Operating Systems Lecture Notes

Anthony Catterwell

April 9, 2019

Contents

1 Introduction

2 Operating System Structure

2.1 Architectural impact

Architectural features affecting OS*s*

- These features were built primarily to support OS*s*:
  - timer (clock) operationg
  - synchronisation instructions
  - memory protection
  - I/O control operations
  - interrupts and exceptions
  - protected modes of operation (kernel vs. user mode)
  - privileged instructions
  - system calls (including software interrupts)
  - virtualisation architectures
- ASPLOS

2.2 User operating interaction

2.2.1 User v.s. kernel

Privileged instructions

- Some instructions are restricted to the OS
  - known as *privileged* instructions
- Only the OS can:
  - directly access I/O devices
  - manipulate memory state management (page table pointers, TLB loads, etc.)
  - manipulate special *mode bits* (interrupt priority level)
- Restrictions provide safety and security

OS protections

- So how does the process know if a privileged instruction should be executed?
  - the architecture must support at least two modes of operation: kernel mode, and user mode
  - mode is set by status bit in a protected processor register.
    * user programs execute in user mode
    * OS executes in kernel (privileged) mode (OS == kernel)
  - Privileged instructions can only be executed in kernel (privileged) mode
    * if code running in user mode attempts to execute a privileged instruction, the illegal execution trap.

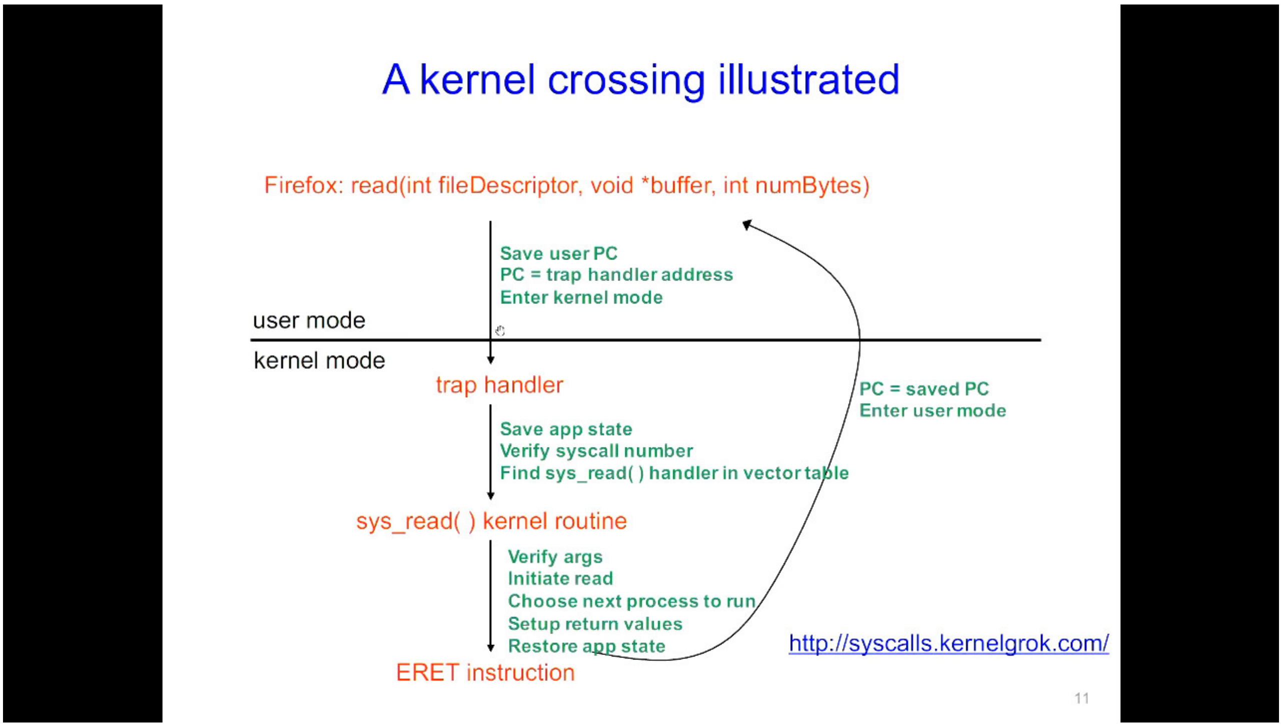
Crossing protection boundaries

- So how do user programs do something privileged?
  - e.g. how can you write to a disk if you can't execute any I/O instructions?
- User programs must call on OS procedure — that is to ask the OS to do it for them.
  - OS defines a set of system calls
  - User-mode program executes system call instruction
- Syscall instruction
  - like a protected procedure call

2.2.2 Syscall

Syscall

- The syscall instruction *atomically*:
  - saves the current PC
  - sets the execution mode to privileged
  - sets the PC to a handler address
- Similar to a procedure call
  - Caller puts arguments in a place the callee expects (registers, or stack)
    * One of the args is a syscall number, indicating which OS function to invoke
  - Callee (OS) saves caller's state (registers, other control states) so it can use the CPU
  - OS function code runs
    * OS must verify caller's arguments (e.g. pointers)
  - OS returns using a special instruction
    * Automatically sets PC to return address and sets execution mode to user.



System call issues

- A syscall is not a subroutine call, with the caller specifying the next PC.
  - the caller knows where the subroutines are located in memory; therefore they can be the target of an attack.
- The kernel saves state?
  - Prevents overwriting of values
- The kernel verify arguments
  - Prevents buggy code crashing the system
- Referring to kernel objects as arguments
  - Data copied between user buffer and kernel buffer.

Exception handling and protection

- *All* entries to the OS occur via the mechanism just shown
  - Acquiring privileged mode and branching to the trap handler are inseparable
- Terminology
  - *Interrupt*: asynchronous; caused by an external device
  - *Exception*: synchronous; unexpected problem with instruction
  - *Trap*: synchronous; intended transition to OS due to an instruction

  In all three cases, they are instances of where something strange happens, and the OS takes control: whether by accident, or by intention.
- Privileged instructions and resources are the basis for most everything: memory protection, protected I/O, limiting user resource consumption.

2.3 Operating System structure

2.3.1 Layers

Operating System structure

- The OS sits between application programs and the hardware
  - it mediates access and abstracts away ugliness
  - programs request services via traps or exceptions
  - devices request attention via interrupts

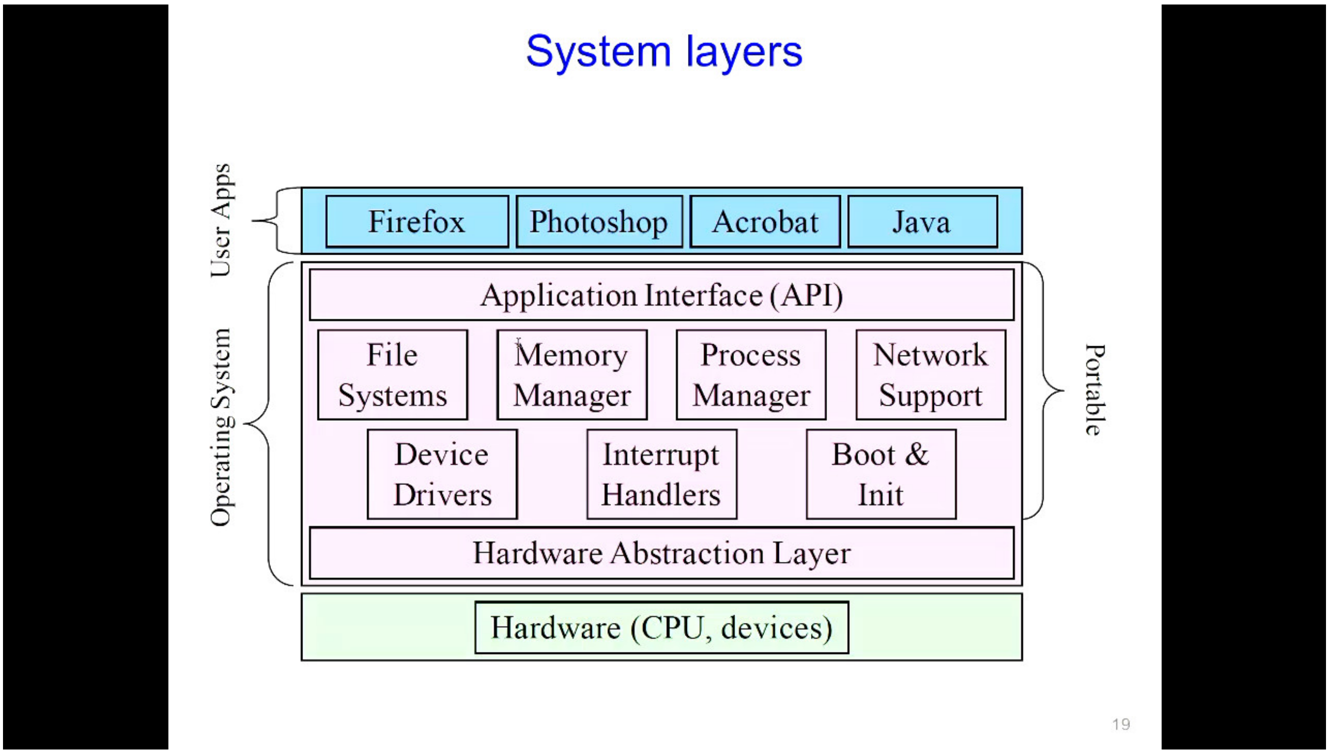
Operating system design and implementation

- Design and implementation of OS not "solvable", but some approaches have proven successful.
- Internal structure of different OS*s* can vary widely.
- Start the design by defining goals and specifications.
- Affected by choice of hardware, type of system.
- *User* goals, and *system* goals
  - User goals: OS should be convenient to use, easy to learn, reliable, safe, and fast
  - System goals: OS should be easy to design, implement, and maintain, as well as flexible, reliable, error-free, and efficient.
- Important principle to separate

  - **Policy**: *What* will be done?
  - **Mechanism**: *How* to do it?

