ECE344: Operating Systems

Arnav Patil

University of Toronto

Contents

1 Introduction **2**

2 The Kernel **2**

3 Libraries **3**

4 Process Creation **3**

5 Process Management **4**

6 Basic IPC **5**

7 Process Practice **6**

8 Subprocesses **7**

9 Basic Scheduling **8**

10 Advanced Scheduling **9**

11 Virtual Memory **11**

12 Page Tables **12**

13 Page Table Implementation **13**

14 Virtal Memory Lab Primer **14**

15 Priority Scheduling and Memory Mapping **15**

16 Threads **15**

17 Threads Implementation **16**

18 Sockets **18**

19 Midterm Review **20**

20 User Threads Lab Primer **20**

21 Locks **20**

22 Locks Implementation **21**

1 Introduction

- Any software that gets written either:
  - is the operating system, or
  - interacts with the operating system.
- The OS as a resource manager:
  - allows multiple programs to execute at the same time
  - manages/protects memory, IO devices, etc

Three Core OS Concepts

- Virtualization – share a memory/resource by mimicking multiple independent copies
- Concurrency – handles multiple things happening
- Persistence – retains data consistency w/o power

2 The Kernel

- The OS provides the illusion that each program has full access to all resources – on its own machine
- Called virtualization because one physical machine but illusion of multiple virtual machines
- Kernel mode is a privilege level on a CPU that gives access to more instructions
  - Different architectures call it different things
  - Instructions allow only trusted SW to interact with the HW
- Program – file containing instructions and data
- Process – instance of a process being executed
- User mode > supervisor > hypervisor > machine

System Calls

- Transitions b/w user and kernel mode, OS API
  - Create/destroy threads, allocate/deallocate memory, etc.
- API – abstracts details and describes arguments and return value of a function
- ABI – specifies details, specifically how to pass arguments and where the return value is
- E.g. say a program calls `read()`
  - Execution goes via library and issues 'heap'
  - Trap invokes the kernel which accesses the disk
  - Kernel returns the results to the program
- The kernel is a long, constantly running program
  - Link using libraries, there's no `main()`
  - Lets you load code (modules)
  - Code executes on-demand

- If you write a kernel module, you can execute privileged instructions
- Monolithic kernels have less features in user mode, microlithic kernels have more
- Hybrid kernels are between monolithic and microlithic
- Nano/pico kernels have even more features in user mode
- ISAs – x86/64 (amd64), arm64 (aarch64), riscv

File Descriptors

- IPC – inter-process communication is trasferring data between processes
- File descriptor – resource that uses read/write (stores as an index)

3 Libraries

- Libraries are used as part of the OS
- Apps may pass through multiple layers of libraries
- An OS consists of the kernel and libraries required for your applications

Dynamic Libraries Are For Reusable Code

- C standard library is a dynamic library (`.so`) like any other on the system
  - Collection of `.o` files containing function definitions
- OS loads `libc.so` in memory only once during boot

Static vs Dynamic Libraries

- Drawbacks to static:
  - Statically linking prevents re-using libraries
  - Any updates to a static library requires the executable to be recompiled

Dynamic Library Updates Can Break Executables

- A dynamic library update may subtly break an ABI causing a crash
- `structs` are laid out in memory w/ the fields matching the order of declaration
- Semantic versioning meets developer's expectations
  - Given a version number MAJOR.MINOR.PATCH increment:
    * MAJOR when you make incompatible API/ABI changes
    * MINOR when you add functionality in a backwards compatible manner
    * PATCH when you make backwards compatible bug fixes

4 Process Creation

- Process is a running instance of a program
  - Virtual registers
  - Virtual memory
  - File descriptors – an array of numbers that points to files that the kernel is managing

Process Control Blocks

- In Linux this is the `task_struct`
- Contains process state, CPU registers, scheduling, memory management, IO status info
- Each process gets a unique PID assigned to it
- We can read processes using the `proc` filesystem
  - `/proc` doesn't contain real files, but we can use it as such
  - Every directory that's a number is a currently running process (PID)
- Windows – we load program into memory and create the PCB
- Linux – decomposes process creation into more flexible abstractions

Cloning a Process

- Pause currently running process, copy its PCB into a new one. This reuses all info from the old process, including variables
- Distinguished by a parent-child relationship
- `int fork(void)` creates a new process. Returns:
  - -1 on failure
  - 0 in the child process
  - the child's PID in the parent process
- `execve()` replaces a process with another program
  - `pathname` – path of the program to load
  - `argv` – array of strings, arguments to process
  - `envp` – same as argv but for environment
- Modern OS's are smart and won't let you make infinite forks.

