Operating Systems Lecture Notes

Anthony Catterwell

April 15, 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.

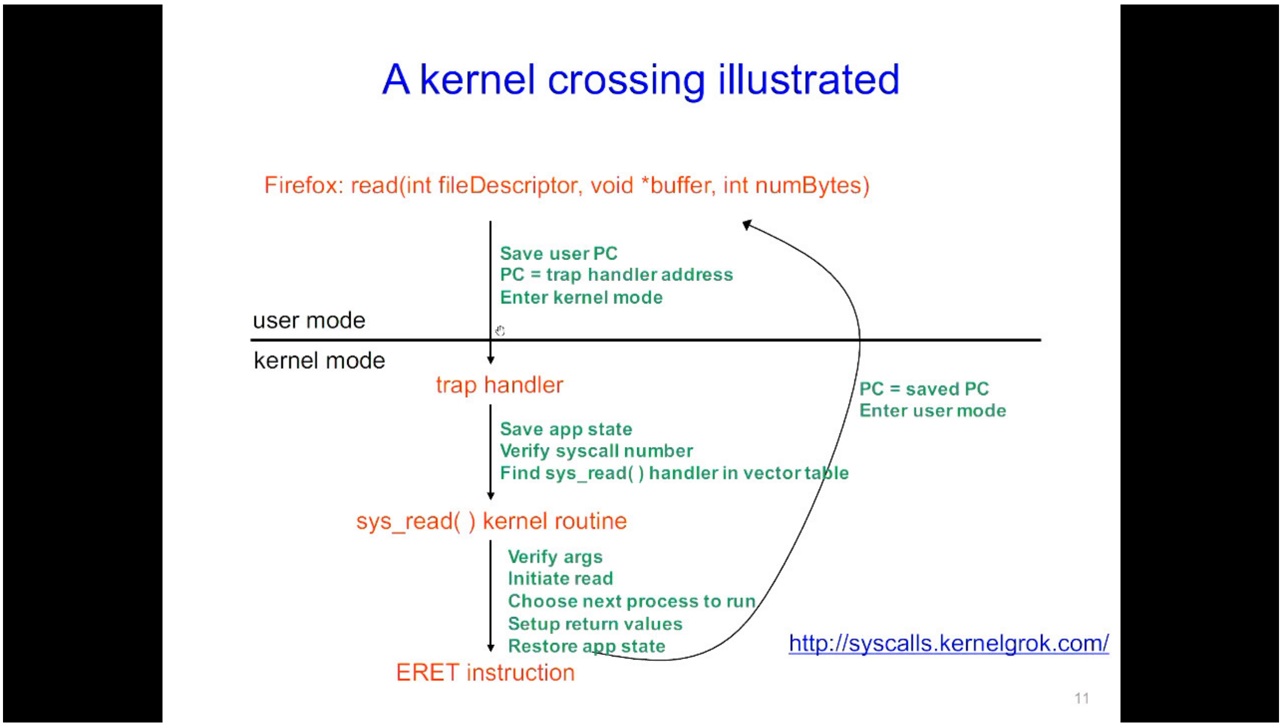
Crossing protection boundaries

- 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.



A kernel crossing illustrated

Firefox: read(int fileDescriptor, void *buffer, int numBytes)

Save user PC
PC = trap handler address
Enter kernel mode

user mode

kernel mode

trap handler

Save app state
Verify syscall number
Find sys_read( ) handler in vector table

sys_read( ) kernel routine

Verify args
Initiate read
Choose next process to run
Setup return values
Restore app state

ERET instruction

PC = saved PC
Enter user mode

http://syscalls.kernelgrok.com/

11

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

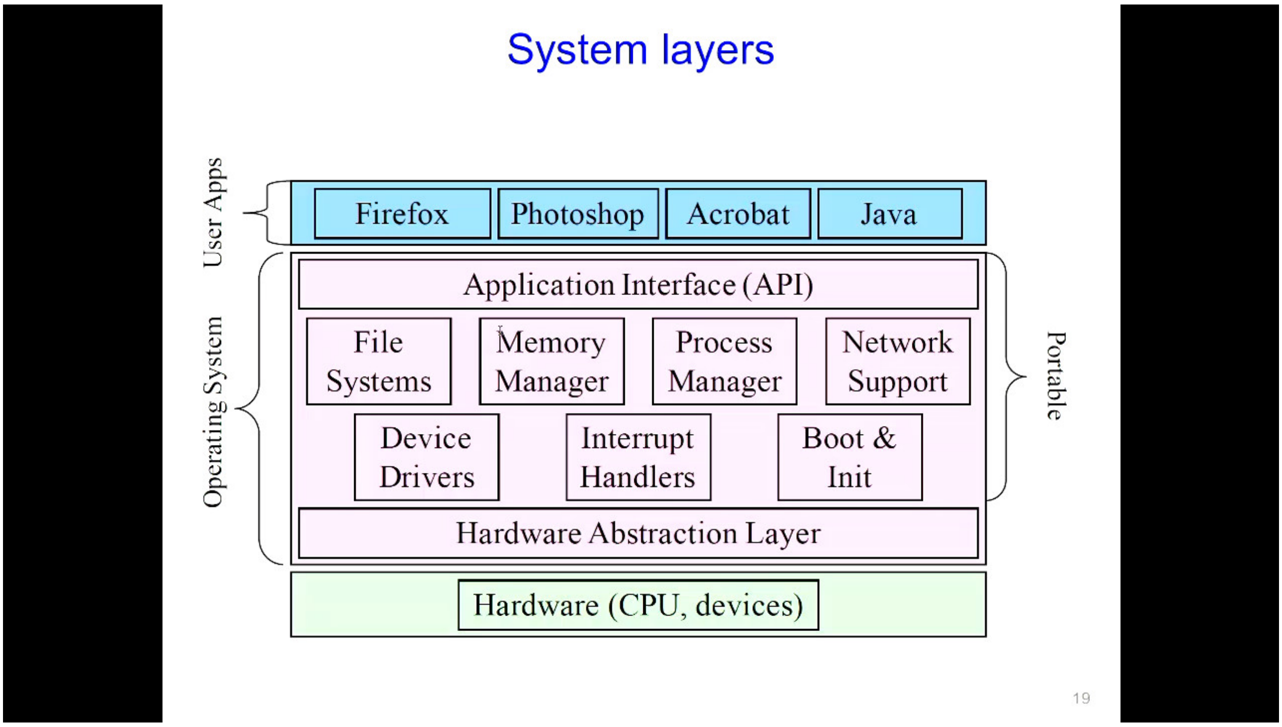
Operating system design and implementation

- 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



System layers

User Apps

Firefox | Photoshop | Acrobat | Java

Operating System

Application Interface (API)

File Systems | Memory Manager | Process Manager | Network Support

Device Drivers | Interrupt Handlers | Boot & Init

Hardware Abstraction Layer

Portable

Hardware (CPU, devices)

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

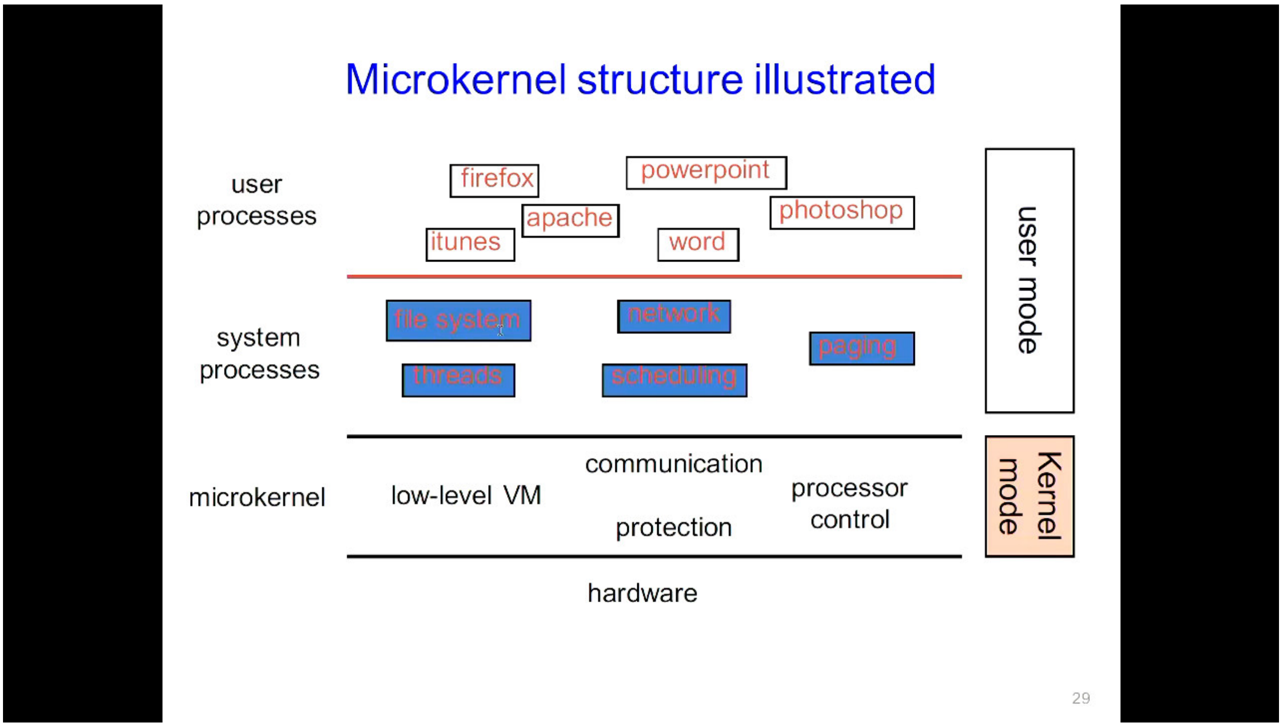
- Disjunction between model and reality: systems modelled as layers, but not really built that way.