- Mechanisms determine how to do something, policies decide what will be done.
- The separation of policy from mechanism is a very important principle, it allows maximum flexibility if policy decisions are to be changed later (e.g. timer).
- Specifying and designing an OS is a highly creative task of *software engineering*.

System layers



Major OS components

- processes
- memory
- I/O
- secondary storage
- file systems
- protection
- shells
- GUI
- networking

OS structure

- There's no clear hierarchy within an OS — each of them needs access to different things.
- An OS consists of all these components, plus:
  - many other components

- system programs (privileged, and non-privileged)

- Major issue:
  - how do we organize all this?
  - what are all of the code modules, and where do they exist?
  - how do they cooperate?
- Massive software engineering and design problem
  - design a large, complex program that: performs well, is reliable, is extensible, and is backwards compatible.

2.3.2 Examples

Monolithic design

- Traditionally, OS*s* (like UNIX) were built as a *monolithic* entity User programs — OS (everything) — hardware
- Major advantage: cost of module interactions is low (procedure call)
- Disadvantages:
  - hard to understand
  - hard to modify
  - unreliable (no isolation between system modules)
  - hard to maintain
- What is the alternative?
  Find a way to organise the OS in order to simplify its design and implementation.

Layering

- The traditional approach is layering
  - implement OS as a set of layers
  - each layer presents an enhanced *virtual machine* to the layer above
- The first description of this approach was Dijkstra's THE system
  - Layer 5: *Job managers* execute users' programs
  - Layer 4: *Device managers* handle devices and provide buffering
  - Layer 3: *Console manager* implements virtual consoles
  - Layer 2: *Page manager* implements virtual memories for each process
  - Layer 1: *Kernel* implements a virtual processor for each process
  - Layer 0: *Hardware*
- Each layer can be tested and verified independently
- Imposes a hierarchical stricture
  - but real systems are more complex: file systems require VM services (buffer); VM would like to use files for its backing store
  - strict layering isn't flexible enough
- Poor performance: each layer crossing has *overhead* associated with it

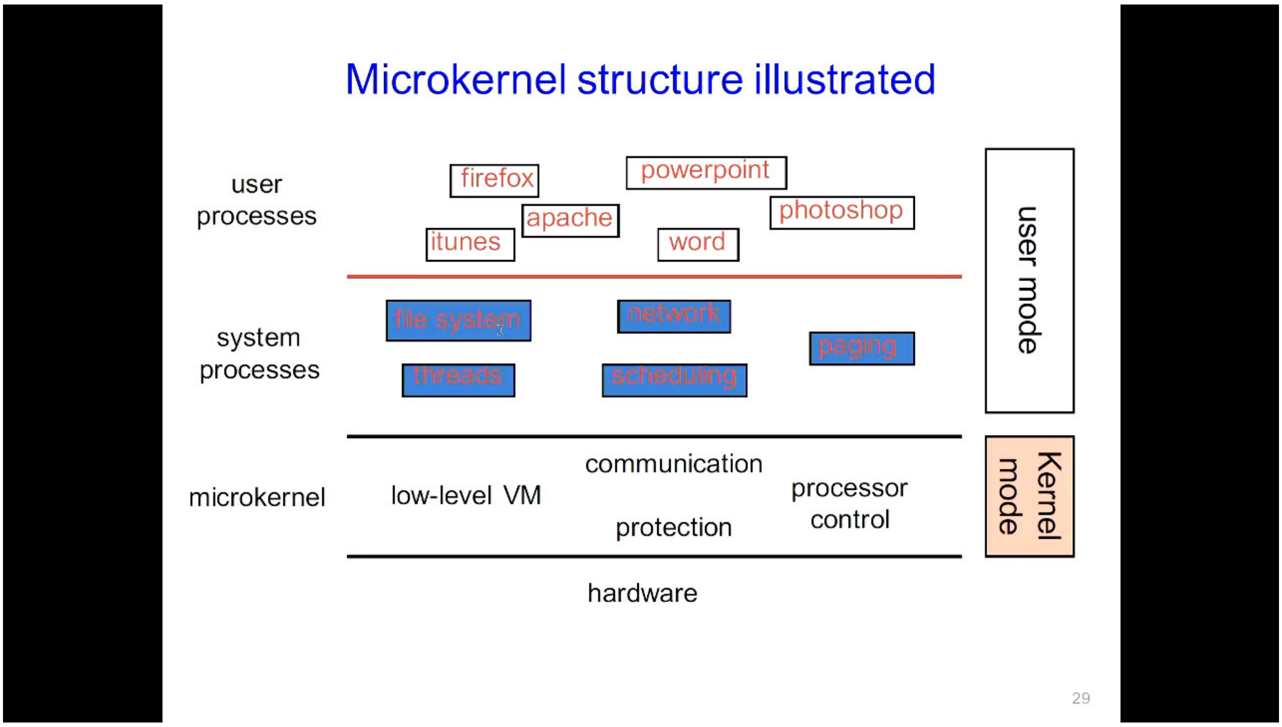
- Disjunction between model and reality: systems modelled as layers, but not really built that way.

Hardware abstraction layer

- An example of layering in modern operating systems
- Goal: separates hardware-specific routines from the *core* OS
  - Provides portability
  - Improves readability

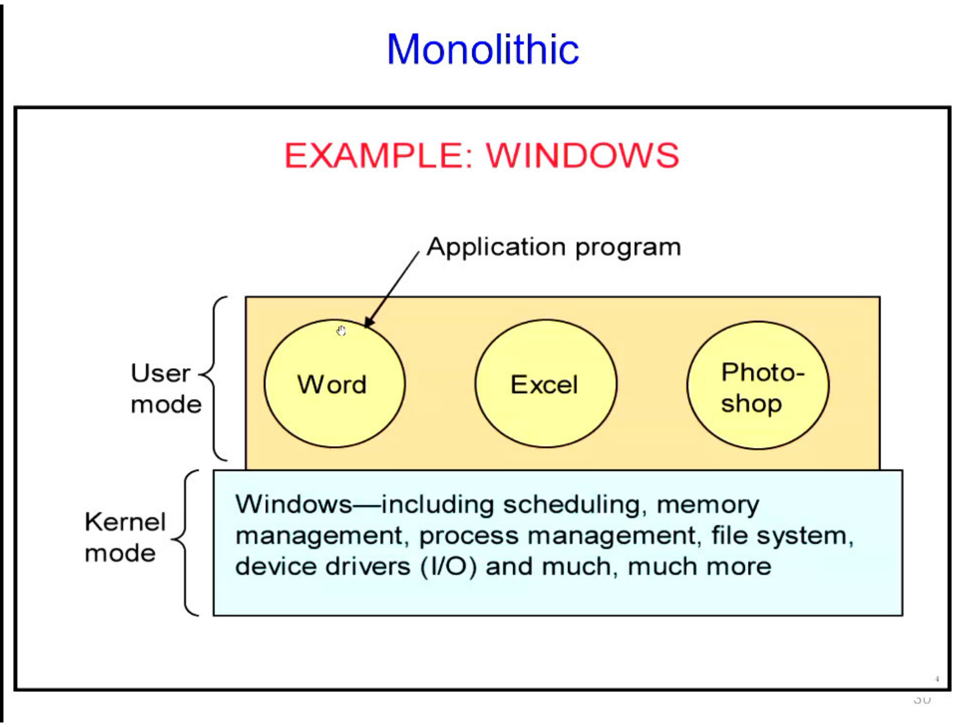
Microkernels

- Popular in the late 80s, early 90s
- Goal: minimize what happens in kernel; item organize rest of OS as user-level processes.
- This results in:
  - better reliability (isolation between components)
  - easy of extension and customisation
  - poor performance (user/kernel boundary crossings)
- First microkernel system was Hydra (CMU, 1970)
  - Contemporaries: Mach (CMU), Chorus (French UNIX-like OS), OS X (Apple), in some ways NT (Microsoft)

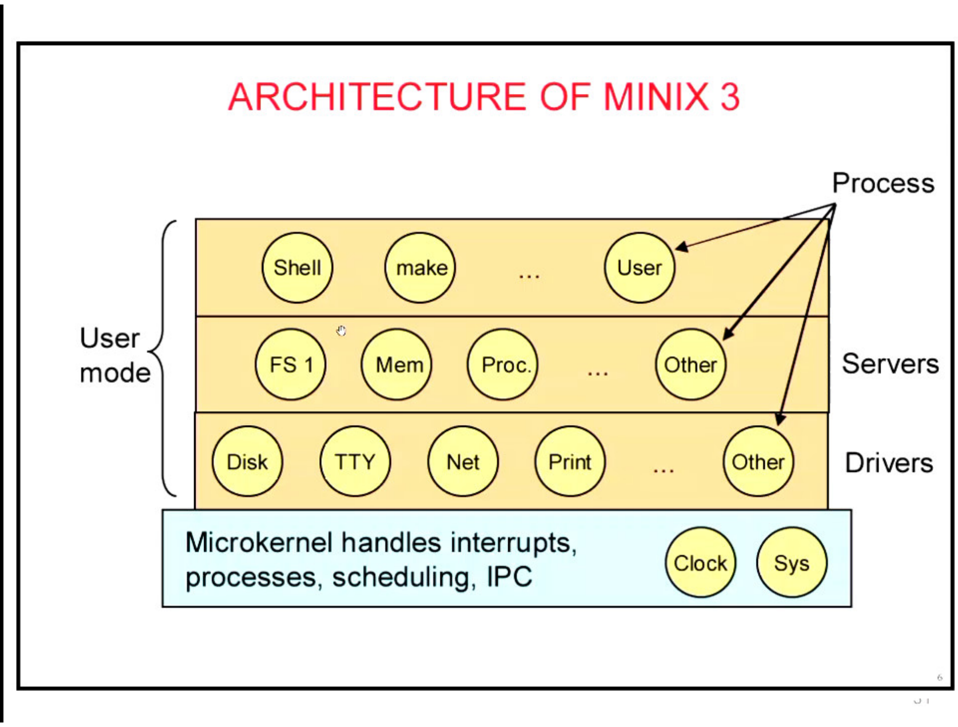


Comparison of OS structures

Windows



MINIX 3



Loadable kernel modules

- (Perhaps) the best practice for OS design
- Core services in the kernel, and others dynamically loaded
- Common implementations include: Solaris, Linux, etc.
- Advantages
  - convenient: no need for rebooting for newly added modules
  - efficient: no need for message passing unlike micro-kernel