5 Process Management

Linux Process Management

- Can read process by doing `/proc/<PID>/states/ grep state`
  - R – running and runnable
  - S – interruptible sleep
  - D – uninterruptible sleep
  - T – stopped
  - Z – zombie
- The kernel allows us to explicitly stop processes but we must restart them
- After the kernel initializes, it creates a single process
  - Looks for program in `/sbin/init`
- `init` is responsible for executing every other process, it must always be active, else kernel thinks you're shutting down
- Using `htop` helps us keep track of existing processes
- Kernels will eventually recycle PIDs of finished processes

Maintaining the Parent-Child Relationship

- OS sets exit status to a process that's finished executing
- Minimum acknowledgment the parent has to do is read the child's exit status
- Two possible situations:
  - Child exits first (zombie process)
  - Parent exits first (orphan process)
- `wait(status)` – where to store wait status of the process
  - -1 on failure
  - 0 for non-blocking calls w/ no child change
  - PID of child process with a change
- OS can only remove the zombie's entry after `wait()` retrieves its PID

Zombie Processes

- Process has been terminated but hasn't been acknowledged
- Process may have an error where it never reads the child's exit status
- OS can interrupt the parent process to acknowledge the child
- This is a suggestion, and the parent is free to ignore it
  - Basic form of IPC, called a signal
- The OS has to keep a zombie process until it's been acknowledged
- If the parent process ignores it, the zombie has to wait to be reparented

Orphan Processes Need a New Parent

- Child processes still need a process to acknowledge its exit
- OS re-parents the child to `init` – `init` now responsible for acknowledging the child
- `init` accepts all orphans, dead or alive

6 Basic IPC

- IPC is transferring bytes between two of more systems
- Reading/writing is a form of IPC
- `read` just reads data from a file descriptor
- No EOF character, just returns 0 bytes read
  - Kernel returns 0 on a closed file descriptor
- We need to check for errors
- `write` similarly returns the number of bytes written, but we can't always assume success

Standard File Descriptors

- We could close fd 0 (standard input) and open a file instead
- Signals are a form of IPC that interrupts
- Kernel sends a number to your program indicating the type of signal
  - Kernel's handlers either ignore the signal or terminate
  - If the default handler occurs, the exit code will be 128 + the signal number
- `Ctrl+C` sends `SIGINT` – signal interrupt from keyboard

Setting Own Signal Handlers

- Declare a function w/ no return and 1 int argument
- Some common interrupts on Linux:
  - 2: `SIGINT` (keyboard)
  - 9: `SIGKILL` (terminate)
  - 11: `SIGSERV` (seg fault)
  - 15: `SIGTERM` (terminate)
- Processes can be interrupted at any point of execution, and resumes once the signal handler returns – example of **concurrency**
- `kill PID` sends `SIGTERM` signal but won't terminate if the process is in uninterruptible sleep
- `kill -9 PID` will kill the process no matter what

Non-Blocking Calls

- A non-blocking call returns immediately so we can check if something happens
- To turn `wait` into a non-blocking call, we can use the flag `WNOHANG` in options
- To react to changes in a non-blocking call we can either use polling or an interrupt

7 Process Practice

- Uniprogramming is for old-batch OSs
- Uniprogramming is when only one process is running at a time – no parallelism and no concurrency
- Multiprocessing – parallel or concurrent both possible, we want parallel AND concurrent

Scheduler Decides When to Switch

- To create a process, the OS has to at least load it into memory
- While maintaining, the scheduler decides when it's running
- First we focus on mechanics of switching processes

Core Scheduling Loop

1. Pause currently running process
2. Save its state so you can restore later
3. Get next process to run from scheduler
4. Load next process' state

- Cooperative multitasking is when the process uses a syscall to tell the OS to pause it
- True multitasking is when the OS retains control and pauses processes

Context Switching

- Name for switching processes
- We have to save all register values using the same CPU the process is already using
- Hardware support for saving state, but we may not want to save everything
- Context switching is pure overhead, we want to minimize