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)



Microkernel structure illustrated

user processes

firefox

powerpoint

apache

photoshop

itunes

word

user mode

system processes

file system

network

paging

threads

scheduling

microkernel

low-level VM

communication

protection

processor control

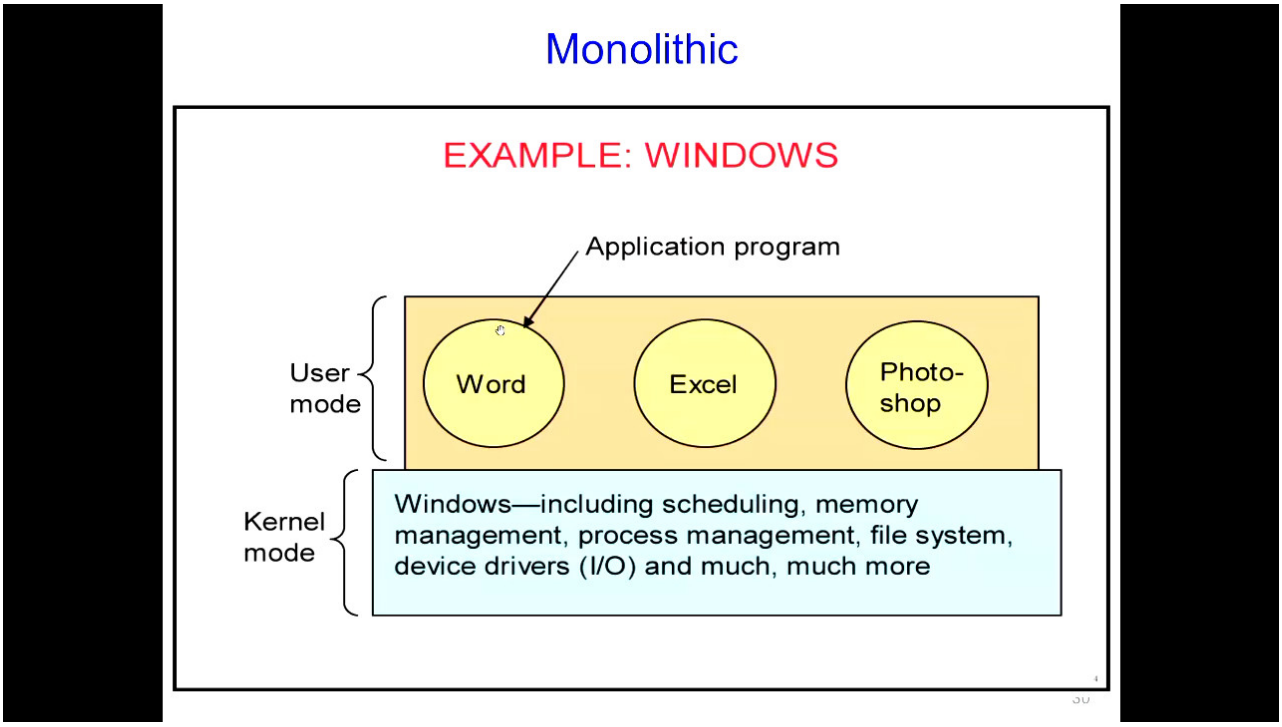
Kernel mode

hardware

29

Comparison of OS structures

Windows

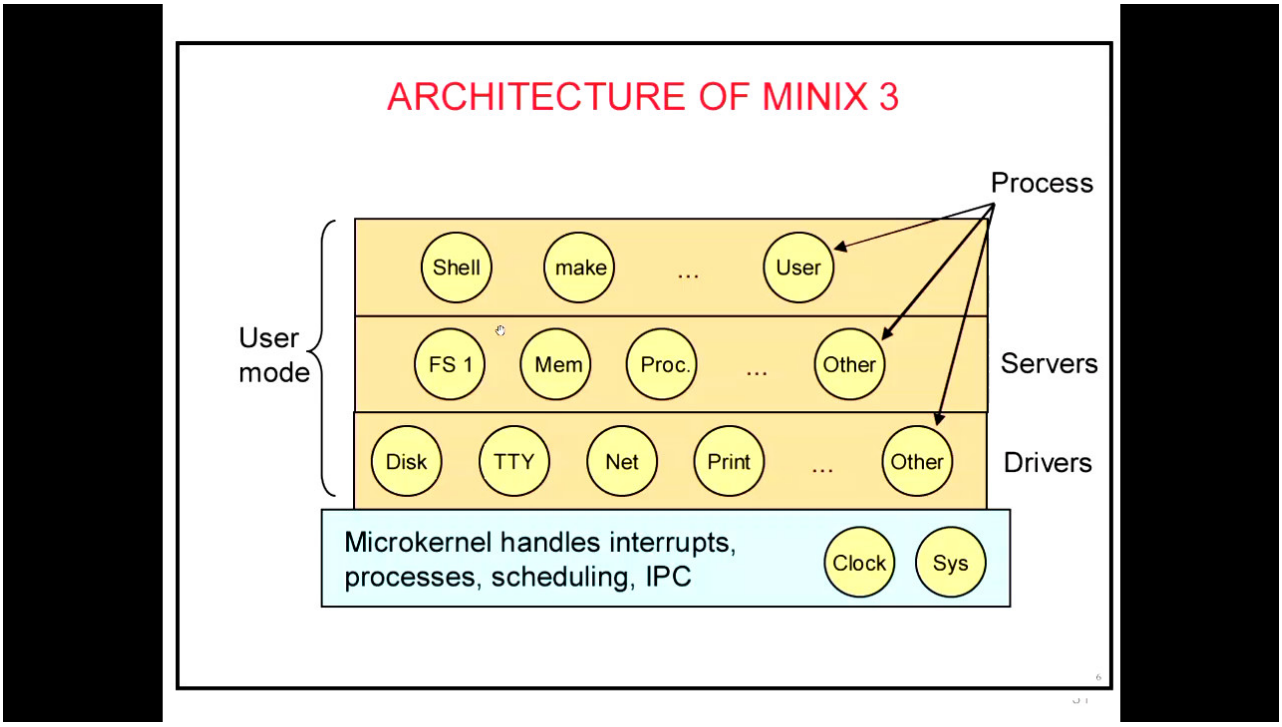


Monolithic

EXAMPLE: WINDOWS

Windows—including scheduling, memory management, process management, file system, device drivers (I/O) and much, much more

MINIX 3



ARCHITECTURE OF MINIX 3

Microkernel handles interrupts, processes, scheduling, IPC

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

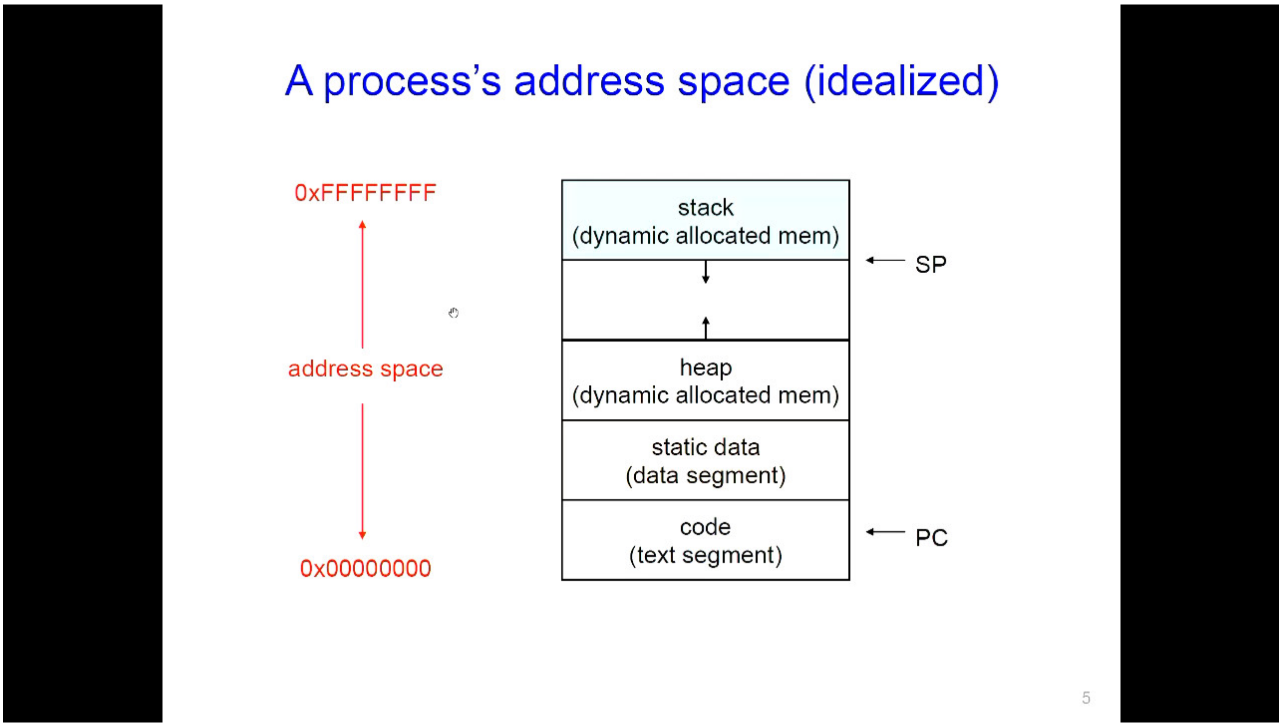
3.1 Process

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).