  - flexible: any module can call any other module unlike layered model

2.4 Summary

- Fundamental distinction between user and privileged mode supported by most hardware
- OS design has been an evolutionary process of trial and error.
- Successful OS designs have run the spectrum from monolithic, to layered, to micro-kernels
- The role and design of an OS are still evolving
- It is impossible to pick one “correct” way to structure an OS

3 Processes

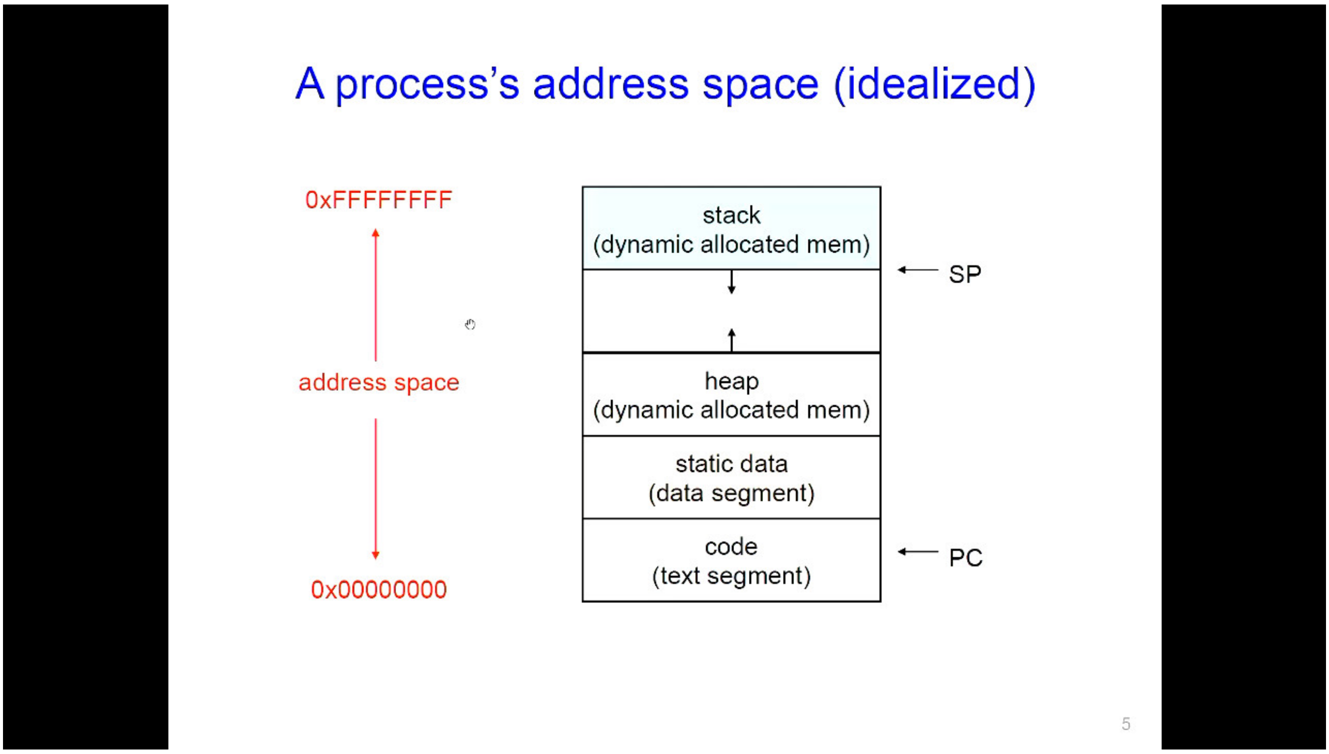
3.1 Process

What is a "process"?

- The process is the OS*s* abstraction for execution
  - A process is a program in execution
- Simplest (classic) case: a *sequential process*
  - An address space (an abstraction of memory)
  - A single thread of execution (an abstraction of the CPU)
- A sequential process is:
  - The unit of execution
  - The unit of scheduling
  - The dynamic (active) execution context (as opposed to the program — static, just a bunch of bytes)

What's "in" a process?

- A process consists of (at least):
  - An *address space*, containing:
    * the code (instructions) for the running program
    * the data for the running program (static data, heap data, stack)
  - *CPU state*, consisting of:
    * the program counter (PC), indicating the next instruction;
    * the stack pointer;
    * other general purpose register values.
  - A set of *OS resources*
    * open files, network connections, sound channels, ...
  - In other words, everything needed to run the program (or to restart, if interrupted).



The OS process namespace

- The particulars depend on the specific OS, but the principles are general;
- The name for a process is called a *process ID* (PID) (an integer);
- The PID namespace is global to the system;
- Operations that create processes return a PID (e.g. fork);
- Operations on processes take PIDs as an argument (e.g. kill, wait, nice).

3.2 Process control block

Representation of processes by the OS

- The OS maintains a data structure to keep track of a process's state
  - called the *process control block* (PCB) or *process descriptor*;
  - identified by the PID.
- OS keeps all of a process's execution state in (or linked from) the PCB when the process isn't running
  - PC, SP, registers, etc.
  - when a process is unscheduled, the state is transferred out of the hardware into the PCB
  - (when a process is running, its state is spread between the PCB and the CPU).

The PCB

- The PCB is a data structure with many, many fields
  - PID
  - parent PID
  - execution state
  - PC, SP, registers

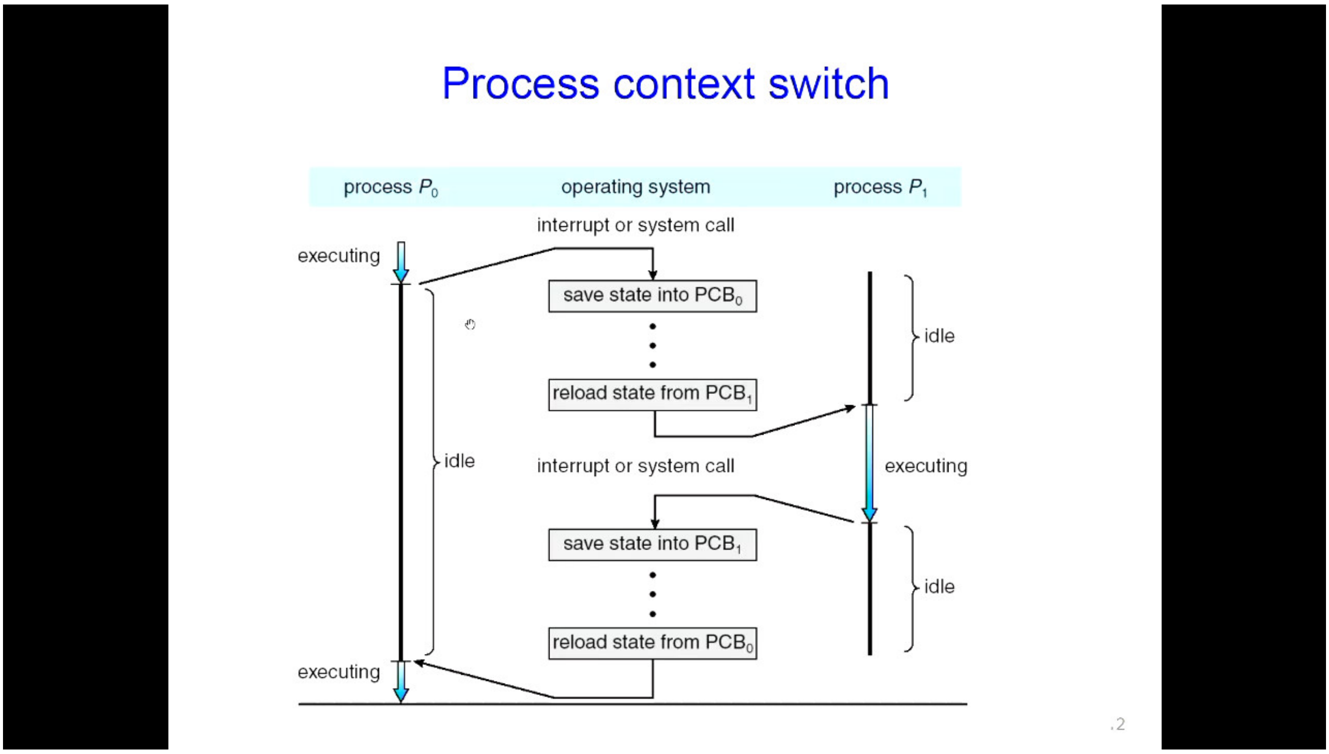
  - address space info
  - UNIX user id, group id
  - scheduling priority
  - accounting info
  - pointers for state queues
- In Linux:
  - defined in `task_struct (include/linux/sched.h)`
  - Over 95 fields!