A New API

- `int pipe(int pipefd[2]);`
  - Returns 0 on success and -1 on failure (sets errno)
  - Forms a one-way communication channel using two file descriptors
  - `pipefd[0]` is read and `pipefd[1]` is the write end
- Kernel-managed buffer, any data written to one is read on the other end

8 Subprocesses

We Want to Send/Receive Data From a Process

1. Create a new process that launches command line argument
2. Send string `"Testing\n"` to that process
3. Receive any data it writes to that process

A More Convenient API – `execlp`

- Doesn't return on success, -1 on failure
- Will let you skip string arrays
- It will also search for executables using the PATH environment variable

Final APIs – `dup` and `dup2`

- Returns a new file descriptor on success, -1 on failure and sets errno
- Copies file descriptor so both refer to the same thing

9 Basic Scheduling

Preemptible and Non-Preemptible Resources

- Preemptible resources can be taken away and used for something else e.g. a CPU
- The resource is then shared through scheduling
- A non-preemptible resource can't be taken away w/o acknowledgment e.g. disk space
- The resource is instead shared through allocations and deallocations
  - Parallel and distributed systems may allow you to allocate a CPU

Dispatchers and Schedulers Work Together

- A dispatchers is a low-level mechanism responsible for context switching
- A scheduler is a high-level policy responsible for deciding which processes to run

Scheduler Runs Whenever a Process Changes State

- For non-preemptible resources – process runs until completion, once started
- Scheduler only makes decision once the process is terminated
- Preemptive allows the OS to run scheduler at will

Important Metrics

- Minimize waiting time and response time
- Maximize CPU utilization
- Maximize throughput
- Fairness

First-Come First-Serve – FCFS

- Most basic form of scheduling
- First process that arrives gets access to CPU
- Processes stored in a FIFO queue

Shortest Job First – SJF

- Always schedule job w/ shortest burst time first
- Still assuming no preemption
- But it is not practical
  - Likely optimal at minimizing average wait time
  - Don't know burst time of each process
  - Long jobs may be starve (or never execute)

Shortest Remaining Time First – SRTF

- Assume that minium waiting time is 1 unit, optimize average waiting time

Round-Robin – RR

- Haven't discussed fairness so far – trade-offs
- OS divides execution into time slices (or quotas)
- Maintain a FIFO queue of processes similar to FCFS
- Pre-empt is still running at end of the quantum and re-add to queue
- RR performance depends on quantum length and job length
  - RR has low response times and good interactivity
  - Fair allocation of CPU and low average waiting time
  - Performance depends on quantum length
    - Too high – becomes FCFS
    - Too low – too many context switches (high overhead)
- RR has poor average waiting time when jobs have similar lengths

Scheduling Trade-Offs

- FCFS – most basic scheduling algorithm
- SJF – tweak to reduce waiting time
- SRTF – uses SJF but with preemption
- RR – optimizes fairness and response time

10 Advanced Scheduling

Adding Priorities

- We may favour some processes over others
- Run high priority processes first, round-robin processes of equal priority
- On Linux, -20 is the highest priority and 19 is the lowest
- We may lead processes to starvation is there's lots of high-priority loads
- One solution is to have the OS dynamically change the priority

Priority Inversion

- We can accidentally change priority of low to high, would depend on if a high-priority process depended on a low-priority process
- Solution is to have priority inheritance – inherit priority of the waiting process
- Idea is to separate processes that users interact with
  - Foreground processes are interactable and need good response time
  - Background processes may just need good throughput

Using Multiple Queues

- Create different processes for foreground and background processes
  - Foreground – RR
  - Background – FCFS
- Now we have to schedule b/w queues
  - RR the queues or use a priority system
- We'll assume symmetric multiprocessing (SMP)
  - All CPUs connected to same physical memory
  - CPUs all have their own (lowest-level) caches
- One approach is to use same scheduling for all CPUs
  - Only one scheduler – adds processes while CPU available
  - Pros: good CPU utilization, fair to all processes
  - Cons: not scalable, poor cache locality
- Another is to create per-CPU schedulers
  - Assign new processes to CPUs with the lowest # of processes
  - Pros: easy to implement, scalable, good cache locality
  - Cons: load imbalance
- We can also compromise b/w global and per-CPU
  - Keep a global scheduler that can rebalance per-CPU queues
    * If a CPU is idle, take a process from another CPU (work stealing)
  - We may have some processes that are more sensitive to caches
  - Using processor affinity
    * Preference of a process to be scheduled on the same core
- Gang scheduling (co-scheduling)
  - Multiple processes may need to be scheduled simultaneously
  - Scheduler on each CPU cannot be completely independent
  - Requires a global context-switching across all CPUs