A process's address space (idealized)

0xFFFFFFFF

address space

0x00000000

stack
(dynamic allocated mem)

heap
(dynamic allocated mem)

static data
(data segment)

code
(text segment)

SP

PC

5

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

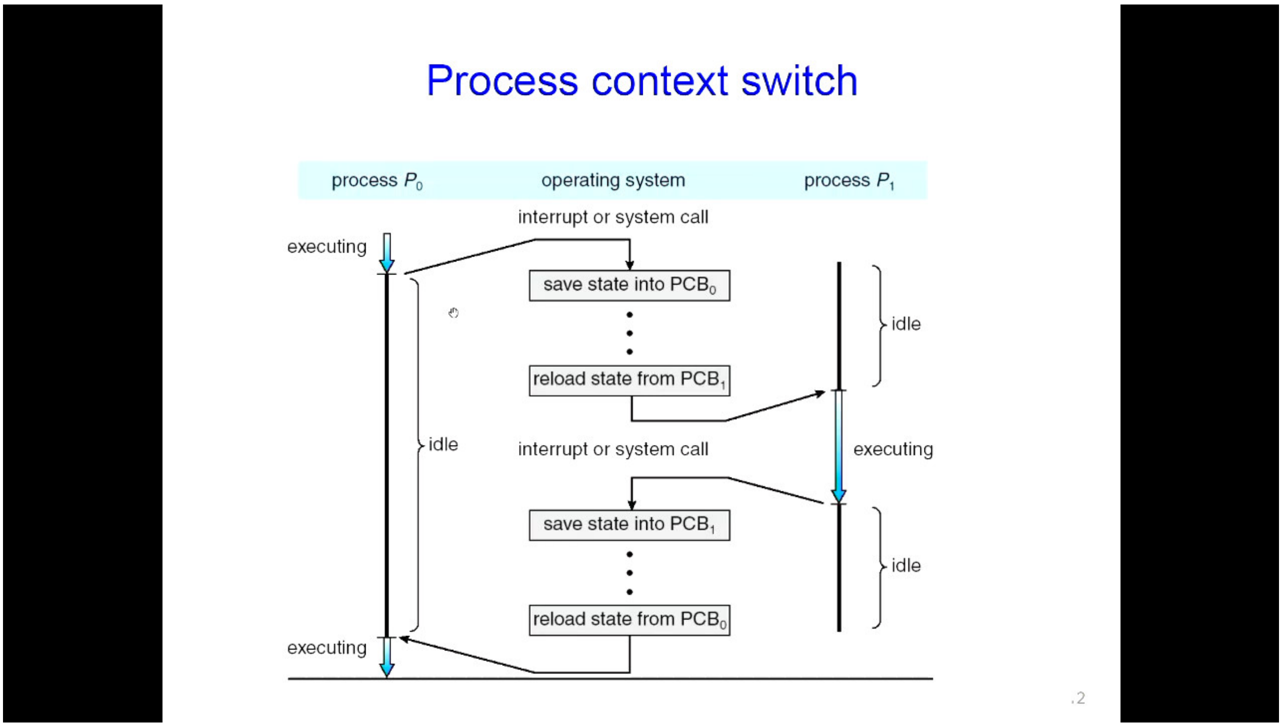
  - address space info
  - UNIX user id, group id
  - scheduling priority
  - accounting info
  - pointers for state queues
- In Linux:
  - defined in `task_struct (include/linux/sched.h)`
  - Over 95 fields!

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 context switch

process P_0 operating system process P_1

interrupt or system call

executing

save state into PCB_0

idle

reload state from PCB_1

idle

interrupt or system call

executing

save state into PCB_1

idle

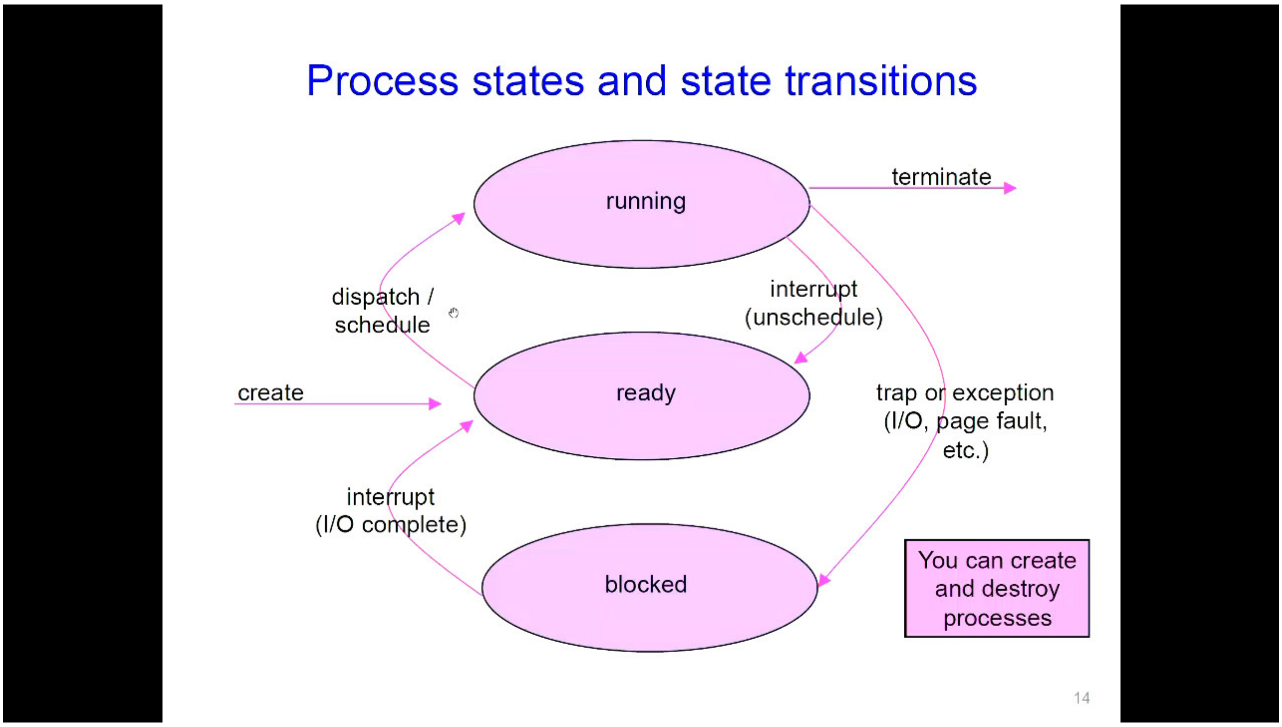
reload state from PCB_0

executing

.2

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?



Process states and state transitions

running

terminate

dispatch / schedule

interrupt (unschedule)

create

ready

trap or exception (I/O, page fault, etc.)

interrupt (I/O complete)

blocked

You can create and destroy processes

14

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

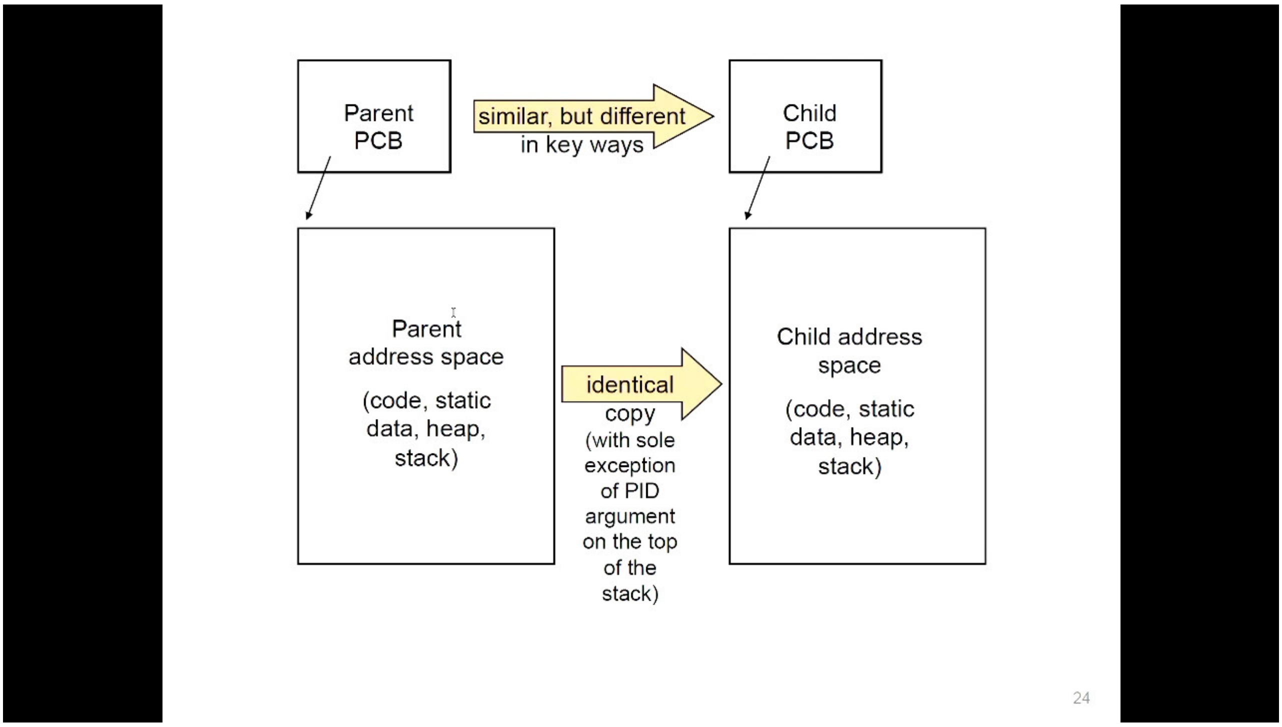
- New processes are created by existing processes
  - creator is called the *parent*;
  - created process is called the *child*;
UNIX: do `ps -ef`, look for PPID field
  - what creates the first process, and when?
on UNIX, this first process is init;
on many Linux distributions, this is SystemD or Runit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



Parent PCB

similar, but different in key ways

Child PCB

Parent address space

(code, static data, heap, stack)

identical copy (with sole exception of PID argument on the top of the stack)