3.3 Process state & context switch

PCBs and CPU state

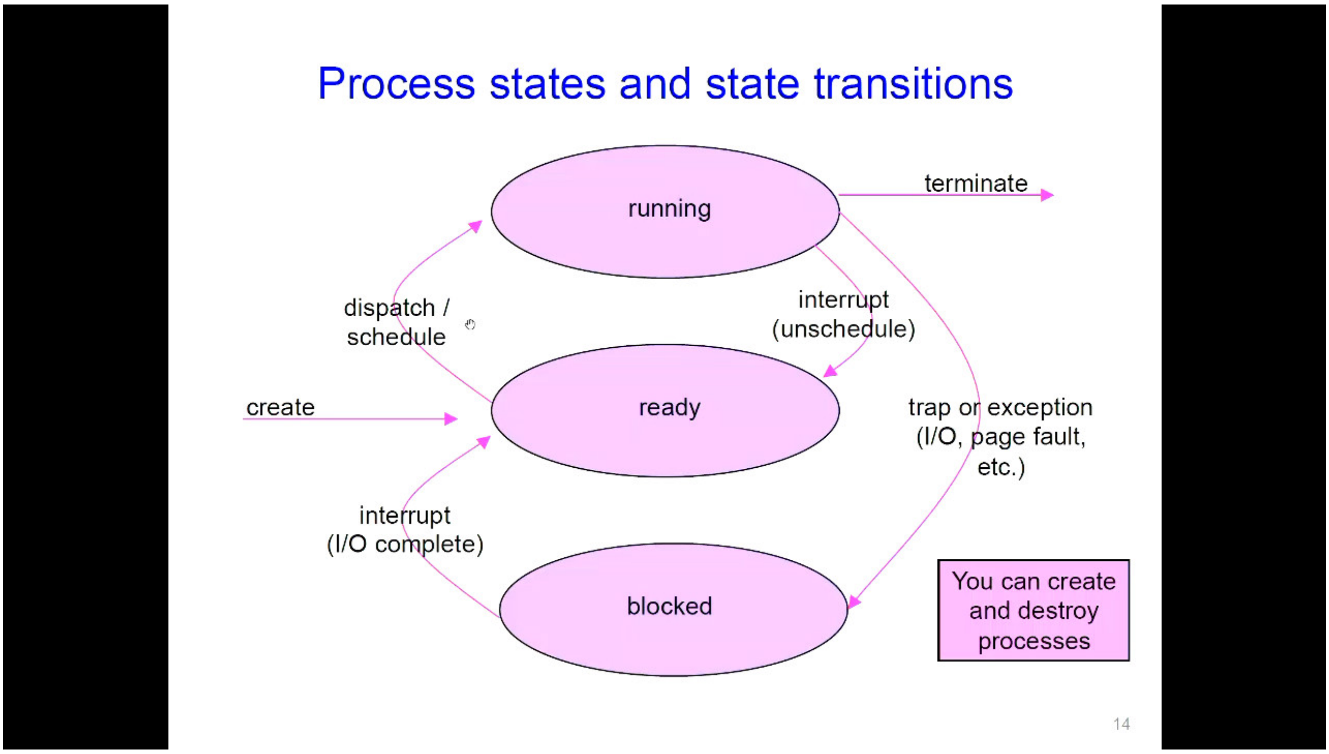
- When a process is running, its CPU state is inside the CPU
  - PC, SP, registers
  - CPU contains current values
- When the OS gets control because of a
  - *Trap*: program executes a syscall
  - *Exception*: program does something unexpected (e.g. page fault)
  - *Interrupt*: A hardware device requests service

  the OS saves the CPU state of the running process in that process's PCB.
- When the OS returns the process to the running state
  - it loads the hardware registers with values from that process's PCB
  - e.g. general purpose registers, SP, instruction pointer
- This act of switching the CPU from one process to another is called a *context switch*
  - systems may do 100s or 1000s of switches per second;
  - takes a few microseconds on today's hardware;
  - still expensive relative to thread-based context switches.
- Choosing which process to run next is called *scheduling.*



Process execution states

- Each process has an *execution state*, which indicates what it's currently doing
  - *ready*: waiting to be assigned to a CPU — could run, but another process has the CPU;
  - *running*: executing on a CPU — it's the process that currently controls the CPU;
  - *waiting* (aka "blocked"): waiting for an event, e.g. I/O completion, or a messing from (or the completion of) another process — cannot make progress until the event happens.
- As a process executes, it moves from state to state
  - UNIX: run `top`, STAT column shows current state
  - which state is a process most of the time?



State queues

- The OS maintains a collection of queues that represent the state of all processes in the system
  - typically one queue for each state (e.g. ready, waiting, ...);
  - each PCB is queued onto a state queue according to the current state of the process it represents;
  - as a process changes state, its PCB is unlinked from one queue, and linked onto another.
- The PCBs are moved between queues, which are represented as linked lists.
- There may be many wait queues, one for each type of wait (particular device, timer, message, ...).

PCBs and state queues

- PCBs are data structures
  - dynamically allocated inside OS memory.
- When a process is created:
  - OS allocates a PCB for it;
  - OS initializes PCB;
  - (OS does other things not related to the PCB);
  - OS puts PCB on the correct queue.
- As a process computes:
  - OS moves its PCB from queue to queue.
- When a process is terminated:
  - PCB may be retained for a while (to receive signals, etc.)
  - eventually, OS deallocates the PCB.

3.4 Process creation and termination

Process creation

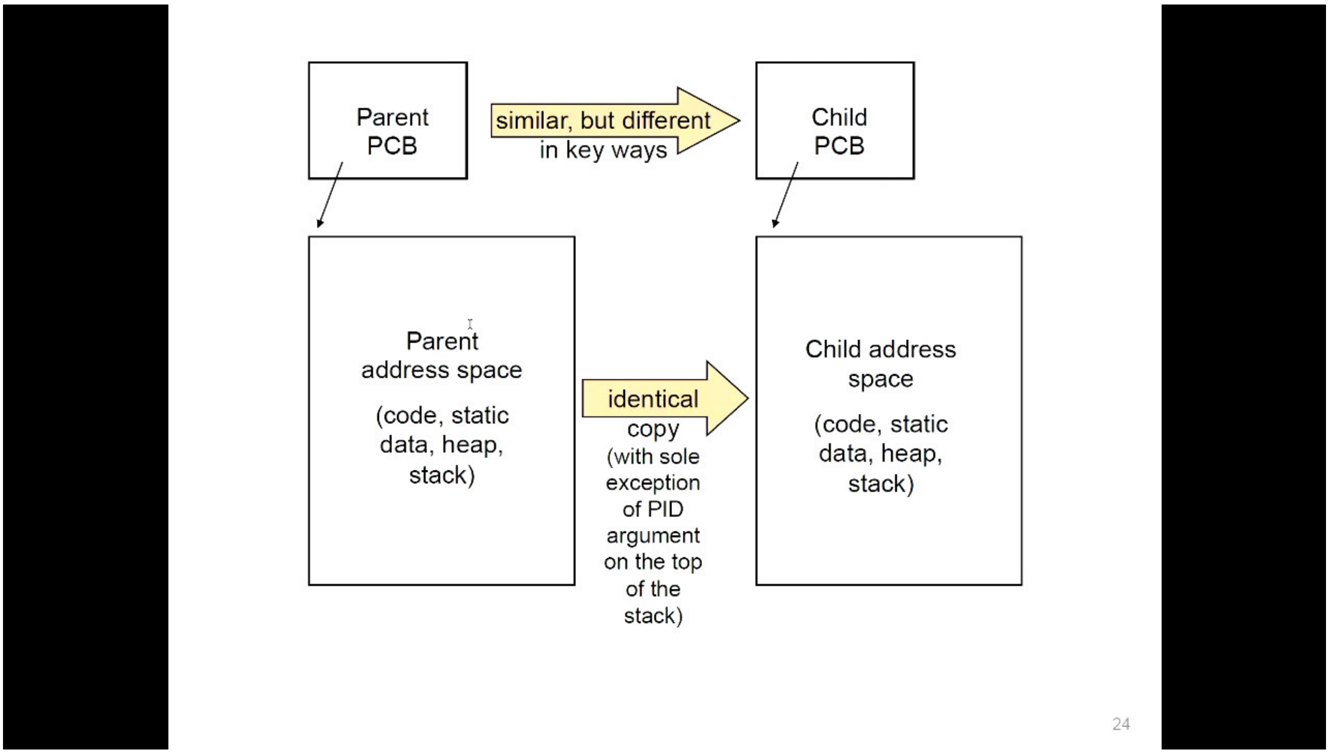
- New processes are created by existing processes
  - creator is called the *parent*;
  - created process is called the *child*;
    UNIX: do `ps -ef`, look for PPID field
  - what creates the first process, and when?
    on UNIX, this first process is init;
    on many Linux distributions, this is SystemD or Runit (on Void).

Process creation semantics

- (Depending on the OS) child processes inherit certain attributes of the parent. E.g.
  - Open file table: implies `stdin`/`stdout`/`stderr`;
  - On some systems, resource allocation to parent may be divided among children.
- (In Unix) when a child is created, the parent may either wait for the child to finish, or continue in parallel.