Real-Time Scheduling

- Real-time means there are time constraints, either for a deadline or rate
- Hard and soft real-time systems

Linux FCFS and RR Scheduling

- Use a multilevel queue scheduler for processes with the same priority
  - Also let the OS dynamically adjust the priority
  - Soft real-time processes – always schedule for the highest priority first
  - Normal processes – adjust priority based on aging

O(1) Scheduling Issues

- Now kernel has to detect processes which are interactive using heuristics
- Processes that sleep a lot may be more interactive

Ideal Fair Scheduling (IFS) is Fairest but Impractical

- Performs way too many context switches
- Have to constantly scan all processes at O(N)

Completely Fair Scheduler (CFS)

- For each runnable process, assign it to a virtual runtime
- At each scheduling point, increase virtual runtime by t $\times$ weight (priority)
  - Virtual time monotonically increases
    * Scheduler selects process based on lowest virtual runtime
    * Compute dynamic runtime based on IFS
- Implemented on red-black tree, self-balancing BST
  - O(log(n)) insert, delete, find operations

11 Virtual Memory

Requirements of Virtual Memory

- Multiple processes must co-exist
- Processes unaware they are sharing physical memory and cannot access each other's data, unless explicitly allowed
- Performance close to actual physical memory
- Limit amount of wasted memory – fragmentation

Segmentation/Segments are Coarse-Grained

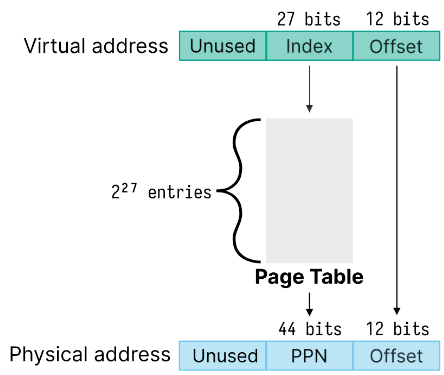
- Each segment is a variable sized – dynamically allocated
- Legacy technique that's not really used anymore
- Segments are large and heard to relocate, but leads to fragmentation
- Each segments contains a base, limit, and permissions
- MMU checks that offset is within limit, then calculates base + offset and does permission checks
  - If not, results in a seg fault

Memory Management Unit (MMU)

- Maps virtual addresses to physical addresses and does permission checks
- One technique is dividing memory into fixed size pages (4096 bytes)
- A page in virtual memory is called a page, and a page in physical memory is called a frame

Addressing

- Typically don't use all 64 bits its virtual address
- CPUs may have diff levels of virtual addresses you can use
- We'll assume 39 bits virtual address space, allows for 512 GiB of memory (called Sv39)
- Implemented with page table indexed by Virtual Page Number (VPN) – looks up Physical Page Number (PPN)



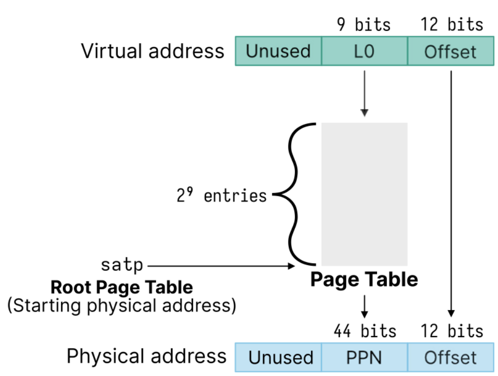
- Page Table Entry (PTE) also stores flags in lower bits

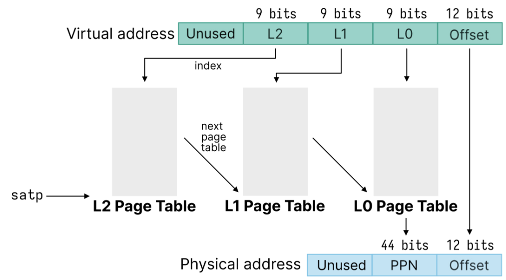
Each Process Gets Its Own Page Table

- When you call `fork()` on a process, it will copy the page table from the product
- Turn off the write permissions so that the kernel can implement copy-on-write
- We don't need to copy the full page table – syscall `vfork()`
  - Shares all memory with the product
  - Only used in very performance sensitive programs

12 Page Tables

- Most programs don't use all virtual memory space, so how do we take advantage?