Child address space

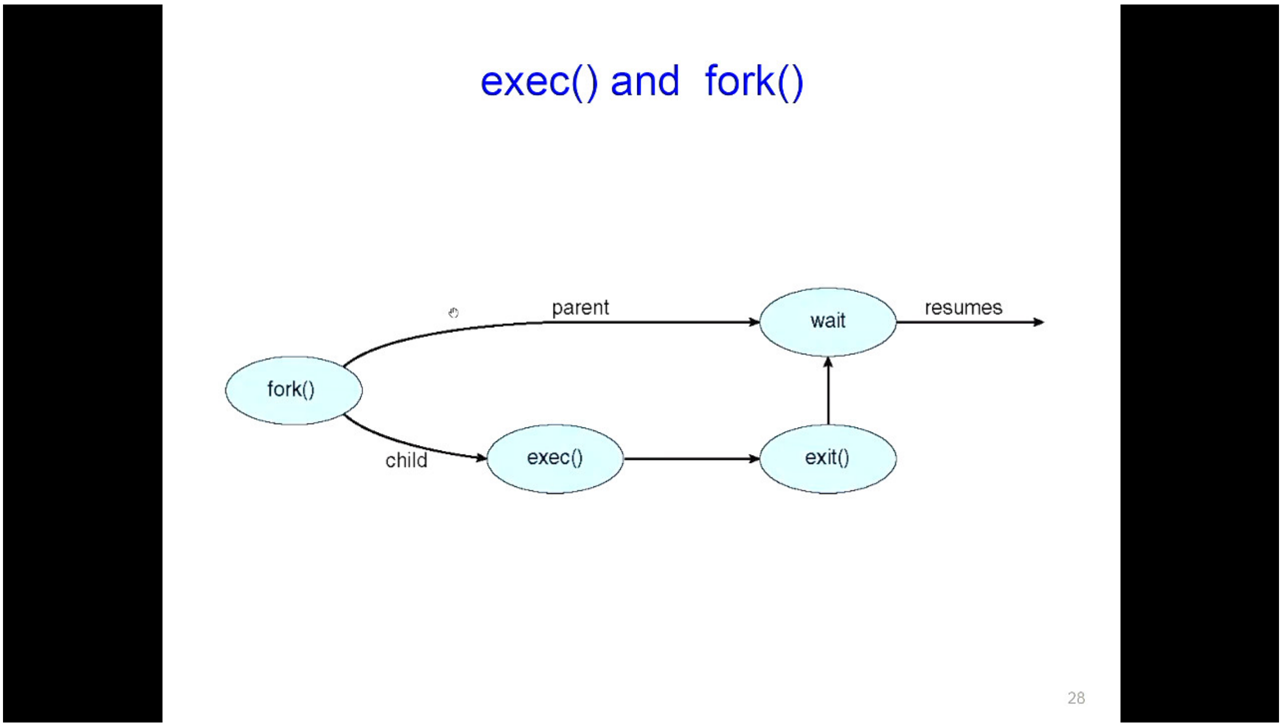
(code, static data, heap, stack)

24

`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!*



exec() and fork()

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

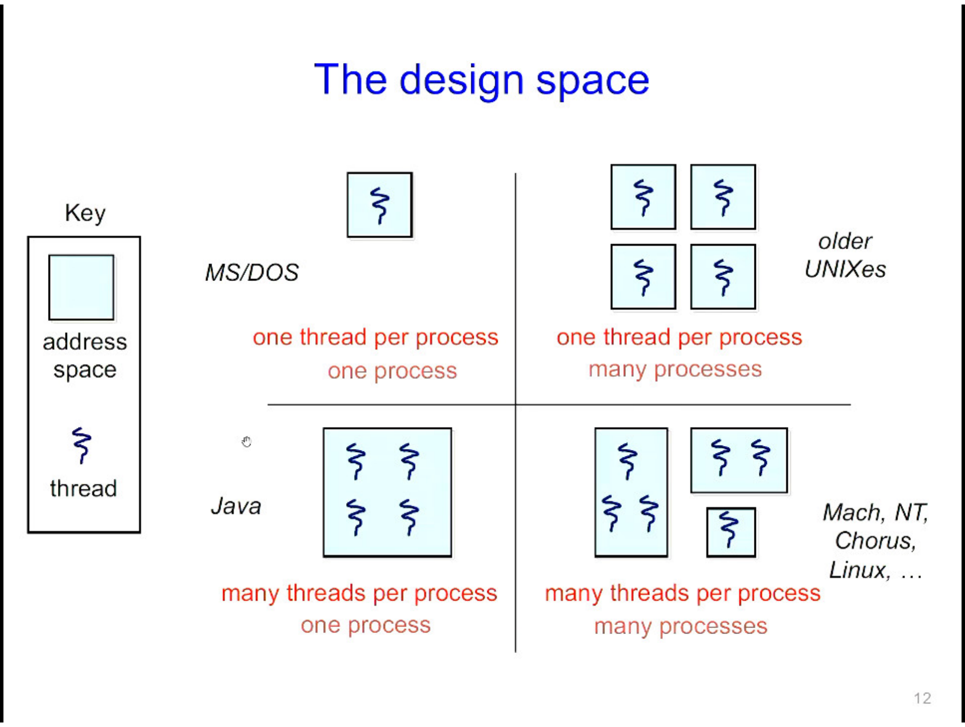
- Different threads in the same process have separate registers and stacks.
- This is cheaper than duplicating the instructions and PCB etc., as required by having multiple processes.

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



The design space

Key

address space

thread

MS/DOS

one thread per process
one process

older UNIXes

one thread per process
many processes

Java

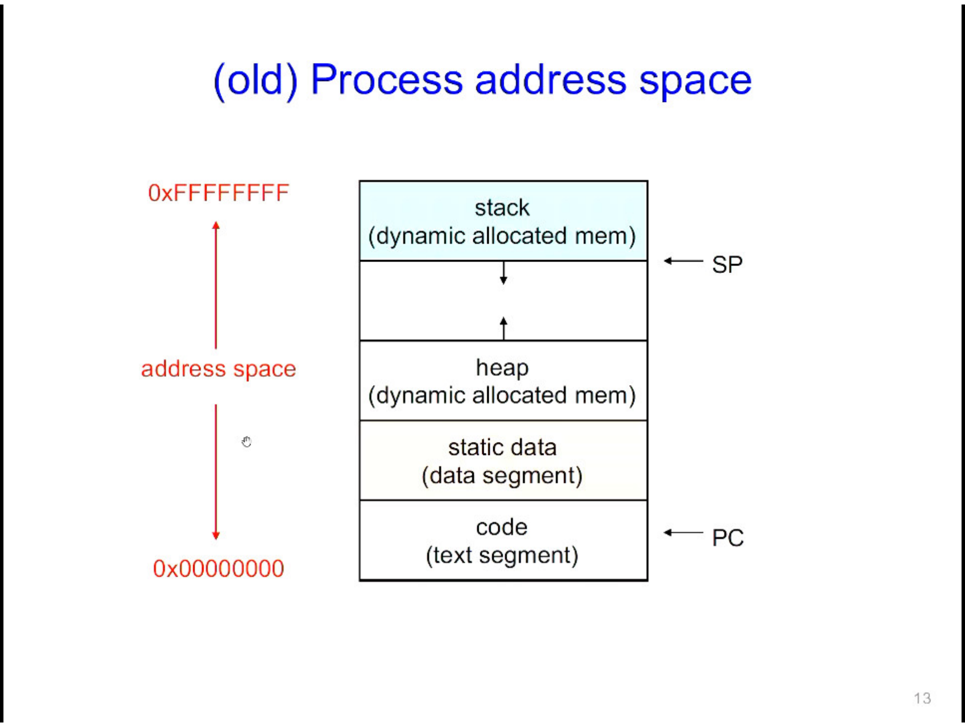
many threads per process
one process

Mach, NT, Chorus, Linux, …

many threads per process
many processes

12

Process address space



(old) Process address space

0xFFFFFFFF

address space

0x00000000

stack
(dynamic allocated mem)

SP

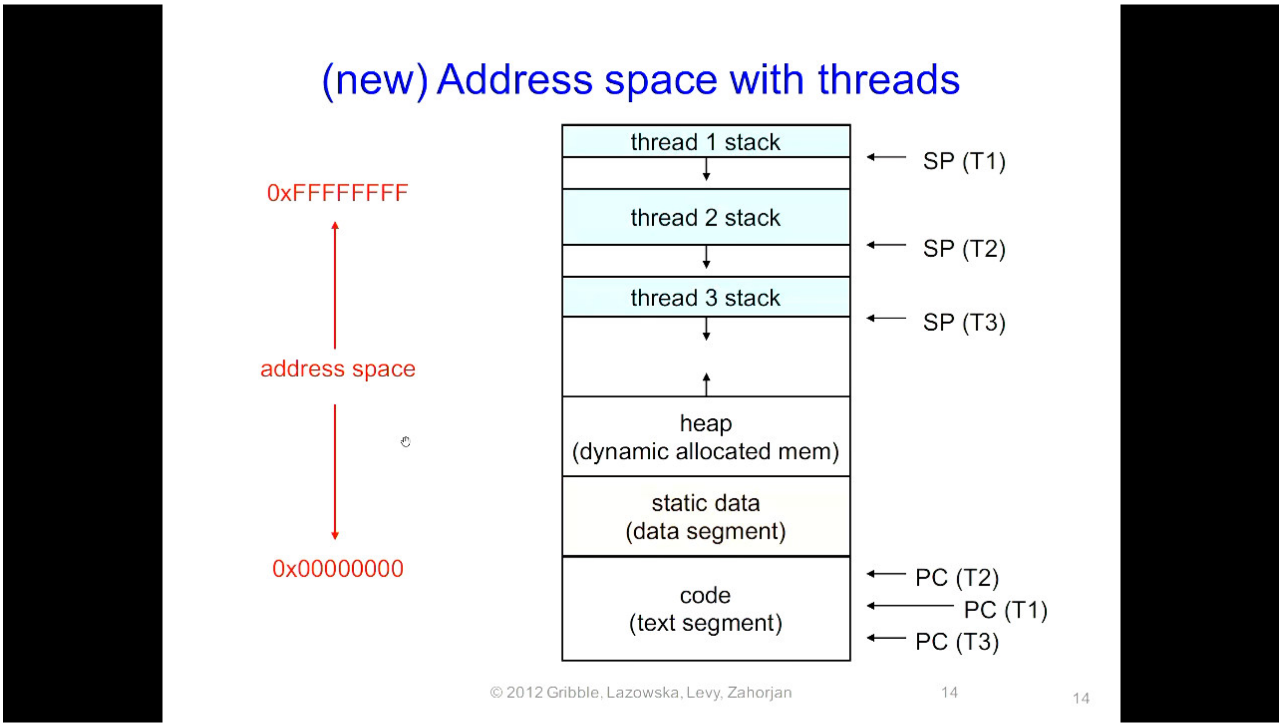
heap
(dynamic allocated mem)

static data
(data segment)

code
(text segment)