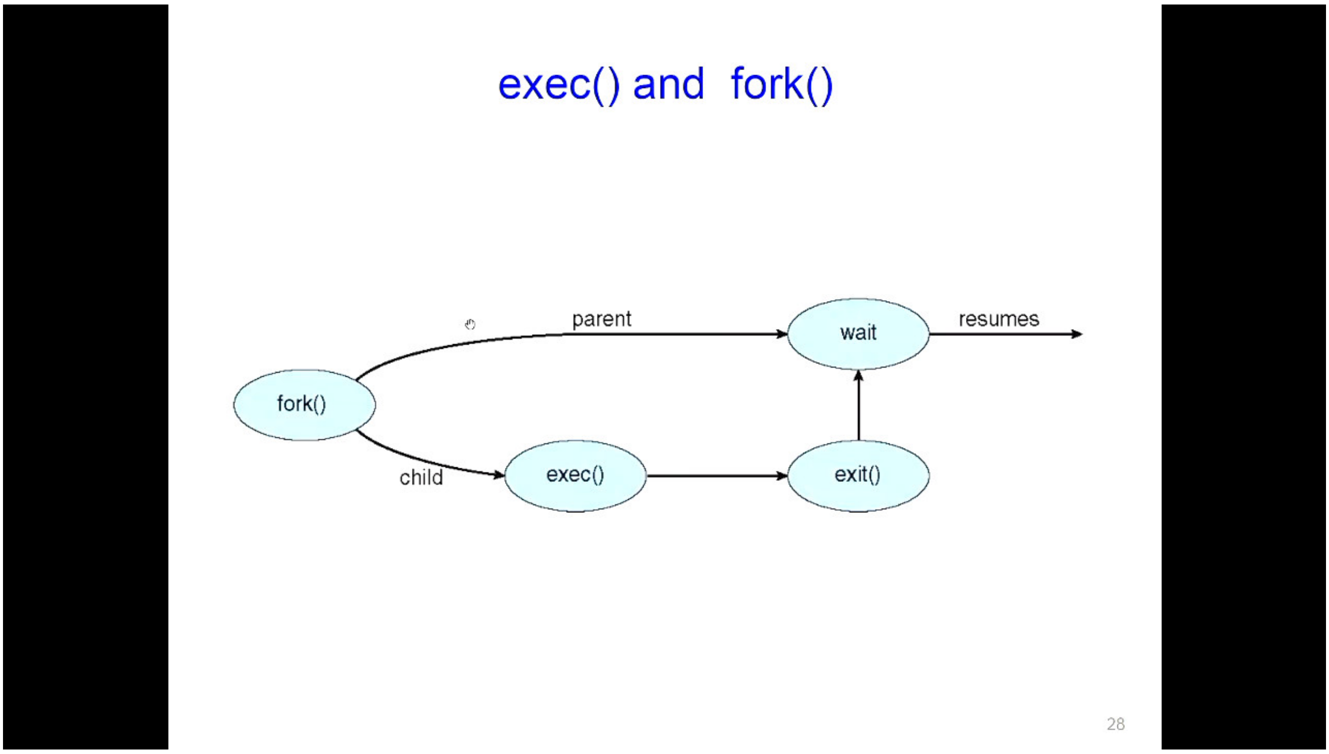
UNIX process creation details

- UNIX process creation through `fork` system call
  - creates and initializes a new PCB
    * initializes kernel resources of new process with resources of parent (e.g. open files)
    * initializes PC, SP to be same as parent.
  - creates a new address space
    * initialises new address space with a copy of the entire contents of the address space of the parent
  - places new PCB on the ready queue.
- the `fork` system call "returns twice"
  - once into the parent, and once into the child
    * returns the child's PID to the parent
    * returns `0` to the child
- `fork` = "clone me".
  The return value is used to determine whether we're the clone or the original.



`exec` **v.s.** `fork`

- Q: So how do we start a new program, instead of just forking the old program?
- A: First `fork`, then `exec`.
- `exec`
  - stops the current process
  - loads program 'prog' into the address space (i.e. overwrites the existing process image)
  - initialises hardware context, args for new program
  - places PCB onto ready queue
  - *does not create a new process!*



Method 1: `vfork`

- `vfork` is the older (now uncommon) of the two approaches.
- Instead of "child's address space is a copy of the parent's", the semantics are "child's address space *is* the parent's",
  - with a "promise" that the child won't modify the address space before doing an `execve`.
  - When `execve` is called, a new address space is created and it's loaded with the new executable.
  - Parent is blocked until `execve` is executed by child.
  - Saves wasted effort of duplicating parent's address space.

Method 2: copy-on-write

- Retains the original semantics, but copies "only what is necessary" rather than the entire address space.
- On `fork`:
  - Create a new address space
  - Initialise page tables with same mappings as the parent's (i.e. they both point to the same physical memory).
    * (No copying of address space contents have occurred at this point — with the sole exception of the top page of the stack.)
  - Set both parent and child page tables to make all pages read-only
  - If either parent or child writes to memory, an exception occurs.
  - When exception occurs, OS copies the page, adjusts page tables, etc.

3.5 Summary

- Process

- PCB
- Process state
- Context switch
- Process creation and termination

4 Threads

4.1 Process vs Threads

What's *in* a process?

- A process consists of (at least):
  - An *address space*, containing
    - the code (instructions) for the running program
    - the data for the running program
  - *Thread state*, consisting of
    - The PC, indicating the next instruction
    - The SP, indicating the position on the stack
    - Other general purpose registers
  - A set of *OS resources*
    - Open files, network connections, sound channels, . . .
- Decompose . . .
  - address space
  - *thread of control* (stack, SP, PC, registers)
  - OS resources

Motivation

- Threads are about *concurrency* and *parallelism*
- One way to get concurrency and parallelism is to use multiple processes
  - The programs (code) of distinct processes are isolated from each other
- Threads are another way to get concurrency and parallelism
  - Threads *share a process* — same address space, same OS resources
  - Threads have private stack, CPU state — are schedulable

What's needed?

- In many cases
  - Everybody wants to run the same code
  - Everybody wants to access the same data
  - Everybody has the same privileges
  - Everybody uses the same resources (open files, network connections, etc.)
- But you'd like to have multiple hardware execution states:
  - an execution stack and SP
    - traces state of procedure calls made
  - the PC, indicating the next instruction
  - a set of general-purpose processor registers and their values

How could we achieve this?

- Given the process abstraction as we know it:
  - for several processes
  - cause each to *map* to the *same* physical memory to share data (`shmget`),
- This is really inefficient
  - space: PCB, page tables, etc.
  - time: creating OS structures, fork/copy address space, etc.

Can we do better?

- Key idea:
  - separate the concept of a *process* (address space, OS resources)
  - ...from that of a minimal *thread of control* (execution state: stack, SP, PC, registers),
- This execution state is usually called a *thread*, or a *lightweight process.*

Threads and processes

- Most modern OS*s* support two entities:
  - the *process*, which defines the address space and general process attributes (such as open files, etc.)
  - the *thread*, which defines a sequential execution stream within a process.
- A thread is bound to a single process / address space
  - address spaces, however, can have multiple threads executing within them
  - sharing data between threads is cheap: all see the same address space
  - creating threads is cheap, too!
- *Threads become the unit of scheduling*
  - processes / address spaces are just *containers* in which threads execute.

Single and Multi-threaded Processes

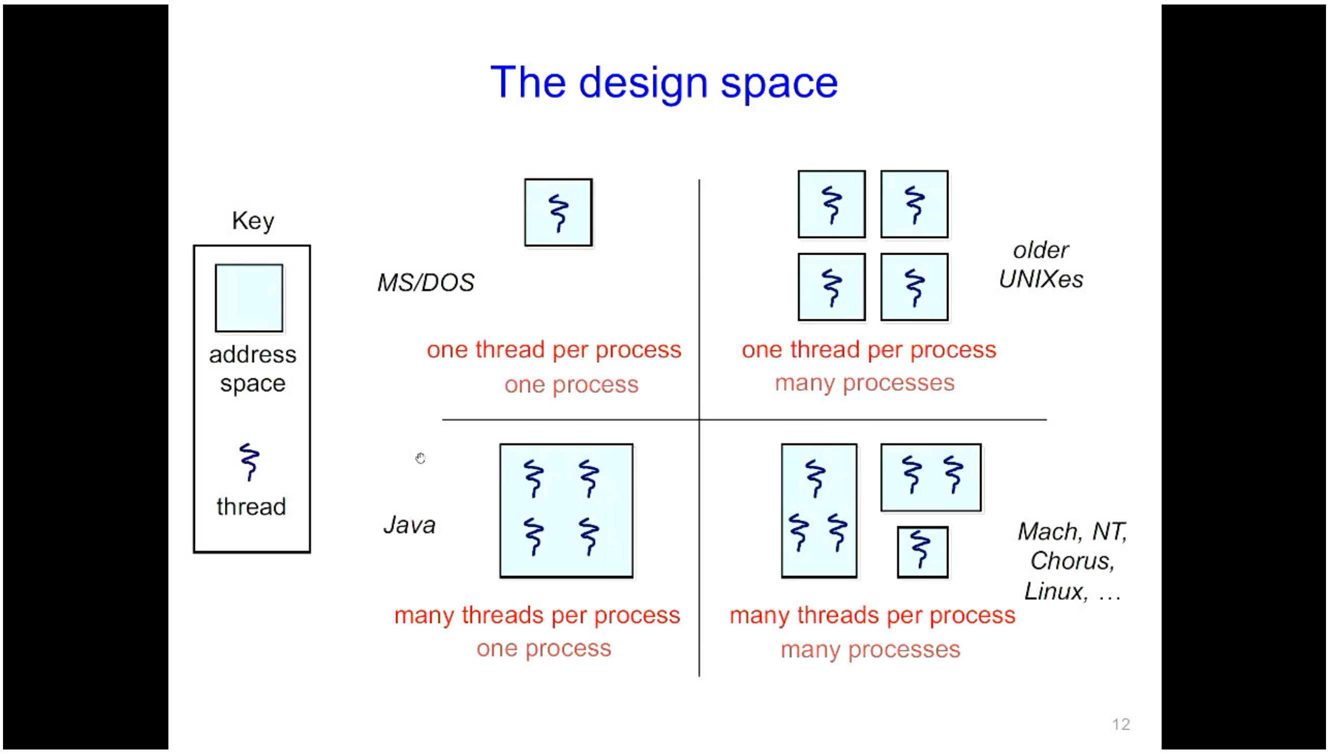
- Different threads in the same process have separate registers and stacks.
- This is cheaper than duplicating the instructions and PCB etc., as required by having multiple processes.