- Multi-level page tables save space for sparse allocations

- Given physical pages, the OS uses a free (linked) list
- Unused pages contain the next pointer in the free list
  - Physical memory gets initialized at boot, remember
- To allocate, remove from the free list, and to deallocate, add back to the free list

13 Page Table Implementation

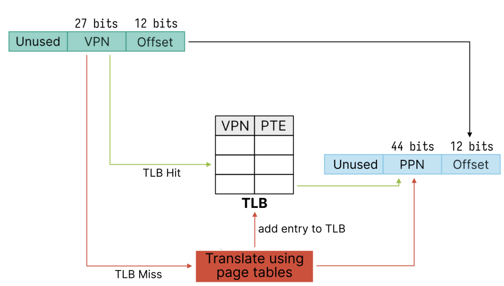
Alignment: Memory Eventually Lines Up With Byte 0

- If pages are 4096 bytes, then pages always start when all offset bytes (12) are 0
- How many levels do I need?
  - We want each page table to fit into a single page
  - Find number of PTEs we could have in a single page 2^{10}

$$\text{Number of levels} = \frac{\text{Virtual bits} - \text{Offset bits}}{\text{Index bits}}$$

Page Table for Every Memory Access is Slow

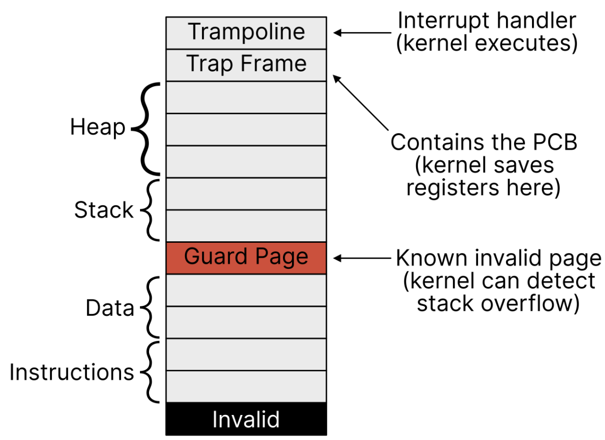
- Need to follow pointers across multiple levels of page tables
- We'll likely access the same page multiple times temporally close to the first
  - Process may only need a few VPN-PPN at a time
  - Use a computer science classic – Caching!



TLB – Translation Look-aside Buffer

Context Switching Requires Handling the TLB

- Can flush the cache or attach a PID to the TLB
- RISC-V & most implementations just flush the cache
- `sbrk` call grows or shrinks your heap, but recall, the stack has a set limit
- To grow, will grab pages from the free lists to fulfill the request
  - Kernel sets `PTE_V` (a valid bit) and other permissions
- Difficult to use in memory allocations, rarely shrinks the heap – will stay claimed as the kernel can't free pages
- Memory allocations use `mmap` to bring in large blocks of virtual memory



- The kernel can allow for processes to ccess fixed virtual memory/data without using a system call

Page Faults

- Type of exception for virtual memory accesses
  - Generated if it cannot find a translation or the permission check fails
- Allows the OS to handle it
- MMU is the hardware that uses page tables:
  - May be a single page table (wasteful)
  - Use kernel-allocated pages from a free list
  - Be multi-level to save pages for sparse allocations
  - Use a TLB to speed up memory accesses

14 Virtal Memory Lab Primer

Not covered in these notes.