PC

13



(new) Address space with threads

© 2012 Gribble, Lazowska, Levy, Zahorjan 14

14

4.3 Design space of process/threads

Process/thread separation

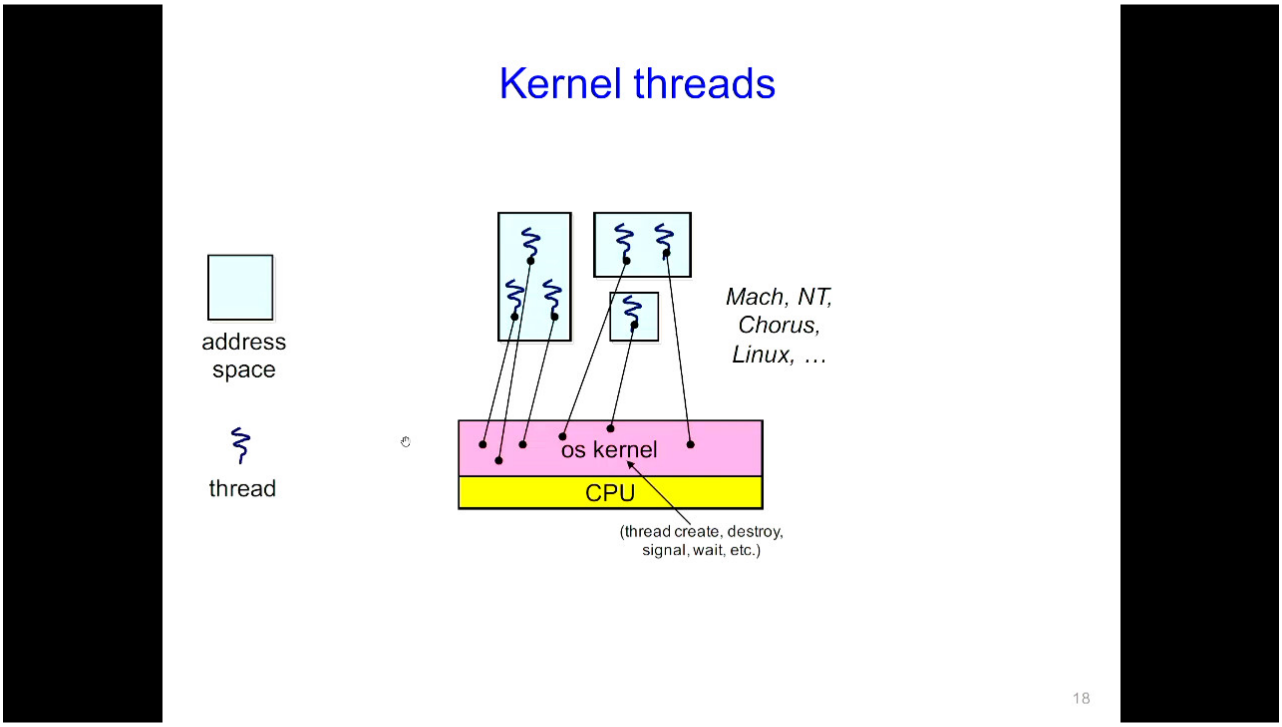
- Concurrency (multi-threading) is useful for:
  - handling concurrent events (e.g. web servers and clients)
  - building parallel programs (e.g. matrix multiply, ray tracing)
  - improving program structure (the Java argument),
- Multi-threading is useful even on a uniprocessor
  - even though only 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

address space
thread
Mach, NT, Chorus, Linux, …
os kernel
CPU
(thread create, destroy, signal, wait, etc.)
18

Kernel Threads

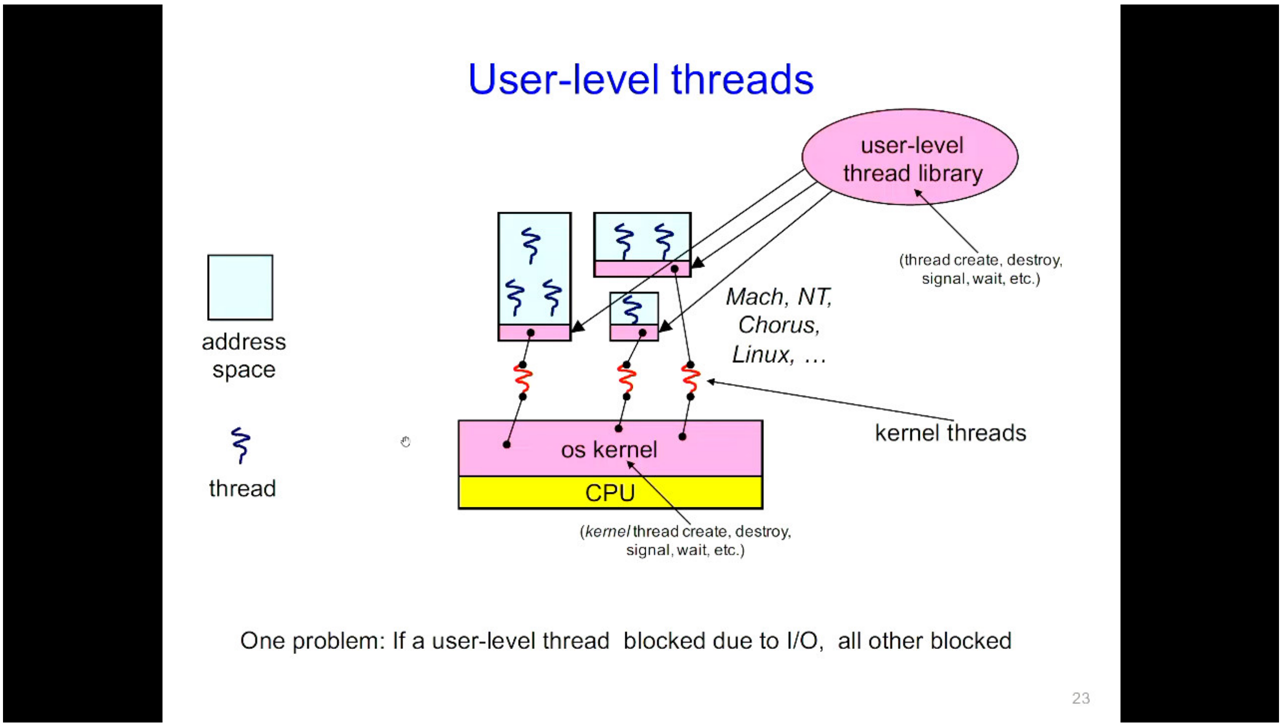
- OS now manages threads *and* processes / address spaces
  - all thread operations are implemented in the kernel
  - OS schedules all of the threads in a system
    - if one thread in a process blocks (e.g. on I/O), the OS knows about it, and can run other threads from that process
    - possible to 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
thread library
(thread create, destroy, signal, wait, etc.)
address space
thread
Mach, NT, Chorus, Linux, …
kernel threads
os kernel
CPU
(*kernel* thread create, destroy, signal, wait, etc.)

One problem: If a user-level thread blocked due to I/O, all other blocked

23

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

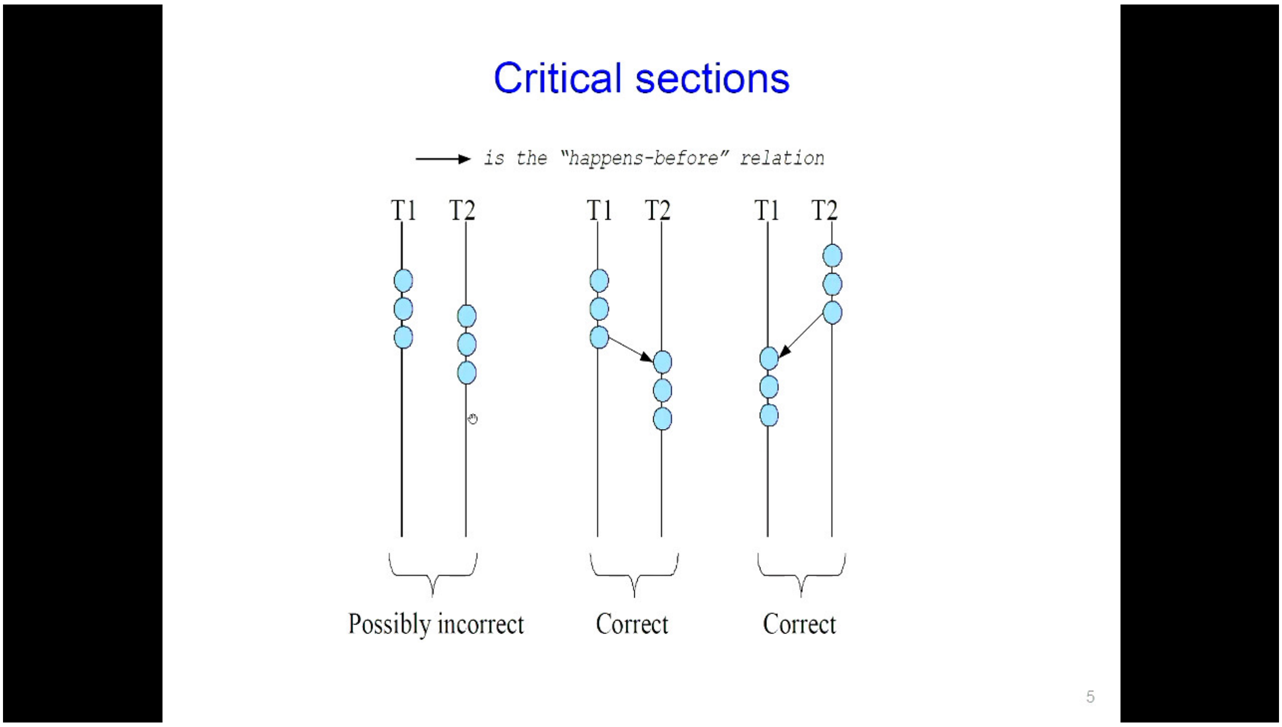
5 Synchronisation

Temporal relations

- User view of parallel threads
  - Instructions executed by a single thread are totally ordered
    - $A < B < C < \dots$
  - 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.