4.2 Concurrency

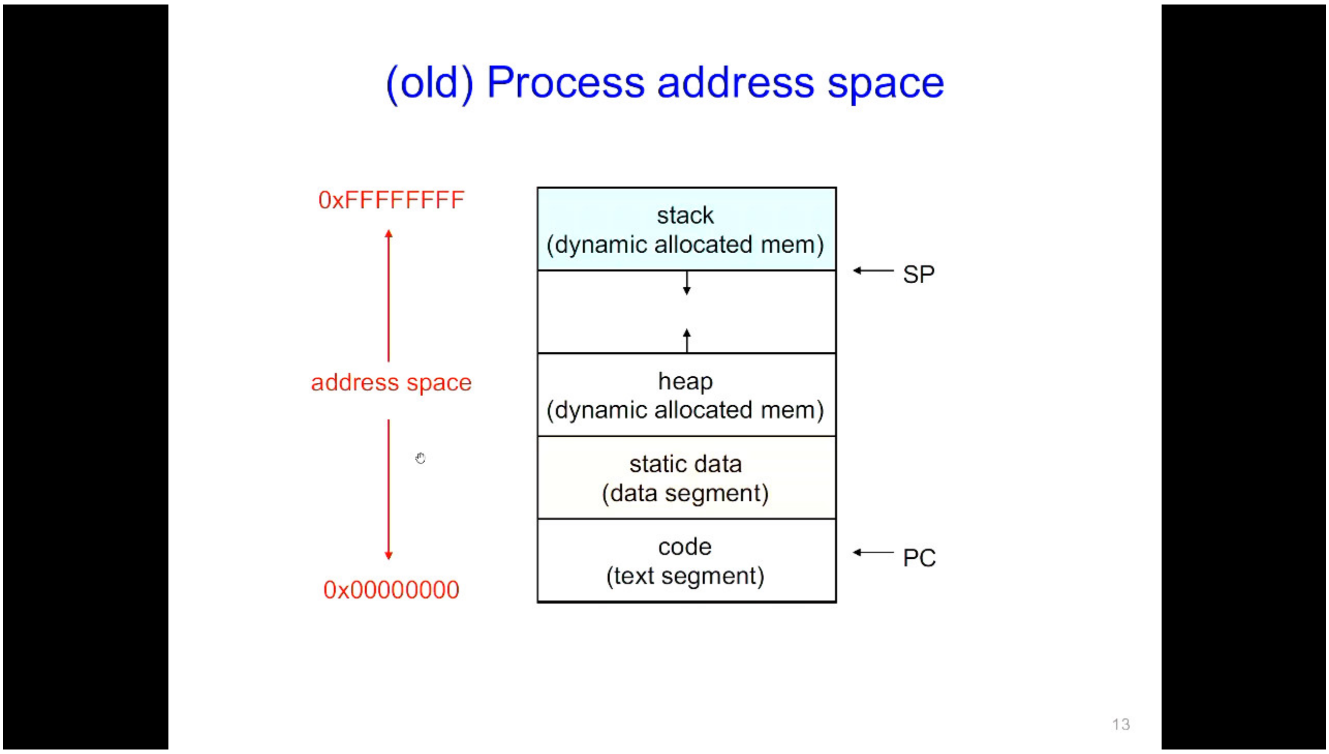
Communication

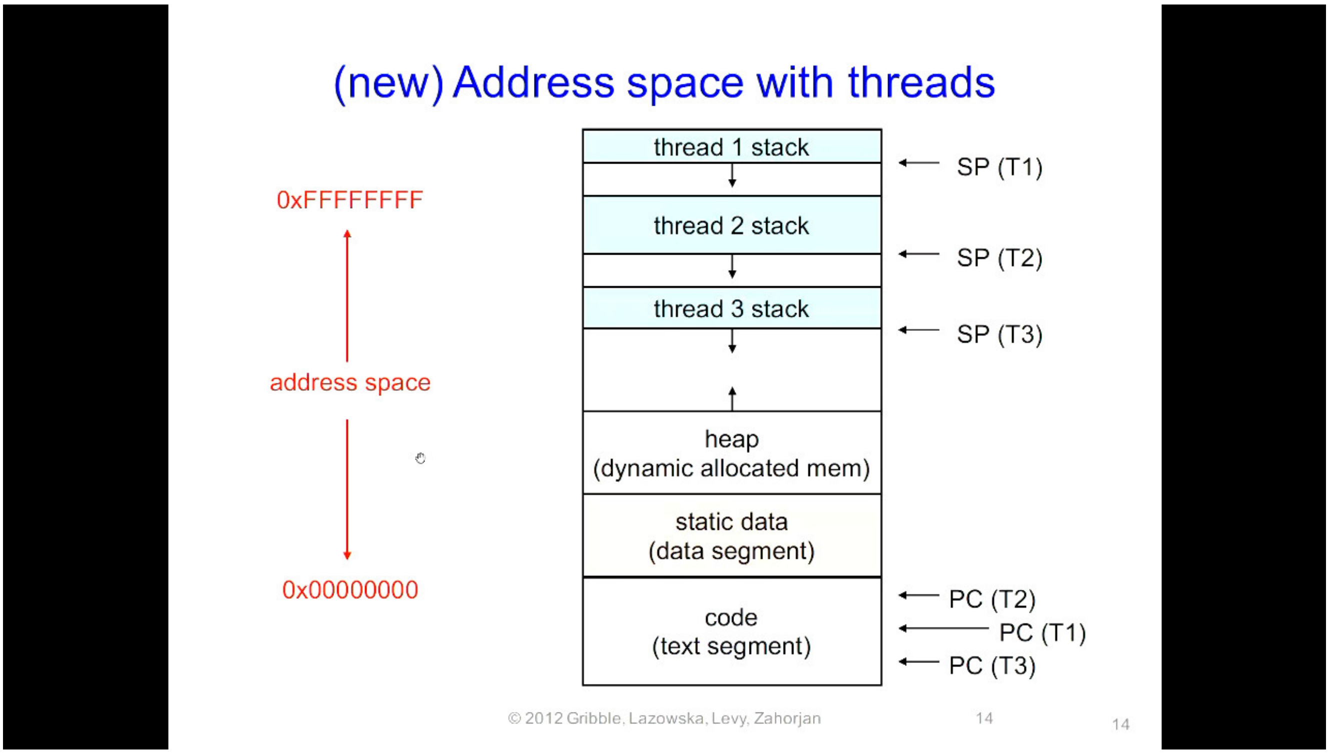
- Threads are concurrent executions sharing an address space (and some OS resources)
- Address spaces provide isolation
  - If you can't name an object, you can't read or write to it
- Hence, communicating between processes is expensive
  - Must go through the OS to move data from one address space to another
- Because threads are in the same address space, communication is simple/cheap
  - Just update a shared variable!

The design space



Process address space





4.3 Design space of process/threads

Process/thread separation

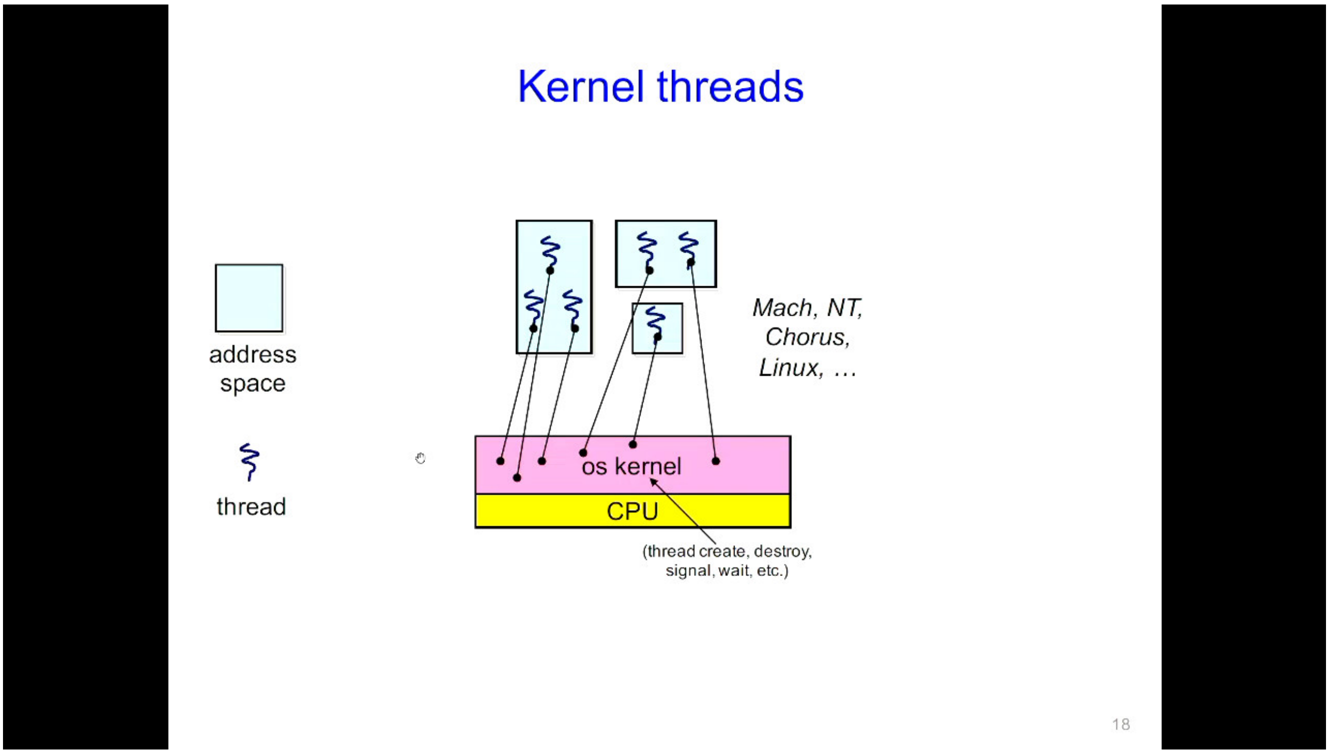
- Concurrency (multi-threading) is useful for:
  - handling concurrent events (e.g. web servers and clients)
  - building parallel programs (e.g. matrix multiply, ray tracing)
  - improving program structure (the Java argument),
- Multi-threading is useful even on a uniprocessor
  - even though only one thread can run at a time
- Supporting multi-threading — that is, separating the concept of a *process* (address space, files, etc.) from that of a minimal *thread of control* (execution state), is a big win
  - creating concurrency does not require creating new processes
  - "faster / better / cheaper"