15 Priority Scheduling and Memory Mapping

Dynamic Priority Scheduling

- May also be called feedback scheduling
- We let the algorithm manage priorities – we set time slices and measure CPU usage
- Increase the priority of processes that don't use their full time slices, and decrease priority of those that do
- Each process gets assigned a priority when started, P_n
- Pick the lowest number to schedule, if it yields, then pick the next lowest
  - Break ties with arrival order
  - If a lower number becomes ready, switch to it
- Record how much time each process executes for in this priority interval, C_n
- At the end of the priority interval, update each process

$$P_n = \frac{P_n}{2} + C_n$$

- Reset C_n back to 0 at the end of the priority interval

16 Threads

Aside: Concurrency and Parallelism Aren't the Same

- Concurrency – switching between 2 or more tings
  - Goal: make progress on multiple things
- Parallelism – running two or more things at the same time
  - Goal: run as fast as possible

Threads Are Like Processes With Shared Memory

- Same principle as a process, except they share memory by default
- They have their own registers, PC, & stack
- They have the same address space, so changes appeared in each thread
- Need to explicitly state if any memory is specific to a thread

One Process Can Have Multiple Threads

- By default, a process executes code in its own address space
- Threads allow multiple executions in the same address space
- Threads can express concurrent (assuming 1 CPU)
- A process can appear like it's executing multiple locations at once
  - OS is context switching within a process

Threads Are Lighter Weight

- Processes
  - Independent code/data/heap
  - Independent execution
  - Has own stack/registers
  - Expensive creation/switching
  - Completely removed from OS on exit
- Threads
  - Shared code/data/heap
  - Lives within an executing process
  - Has own stack/registers
  - Cheap creation and context switching
  - Stack removed from process on exit

Detached Threads

- Joinable threads – wait for someone to call `pthread_join()` then they release their resources
- Detached threads – release resources when they terminate

17 Threads Implementation

Multithreading Models

- Where do we implement threads?
  - User threads of kernel threads
- User threads are completelt in user space
  - Kernel doesn't treat the threads any differently
- Kernel threads are implemented in kernel space
  - Kernel manages everything for you, and can treat threads specially

Thread Tables

- Similar to process tables – could be in user space or kernel space
- User threads need a run-tome system to determine scheduling
- For pure user-level threads
  - Fast to create and destroy – no syscall, no switching
  - One threads blocks, whole process blocks (no true parallelism)
- For kernel-level threads
  - Slower, creation needs syscalls
  - If one thread blocks, the kernel can schedule another
- Thread library maps user threads to kernel threads

- Many-to-one – completely implemented in user space, kernel sees only one process
- One-to-one – one user thread maps directly to one kernel thread
- Many-to-many – many user-level threads map to many kernel-level threads

Many-to-One

- Is a pure user-space implementation
- Fast and portable – doesn't depend on system
- Drawback is that one thread blocking causes all threads to block
- Cannot achieve true parallelism

One-to-One

- Thin wrapper around syscall to make easier to use
- Exploits full parallelism of machine
- We need to use slower syscalls and we lose some control
- Typically, this is the actual implementation used, assume this for Linux

Many-to-Many (Hybrid)

- Key idea: there are more user-level threads than kernel-level threads
- Cap number of kernel-level threads to number we can run in parallel
- Can get most out of multiple CPUs and reduce number of syscalls
- Leads to a complicated thread library
  - Could end up blocking other threads

Threads Complicate the Kernel

- How should `fork` work with a process with multiple threads?
  - Linux copies only the thread that called `fork`
  - If it hits `pthread_exit()` then it'll always exit with 0

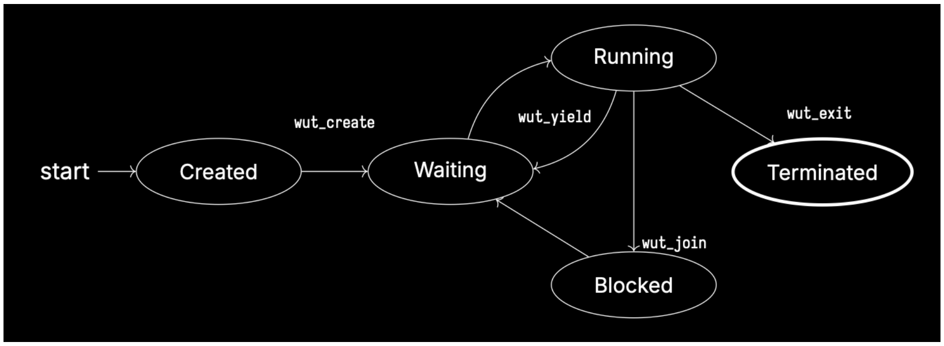
Signals

- Linux picks one random thread to handle the signal
  - Makes concurrency hard, any thread could be interrupted

Thread Pools

- Goal of many-to-many is to avoid creation costs
- Thread pool creates a certain number of threads and a queue of tasks
- As requests come in, wake them up and give them work
  - Reuse and put to sleep when there is no work

Many-to-One



Thread Scheduler

- Create a list (queue), run the thread at the front, when it yields, add it to the back
- You'll have to context switch (remember to save register)
- These are cooperative threads (they have to be nice)

Next Complication

- A program that creates 8 threads increments a program 1000 times.

