread on when it gets to front Uses the `wait` and `signal` again to only wake threads when needed

If you call wait, you always to go sleep. If consumer sees nothing in buffer then will call wait and sleep itself and wait for producer to be done

If you call signal and thing has changed keep going Otherwise then you do nothing. Used for situation where producers go after another to make sure buffer values are not overwritten.

Which thread runs? If you allow thread to call signal, to wake up another thread then be careful to make sure thta you don't interfere with its operation.

Java monitors Every object has a lock. This is to allow `synchronized` methods to work If you call a `synchronized` method then you're asking monitor associated with object if you can go in and do somehting. Has inherent `wait()` and `signal()` for each object.

What happens when there is a recursive call to `lock()` Throw error if encounters lock by `self` If continue then need to count how many times locked and unlocked. Make sure that you unlock it as many times as you locked it.

`pthreads` Can specify if you want a recursive lock or traditional lock. Tranditional lock will not allow same process to lock something twice.

They are different One condition variable for `signal & wiat` Not real monitors

11 Lecture 12

Dining Philosophers DESCRIPTION

First solution Does wait() on both left and right Eat Put them back

They all grab fork on the right and then can't get one on the left. Then die. REST OF THIS

Second Solution Basically the people either have both forks, or no forks Uses simultaneous wait and signal for getting and returning forks No deadlock but doesn't solve starvation problem

REST OF THIS

Simultaneous wait Uses a `try-lock` Pick up one and see if can grab other too, if can't then put one you have picked up back.

Problem with this is it is really slow. The constant contention makes it run a lot slower than first solution, even though that one ran into deadlock It is possible that one single person never eats. A given person can only eat if both its neighbours aren't eating.

Just to be safe 1 THIS

2 THIS

Equivalence If have semaphore then can impelement monitor, and vice versa

Implementing semaphore with monitor is easy Other way around is harder due to lack of conditional. Associate semaphore with condition variables.

Lock free algorithms Locks aren't the only way to protect resource/datastructure Libraries exist for this for most languages

Lock free modification uses `cas()` Compare and swap Basically check if value checked has changed. If not then replace with new value. Keep checking until the value

Deadlock When 2 processes want something that the other has and vice versa. (Classic deadlock) Anything waiting for a resource that the 2 processes fighting for also indirectly joins the deadlock.

Conditions

1. Circular linked list
2. Resources can't be shared
3. Only owner can release resource
4. Process holding resource while requesting another. **Often forgotten**

Detection Use graph and find cycles Allocation are node, request are edge

Results Deadlock has to be resolved somehow Killing process. Bad since you don't know current state of process. Can use priority or age to select process to delete. May enter back into deadlock instantly. Can remove all processes. Overkill but solves problem. Can rollback or restart. But make sure that same process not deleted/rolledback constantly

Killng stuff not generally good idea due to potentially inconsistent state.

Prevention Ordering of resources Make circular list of processes, so you can only get resources in a predetermined order. Prevents the basic deadlock situation.

If A-¿B-¿C. If have A can get B If have B, have to release, get A then get B

Resources not sharable.

Only owner can release resource

Process can hold resource while requesting another One resource at a time Return before requesting Allocate all resources at same time.

Avoidance More of a runtime thing. Very conservative stratergy Check requests' safety, before allowing it to go ahead. Might end up have resource free while something waits for it due it it possibly making deadlock later.

Occur when both processes have resources and both require one more from depleted resource pool.

Banker Algorithm If given request, then assume permisison granted. Go through processes, and see what processes it wants not and in the future, compare with the list of available resources. Then actually grant it access if it meets this. Otherwise do nothing.

Check if the processes can finish with given resources left if trying to allocate to another. Need to ensure that all processes can definitely finish. "Deadlock may occur" not allowed. Only "deadlock can't occur" allowed

12 Lecture 13

Distributed Deadlock Instead of worrying about order that you get resources, instead order processes that are executed by their priorities. Lower levels are rolled back and higher ones go earlier

Detection A -¿ B means that A is waiting on something B holds Sometimes that you can only see deadlock when you combine multiple graphs. Can have something ask all sites for their waitFor graphs (to try find global deadlocks)

Centralised deadlock detection Timing is an issue and may lead to false positives. Graph only approximation of real resource allocation. Can use timestamps to avoid false deadlock

Distributed approach Extra node (P_{ex}) in each local waitfor graph Local processes waiting on external stuff goes to this P_{ex} If cycle with P_{ex} in it exist, **could** have a deadlock. Each local site can check this. Then goes and ask other sites Deadlock handled if found Otherwise continue as normal

Time stamp prevention wait-die If resource held by older process, younger process can't wait, and dies. Since it hasn't done as much work, and tries again.

If resource held by younger process, older process allowed to wait.

Keep age for when you first try get a resource

wound-wait Younger process allowed to wait for older process. Other away around. Older process doesn't wait for younger process. Kills it and takes resource.

Messages Can be used to control concurrency Sending information to another process can be done by

1. Shared resource
2. Message passing
    - Address message
    - Transport message
    - Notify receipeint ANOTHER ONE
    - `send(destination, message)`
    - `receive(source, message)`
    - `write(message)`
    - `read(message)`

Design decisions Either have sender block or not If writing to a file, want to send information as soon as possible.

Receiver should be blocking, you don't want any interruptions. Receiver can choose a bunch of message types to stick around for and receive if any come.

If sender doesn't block then need somewhere to store the messages until the receiver retrieves the message.

Storaging messages

1. Have sender send message striahg to other process
2. Or have the page saved, and pointer sent so receiver knows where it is
    - Need to not overwrite information until it hsa been read
    - Should also make this address read only for recepient so it can't be tampered with

COMMUNICATION PROCESS TO PROCESS (12)
Indirect Communication Mailbox and ports Mailbox ownership

1. Owned by system
    - Persist without process (if finish)
2. Owned by process
    - Creator pass ability to receive
    - mailbox gone if process finish

UNIX pipes Something that contains data Buffering mechanism Pipe fills designated array with file descriptors (address on open file table) which basically makes it look like it is variable that contains the file/s Can use typical `read` and `write` calls `ls -al | grep Doc | wc -l` Vertical bar is pipe

1. Create new process using fork Call exec on ls Forks to grep
    - ls end is writing, grep end is reading

UNIX does this piping by redirecting `stdin` and `stdout`
If full (due to specified pipe size)