4.4 Kernel threads

Where do threads come from?

- Natural answer: the OS is responsible for creating/managing threads
  For example, the kernel call to create a new thread would
  - allocate an execution stack within the process address space
  - create and initialize a *Thread Control block*
    (SP, PC, register values)
  - stick it on the ready queue
- We call these *kernel threads*
  There is a "thread name space"

- Thread IDs (TIDs)
- TIDs are integers



Kernel Threads

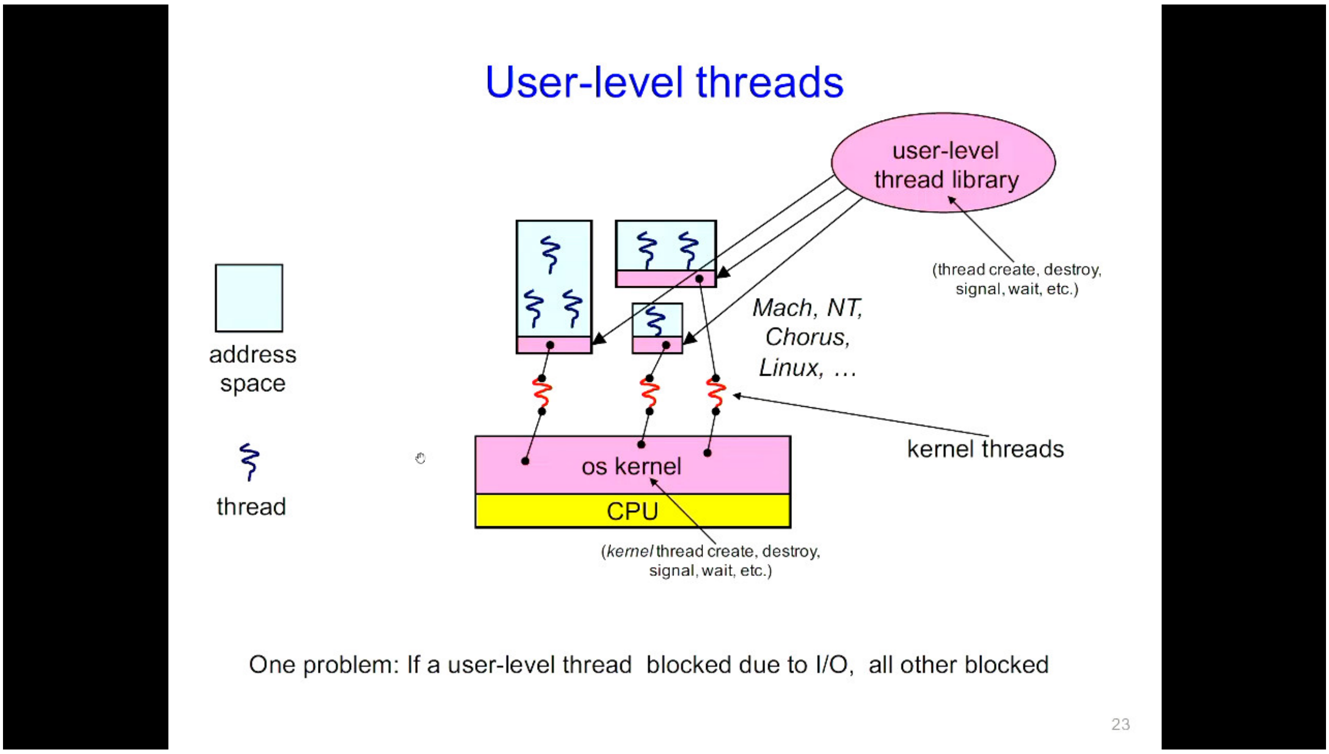
- OS now manages threads *and* processes / address spaces
  - all thread operations are implemented in the kernel
  - OS schedules all of the threads in a system
    - if one thread in a process blocks (e.g. on I/O), the OS knows about it, and can run other threads from that process
    - possible to overlap I/O and computation *inside* a process
- Kernel threads are cheaper than processes
  - less state to allocate and initialise
- But, they're still pretty expensive for fine-grained use
  - orders of magnitude more expensive than a procedure call
  - thread operations are all *system calls*
    - context switch
    - argument checks
  - must maintain kernel state for each thread

4.5 User-level threads

Cheaper alternative

- There is an alternative to kernel threads
- Threads can also be managed at the user level (within the process)

- a library linked into the program manages the threads
  * the thread manager doesn't need to manipulate address spaces (which only the kernel can do)
  * threads differ (roughly) only in hardware contexts (PC, SP, registers), which can be manipulated by user-level code
  * the *thread package* multiplexes user-level threads on top of kernel threads
  * each kernel thread is treated as a *virtual processor*
- we call these *user-level threads*



User-level threads

- User-level threads are small and fast
  - managed entirely by user-level library (e.g. `pthreads`)
  - each thread is represented by a PC, registers, a stack, and a small *thread control block* (TCB)
  - creating a thread, switching between threads, and synchronising threads are done *via procedure calls*
    * no kernel involvement necessary!
- User-level thread operations can be 10–100x faster than kernel threads as a result.

User-level thread implementation

- The OS schedules the kernel thread
- The kernel thread executes user code, including the thread support library and its associated thread scheduler
- The thread scheduler determines when a user-level thread runs
  - it uses queues to keep track of what threads are doing: run, ready, wait

    * just like the OS and processes
    * but, implemented at user-level as a library

Thread context switch

- Very simple for user-level threads:
  - save context of currently running thread
    * push CPU state onto thread stack
  - restore context of the next thread
    * pop CPU state from next thread's stack
  - return as the new thread
    * execution resume at PC of next thread
  - Note: no changes to memory mapping required
- This is all done in assembly language
  - it works at the level of the procedure calling convention

How to keep a user-level thread from hogging the CPU?

- Strategy 1: force everyone to cooperate
  - a thread willingly gives up the CPU by calling `yield`
  - `yield` calls into the scheduler, which context switches to another ready thread
  - what happens if a thread never calls `yield`?
- Strategy 2: use presumption
  - scheduler requests that a timer interrupt be delivered by the OS periodically
    * usually delivered as a UNIX signal (`man signal`)
    * signals are just like software interrupts, but delivered to user-level by the OS instead of delivered to the OS by hardware
  - at each timer interrupt, scheduler gains control and context switches as appropriate.

What if a thread tries to do I/O

- The kernel thread "powering" it is lost for the duration of (synchronous) I/O operation!
  - The kernel thread blocks in the OS, as always
  - It maroons with it the state of the user-level thread
- Could have one kernel thread "powering" each user-level thread
  - "common case" operations (e.g. synchronisation) would be quick
- Could have a limited-size "pool" of kernel threads "powering" all the user-level threads in the address space
  - the kernel will be scheduling these threads, obliviously to what's going on at user-level.