Both Processes and (Kernel) Threads Allow Parallelism

- Each process can have multiple threads
  - Most implementations use one-to-one mapping
  - OS has to handle forks and signals
  - Now have synchronization issues

18 Sockets

- Sockets are another form of IPC
- We've seen pipes and signals and talked about shared memory
- Previous IPC assumes processes are on the same machine
- Sockets enable IPC between physical machines, typically over some network

Servers Follow 4 Steps

1. `socket()` – creates the socket
2. `bind()` – attach the socket to some location
3. `listen()` – indicate you're accepting connection & set the queue limit
4. `accept()` – return the next incoming connection for you to handle

Clients Follow 2 Steps

- `socket()` – create the socket
- `connect()` – connect to some location, socket can now send and receive data

`socket()` System Call

- `int socket(int domain, int type, int protocol);`
- `domain` is the general protocol, specified further by `protocol`
  - `AF_UNIX` – local communication
  - `AF_INET` – using network interface, IPv4
  - `AF_INET6` – using network interface, IPv6
- `type` is usually a stream or datagram socket

Stream Sockets

- Use TCP – all data sent by client appears in same order on the server
- Forms a persistent connection between the client and server
- Reliable but may be slow

Datagram Sockets

- Uses UDP – sends messages between the client and the server
- No persistent connection between client and server
- Is faster but messages may be reordered and dropped

`bind()` System Call

- Sets a socket to an address
- `int bind(int socket, const struct sockaddr *address, socklen_t address_len);`
- `socket` is the file descripter returned by the `socket` syscall.
- Different `sockaddr` structures for different protocols

`listen()` System Call

- Sets queue limit for incoming connections
- `backlog` is the limit of outstanding connections
  - Kernel manages this queue, set to 0 for default

`accept()` System Call

- Blocks until there is a connection

`connect()` System Call

- Allows a client to connect to an address
- If this call succeeds then `sockfd` can be used as a file descriptor

19 Midterm Review

Not covered in these notes.

20 User Threads Lab Primer

Not covered in these notes.

21 Locks

- A data race is when wo concurrent actions access the same variable and at least one of them is a write
- Assume any atomic instruction happens all at once
- Meaning we can't preempt it

Three Access Code (TAC)

- Intermediate code used by compilers for analysis and optimization
- Statements represent one instruction which is a fundamental operation
- GIMPLE is the TAC used by gcc
  - Easier to reason about your code w/o low-level assembly
- Can create mutexes statically or dynamically

Critical Sections

- Be careful to avoid deadlocks if you are using multiple mutexes
- Critical section means only one thread executes in that section of code
- There should only ever be one thread in a critical section at once
- Liveness (aka progress)
  - If multiple threads reach a critical section, only one may proceed
  - Critical section can't depend on outside threads
  - But we can mess up and deadlock
- Bounded waiting (aka starvation free)
  - Waiting threads must eventually proceed
- Should also have minimal overhead
  - Efficient – don't consume resources while waiting
  - Fair – want each thread to wait approximately the same time
  - Simple – should be easy to use and hard to misuse

22 Locks Implementation

- Minimal hardware requirements
  - Loads and stores must be atomic
  - Instructions execute in order
- 2 main algorithms we could use for this
  - Peterson's algorithm
  - Lampart's bakery algorithm
- Compare and swap is a common atomic hardware instruction
- Still has the busy wait problem
  - Consider a uniprocessor system, if you can't get the lock you should yield and let the kernel schedule another process
  - On a multiprocessor system, you could just try again
- Thundering herd problem – multiple threads may be waiting on the same lock

Read-Write Locks

- With mutexes and spinlocks, you have to lock data even for a read since we don't know if a write may happen
- Reads can happen in parallel as long as no one is writing

Summary

- Mutex or spinlocks are the most simple locks
- Need hardware support to implement locks
- Need some kernel support for wakeup notifications
- If we have a load of readers, we should use a read-write lock