Critical sections

→ is the "happens-before" relation

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

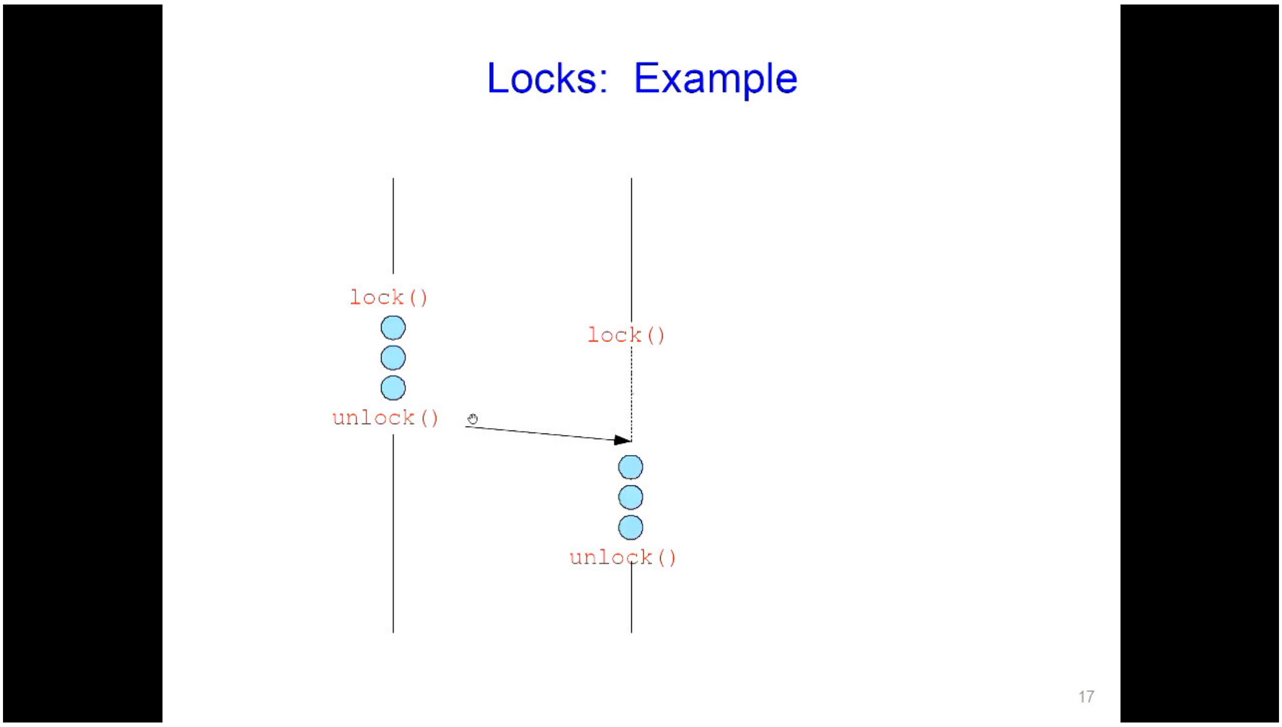
- Spinlocks
  - primitive, minimal semantics — used to build others
- Semaphores (and non-spinning locks)
  - basic, easy to understand, somewhat hard to program with
- Monitors
  - higher level, requires language support, implicit operations
  - easier to program with; Java “`synchronised`”, for example
- 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



Locks: Example

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...))

6 Semaphores, Condition Variables, and Monitors

6.1 Semaphore

Semaphore

- More sophisticated synchronisation mechanism
- Semaphore `S` — integer variable
- Can only be accessed via two atomic operations: `wait` and `signal` (originally called `P` and `V`).
- Definitions

```
wait(S) {
    while (S <= 0); // busy wait
    S--;
}
signal(S) {
    S++;
}
```

- These are performed *atomically*

Semaphore Usage

- *Counting semaphore*: integer value can range over an unrestricted domain
- *Binary semaphore*: integer value can range only between 0 and 1 (same as *lock*)
- Can solve various synchronisation problems
- Consider $\texttt{P}_1$ and $\texttt{P}_2$ that require $\texttt{S}_1$ to happen before $\texttt{S}_2$
  Create a semaphore "`synch`" initialised to 0

```
P1:
    S_1;
    signal(synch);
P2:
    wait(synch);
    S_2;
```

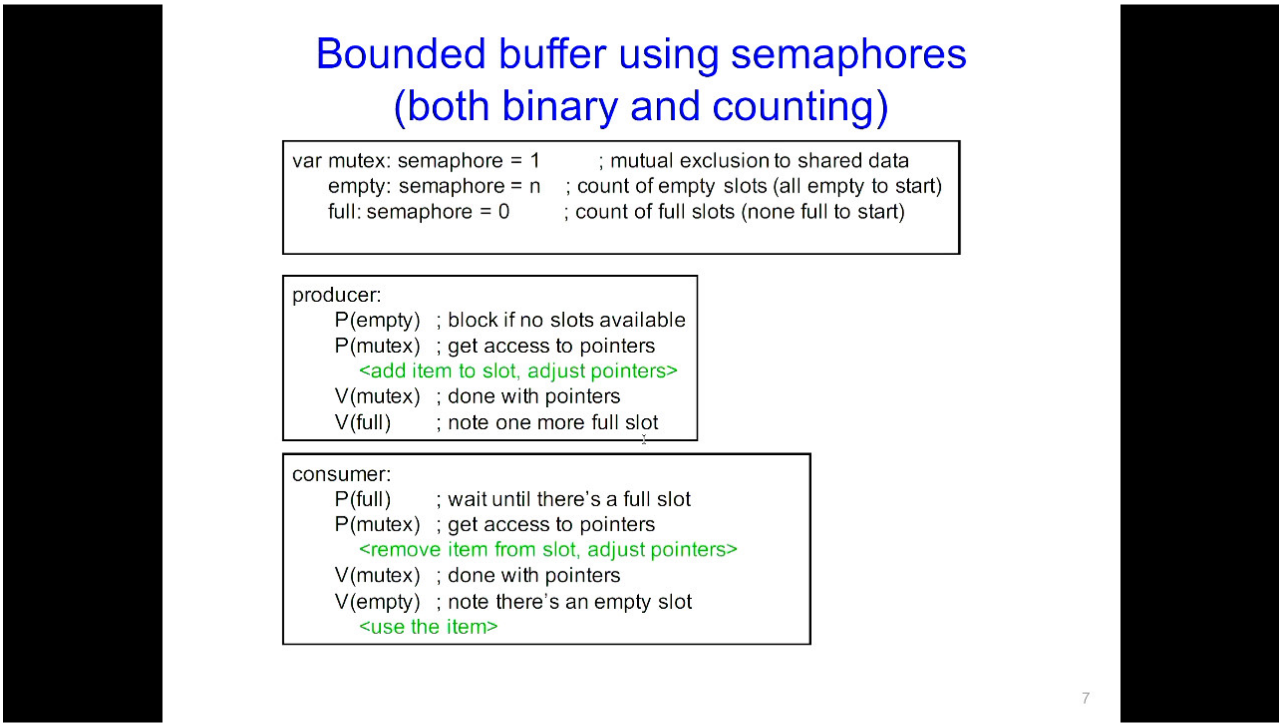
- Can implement a counting semaphore `S` as a binary semaphore.