4.6 Summary

- Multiple threads per address space
- Kernel threads are much more efficient than processes, but still expensive
  - all operations require a kernel call and parameter validation
- User-level threads are:
  - much cheaper and faster
  - great for common-case operations
    - creation, synchronisation, destruction
  - can suffer in uncommon cases due to kernel obliviousness
    - I/O
    - pre-emption of a lock-holder

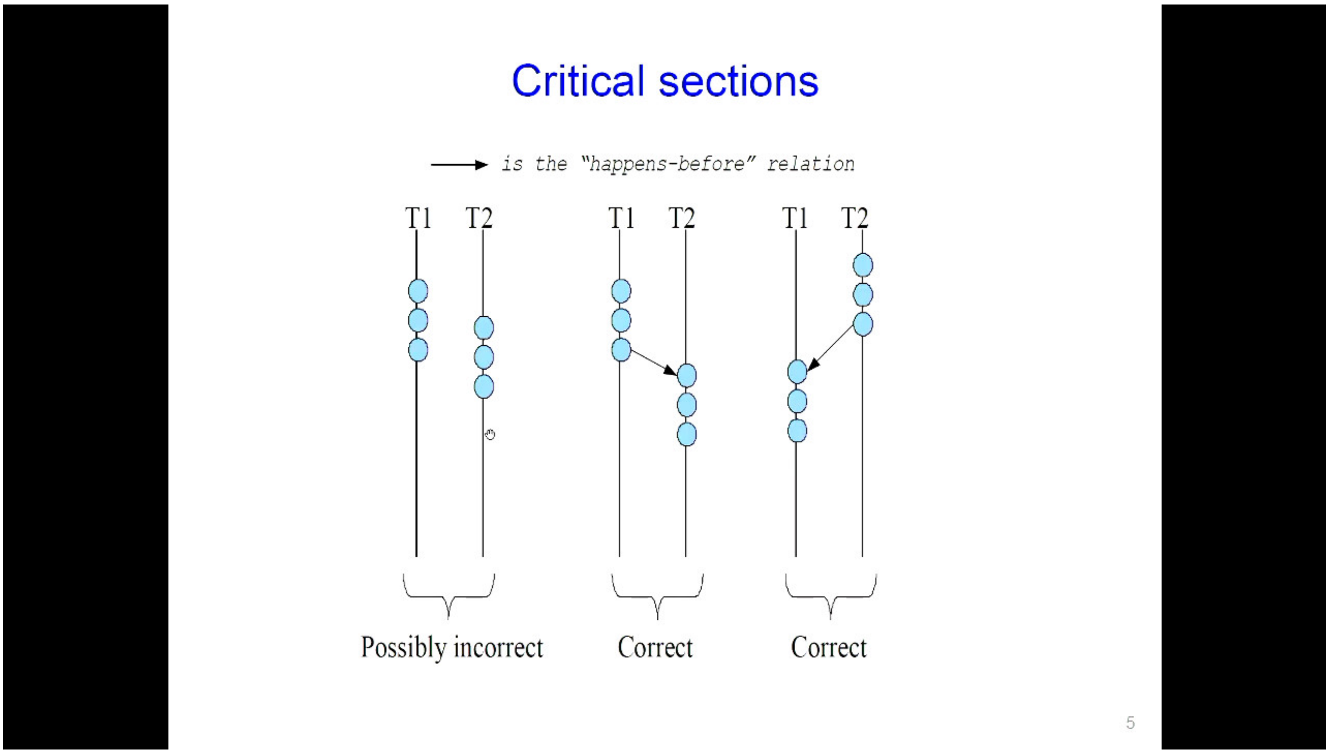
5 Synchronisation

Temporal relations

- User view of parallel threads
  - Instructions executed by a single thread are totally ordered
    - $A < B < C < \ldots$
  - In absence of *synchronisation*:
    - instructions executed by distinct threads must be considered unordered / simultaneous
    - Not $X < X'$, and not $X' < X$
- Hardware largely supports this

Critical sections / mutual exclusion

- Sequences of instructions that may get incorrect results if executed simultaneously are called *critical sections*.
- *Race condition* results depend on timing
- *Mutual exclusion* means "not simultaneously"
  - $A < B$ or $B < A$
  - We don't care which
- Forcing mutual exclusion between two critical section executions
  - is sufficient to ensure correct execution
  - guarantees ordering.



When do critical sections arise?

- One common pattern:
  - read-modify-write of
  - a shared value (variable)
  - in code that can be executed by concurrent threads
- Shared variable:
  - Global and heap-allocated variables
  - NOT local variables (which are on the stack)

Race conditions

- A program has a *race condition* (data race) if the result of an execution depends on timing (i.e. it is non-deterministic)
- Typical symptoms
  - I run it on the same data, and sometimes it prints 0 and sometimes 4
  - I run it on the same data, and sometimes it prints 0 and sometimes crashes

Correct critical section requirements

- *Mutual exclusion*
  At most one thread is in the critical section.
- *Progress*
  If thread T is outside the critical section, then T cannot prevent thread S from entering the critical section.
- *Bounded waiting* (no *starvation*)
  If thread T is waiting on the critical section, then T will eventually enter the critical section (assumes threads eventually leave critical sections).

- *Performance*
  The overhead of entering and exiting the critical section is small with respect to the work being done within it.

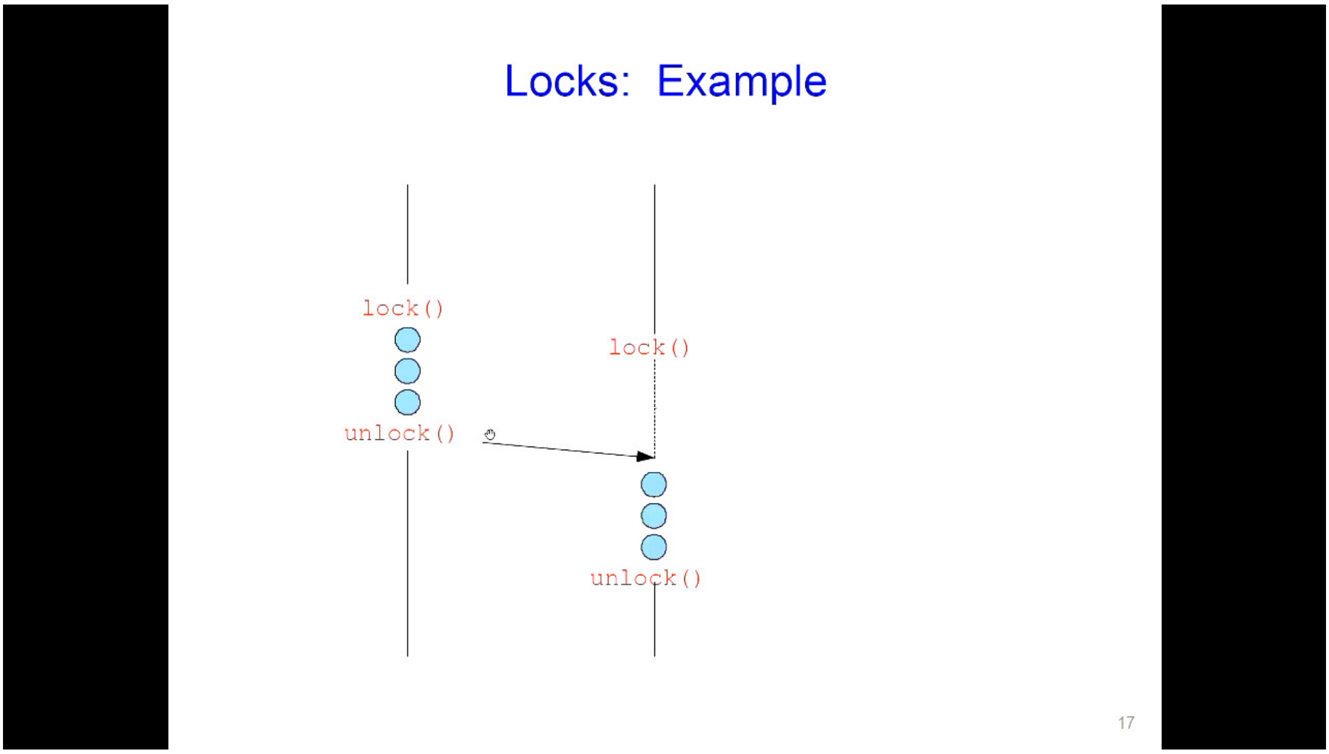
Mechanisms for building critical sections

- Spinlocks
  - primitive, minimal semantics — used to build others
- Semaphores (and non-spinning locks)
  - basic, easy to understand, somewhat hard to program with
- Monitors
  - higher level, requires language support, implicit operations
  - easier to program with; Java "`synchronised`", for example
- Messages
  - Simple model of communication and synchronisation based on (atomic) transfer of data across a channel
  - direct application to distributed systems

5.1 Locks

Locks

- A lock is a memory object with two operations:
  - `acquire`: obtain the right to enter the critical section
  - `release`: give up the right to be in the critical section
- `acquire` *prevents the progress of the thread until the lock can be acquired.*
- Note: terminology varies: acquire/release, lock/unlock



Acquire/release

- Threads pair up calls to `acquire` and `release`
  - between `acquire` and `release`, the thread *holds* the lock
  - `acquire` does not return until the caller "owns" (holds) the lock
    * at most one thread can hold a lock at a time
- What happens if the calls aren't paired
  - I acquire, but neglect to release?
- What happens if the two threads acquire different locks
  - I think that access to a particular shared data structure is mediated by lock A, and you think it's mediated by lock B?
- What is the right granularity of locking?

5.2 Spinlocks

Spinlocks

- How do we implement spinlocks? Here's one attempt:

```
struct lock_t {
    int held = 0;
}
void acquire(lock) {
    while (lock->held);
    lock->held = 1;
}
void release(lock) {
    lock->held = 0;
}
```

- Race condition in acquire.

Implementing spinlocks

- Problem is that implementation of spinlocks has critical sections, too!
  - the acquire/release must be *atomic*
  - compiler can hoist code that is invariant
- Need help from the hardware
  - atomic instructions
    test-and-set, compare-and-swap, ...

Spinlocks: Hardware Test-and-Set

- CPU provides the following as *one atomic instruction*:

```
bool test_and_set(bool *flag) {
    bool old = *flag;
    *flag = True;
    return old;
}
```

- This is a single *atomic* instruction

Implementing spinlocks using Test-and-Set

- So, to fix our broken spinlocks:

```
struct lock{
    int held = 0;
}
void acquire(lock) {
    while (test_and_set(&lock->held));
}
void release(lock) {
    lock->held = 0;
}
```

- *mutual exclusion?* (at most one thread in the critical section)
- *progress?* (T outside cannot prevent S from entering)
- *bounded waiting?* (waiting T will eventually enter)
- *performance?* (low overhead (modulo the spinning part...))