Implementation with no Busy waiting

Each semaphore has an associated queue of threads

```
wait(semaphore *S) {
    S->value--;
    if (S->value < 0) {
        add this thread to S->list;
        block();
    }
}
signal(semaphore *S) {
    S->value++;
    if (S->value <= 0) {
        remove a thread T from S->list;
        wakeup(T);
        }
    }
```

Examples



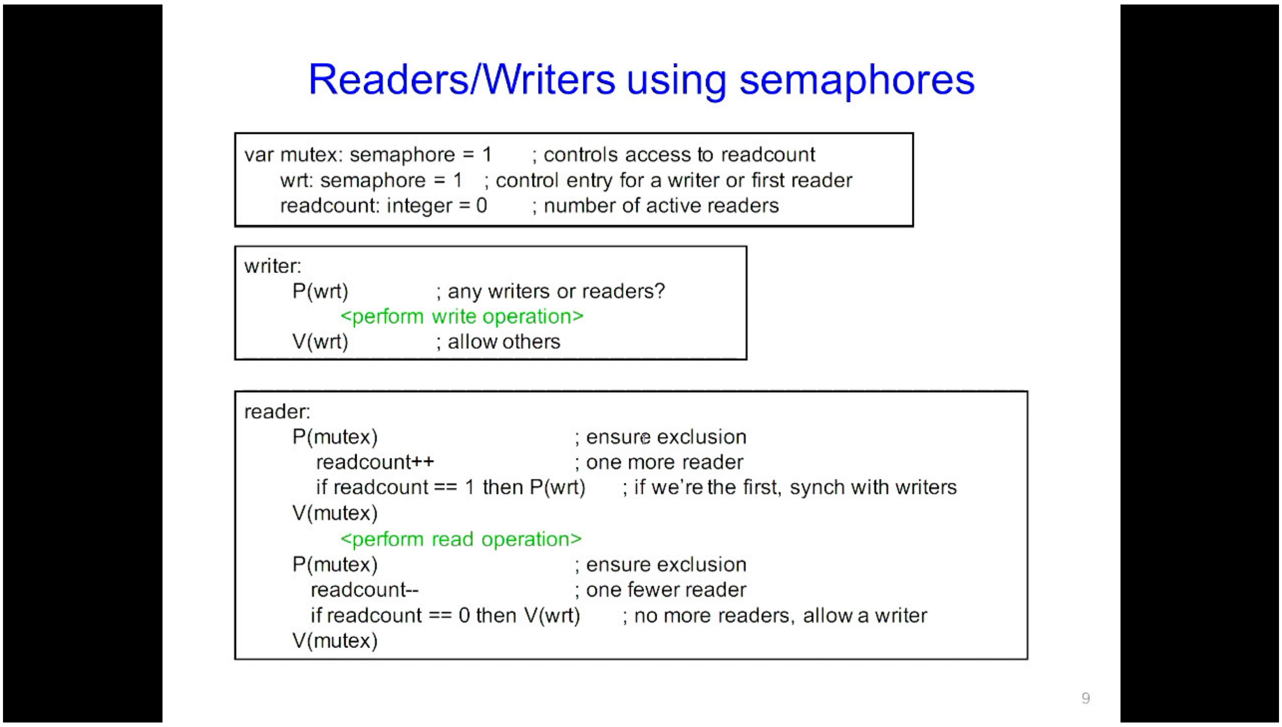
Bounded buffer using semaphores (both binary and counting)

```
var mutex: semaphore = 1         ; mutual exclusion to shared data
    empty: semaphore = n    ; count of empty slots (all empty to start)
    full: semaphore = 0     ; count of full slots (none full to start)
```

```
producer:
    P(empty)  ; block if no slots available
    P(mutex)  ; get access to pointers
      <add item to slot, adjust pointers>
    V(mutex)  ; done with pointers
    V(full)   ; note one more full slot
```

```
consumer:
    P(full)   ; wait until there's a full slot
    P(mutex)  ; get access to pointers
      <remove item from slot, adjust pointers>
    V(mutex)  ; done with pointers
    V(empty)  ; note there's an empty slot
      <use the item>
```

7



Readers/Writers using semaphores

```
var mutex: semaphore = 1      ; controls access to readcount
    wrt: semaphore = 1    ; control entry for a writer or first reader
    readcount: integer = 0      ; number of active readers
```

```
writer:
    P(wrt)            ; any writers or readers?
        <perform write operation>
    V(wrt)            ; allow others
```

```
reader:
    P(mutex)                        ; ensure exclusion
      readcount++                   ; one more reader
      if readcount == 1 then P(wrt)     ; if we're the first, synch with writers
    V(mutex)
        <perform read operation>
    P(mutex)                        ; ensure exclusion
      readcount--                   ; one fewer reader
      if readcount == 0 then V(wrt)     ; no more readers, allow a writer
    V(mutex)
```

9

Semaphores v.s. Spinlocks

- Threads that are blocked at the level of program logic (that is, by the semaphore P operation) are placed on queues, rather than busy-waiting.
- Busy-waiting may be used for the "real" mutual exclusion required to implement P and V
  - but these are very short critical sections — totally independent of program logic
  - and they are not implemented by the application programmer.

Abstract implementation

- `P(sem)`
  - acquire “real” mutual exclusion
    - if `sem` is “available” (¿ 0), decrement sum; *release “real” mutual exclusion*; let thread continue
    - otherwise, place thread on associated queue; *release “real” mutual exclusion*; run some other thread.
- `V(sem)`
  - *acquire “real” mutual exclusion*
    - if threads are waiting on the associated queue, unblock one (place it on the ready queue)
    - if no threads are on the queue, `sem` is incremented
the signal is “remembered” for the next time `P(sem)` is called
  - release “real” mutual exclusion
  - the “`V`-ing” thread continues execution.

Problems with semaphores, locks

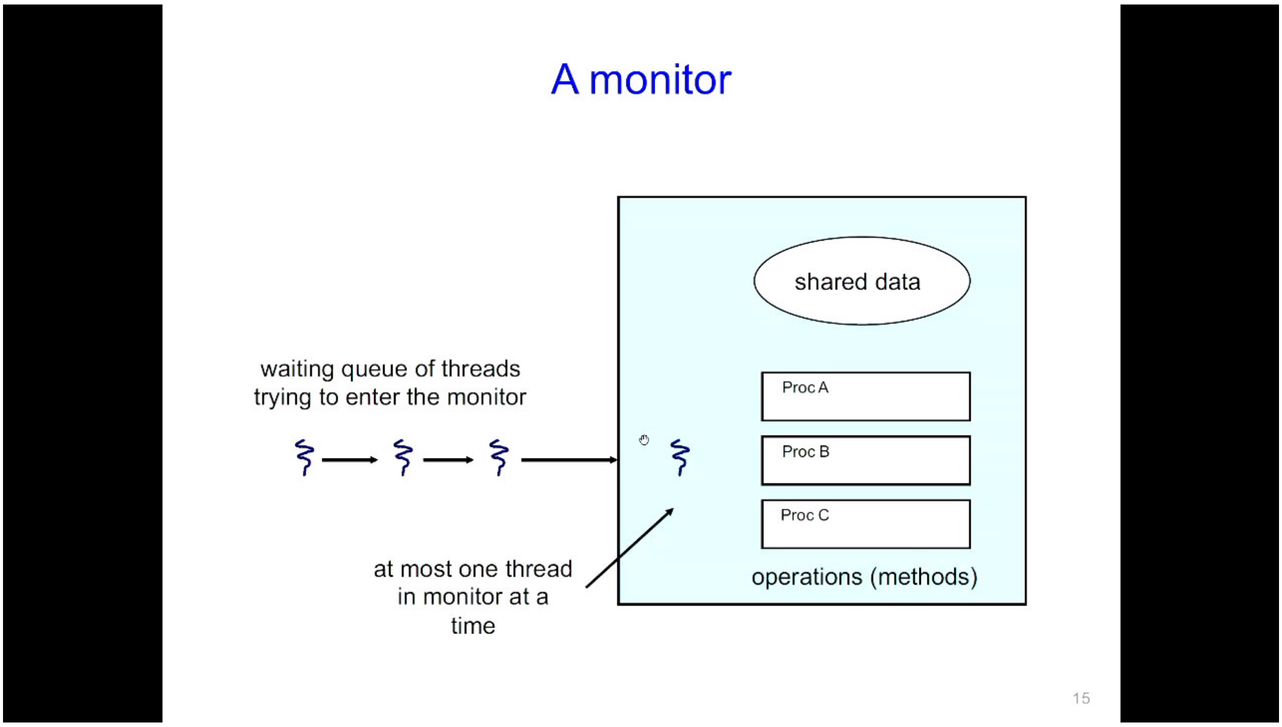
- They can be used to solve any of the traditional synchronisation problems, but it’s easy to make mistakes
  - they are essentially shared global variables
    - can be accessed from anywhere (bad software engineering)
  - there is no connection between the synchronisation variable and the data being controlled by it
  - no control over their use, no guarantee of proper usage
    - Semaphores: will here ever be a `V()`?
    - Locks: did you lock when necessary? Unlock at the right time? At all?
- Thus, they are prone to bugs
  - We can reduce the chance of bugs by “stylising” the use of synchronisation
  - Language help is useful for this.

6.2 Monitors

Monitors

- A programming language construct supports controlled shared data access
  - synchronisation code is added by the compiler.
- A class in which every method automatically acquires a lock on entry, and releases it on exit — it combines:
  - *shared data* structures (object);
  - *procedures* that operate on the shared data (object methods);
  - *synchronisation* between concurrent threads that invoke those procedures.
- Data can only be accessed from within the monitor

  - protects the data from unstructured access;
  - prevents ambiguity about what the synchronisation variable protects.
- Addresses the key usability issues that arise with semaphores.



A monitor

shared data

Proc A

Proc B

Proc C

operations (methods)

waiting queue of threads trying to enter the monitor

at most one thread in monitor at a time

15

Monitor facilities

- "Automatic" mutual exclusion
  - only one thread can be executing inside at any time
    - thus, synchronisation is implicitly associated with the monitor — it "comes for free";
  - if a second thread tries to execute a monitor procedure, it blocks until the first has left the monitor;
    - more restrictive than semaphores,
    - but easier to use (most of the time).
- But, there's a problem...
  Bounded buffer scenario.

Bounded Buffer scenario

- Monitors require condition variables
- Operations on condition varibales
  - `wait (c)`
    - release monitor lock, so somebody else can get in
    - wait for somebody else to signal condition
    - thus, condition variables have associated wait queues
  - `signal (c)`
    - wake up at most one waiting thread
      - "Hoare" monitor: wakeup immediately, signaller steps outside

    - if no waiting threads, signal is lost
        - this is different from semaphores — no history!
- broadcast (c)
    - wake up all waiting threads.

Bounded buffer using (Hoare) monitors

```
Monitor bounded_buffer {
    buffer resources[];
    condition not_full;
    condition not_empty;

    produce(resource x) {
        if (array "resources" is full, determined maybe by a count) {
            wait(not_full);
        }
        insert "x" in array "resources";
        signal(not_empty);
    }

    consume(resource *x) {
        if (array "resources" is empty, determined maybe by a count) {
            wait(not_empty);
        }
        *x = get resource from array "resources";
        signal(not_full);
    }
}
```

Runtime system calls for (Hoare) monitors

- `EnterMonitor (m)` {guarantee mutual exclusion}
- `ExitMonitor (m)` {hit the road, letting someone else run}
- `Wait (c)` {step out until condition satisfied}
- `Signal (c)` {if someone's waiting, step out and let them run}
- `EnterMonitor` and `ExitMonitor` are inserted automatically by the compiler.
- This guarantees mutual exclusion for code inside of the monitor.

Monitor Summary

- Language supports monitors
- Compiler understands them
    - Compiler inserts calls to runtime routines for
        - monitor entry
        - monitor exit
    - Programmer inserts calls to runtime routines for
        - signal
        - wait

  - Language/object encapsulation ensures correctness
    - Sometimes! With conditions, you *still* need to think about synchronisation
- Runtime system implements these routines
  - moves threads on and off queues
  - *ensures mutual exclusion!*

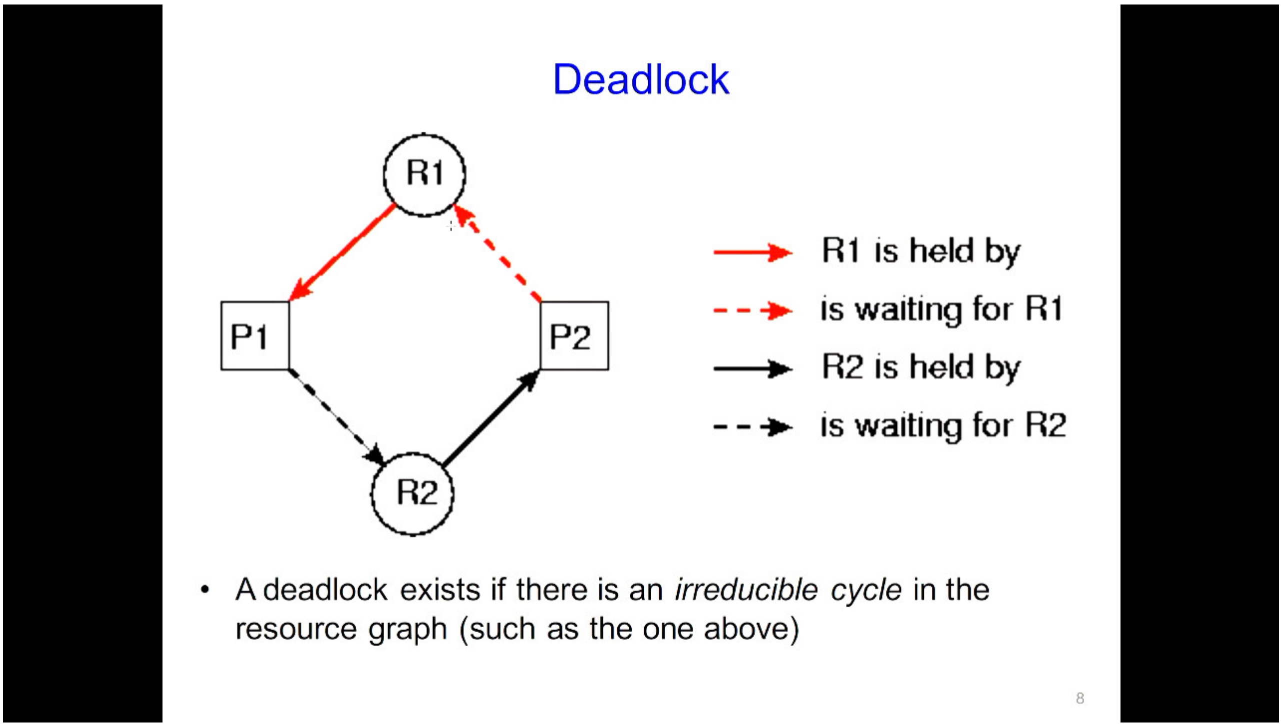
7 Deadlock

Definition

- A thread is deadlocked when it's waiting for an event that can never occur
- Thread `A` is in critical section 1
  waiting for access to critical section 2;
- Thread `B` is in critical section 2
  waiting for access to critical section 1

Four conditions must exist for deadlock to be possible

1. Mutual exclusion
2. Hold and wait
3. No pre-emption
4. Circular wait



Deadlock

→ R1 is held by

⇢ is waiting for R1

→ R2 is held by

⇢ is waiting for R2

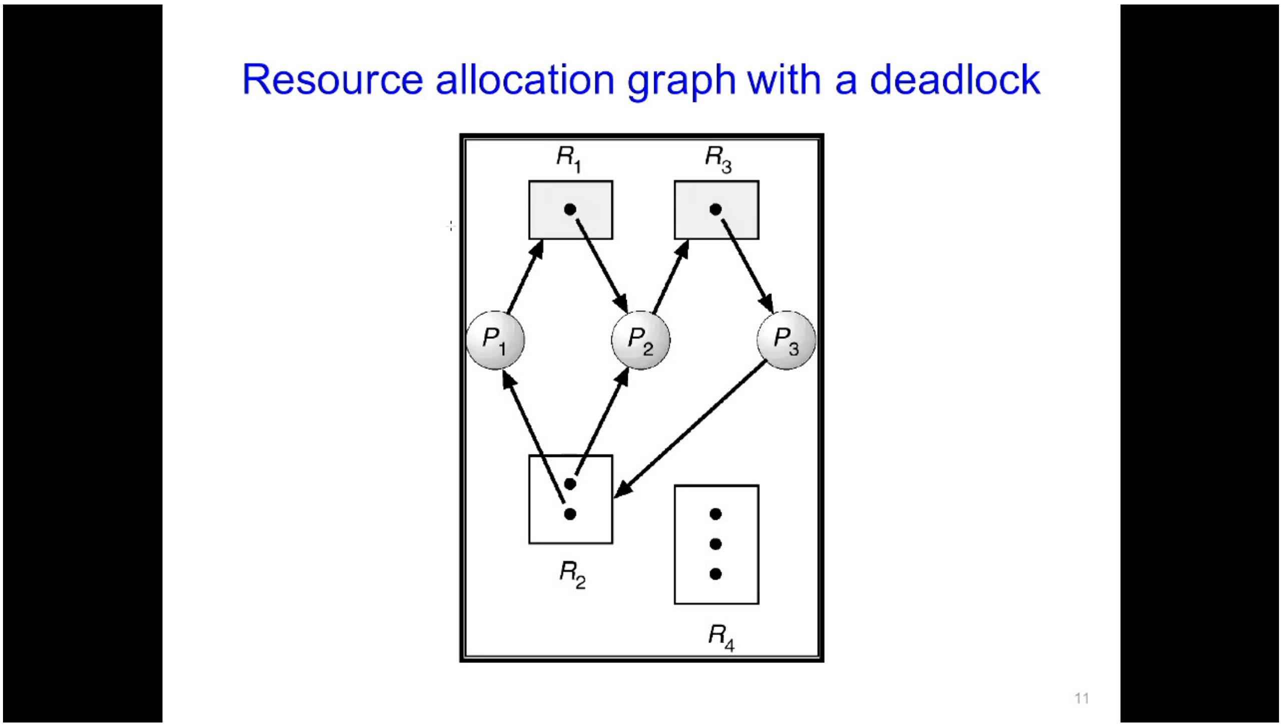
- A deadlock exists if there is an *irreducible cycle* in the resource graph (such as the one above)

8

7.1 Graph reduction

Graph reduction

- A graph can be *reduced* by a thread if all of that thread's requests can be granted
  - in this case, the thread eventually will terminate — all resources are freed — all arcs (allocations) to/from it in the graph are deleted.
- Miscellaneous theorems (Holt, Havender):
  - There are no deadlocked threads if and only if the graph is completely reducible.
  - The order of reductions is irrelevant.



Resource allocation graph with a deadlock

Handling deadlock

- Eliminate one of the four required conditions
  - Mutual exclusion
  - Hold and Wait
  - No pre-emption
  - Circular wait
- Broadly classified as:
  - Prevention, or
  - Avoidance, or
  - Detection (and recovery)

Deadlock prevention

Restrain the ways requests can be made

- Mutual exclusion
  not required for sharable resources (e.g. read-only files); must hold for non-sharable resources.
- Hold and wait
  must guarantee that whenever a process requests a resources, it does not hold any other resources.
  - Low resources utilisation; starvation is possible.
- No (resource) Pre-emption
  - If a process holding some resources requests another unavailable resource all resources currently held are released.
  - Process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting.
- Circular wait

  - impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration.

Avoidance
Less severe restrictions on program behaviour.

- Eliminating circular wait
  - each thread states its maximum claim for every resource type;
  - system runs the Banker's Algorithm at each allocation request
    Banker $\Longrightarrow$ highly conservative

7.2 Banker's Algorithm

Banker's Algorithm example

- Background
  - The set of controlled resources is known to the system.
  - The number of units of each resource is known to the system.
  - Each application must declare its maximum possible requirement of each resource type.
- The, the system can do the following:
  - When a request is made:
    - pretend you granted it;
    - pretend all other legal requests were made;
    - can the graph be reduced?
      - If so: allocate the requested resource.
      - If not, block the thread until some thread releases resources, and then try pretending again.

Safe state

- When requesting an available resource decide if allocation leaves the system in a safe state
- We're in a *safe state* if there exists a sequence $< P_1, P_2, \ldots, P_n >$ of *all* the processes in the systems
  - such that for each P_i, the resources that P_i can still request can be satisfied by currently available resources + resources held by all the P_j, with $j < i$.
- That is:
  - If P_i resource needs are not immediately available, then P_i can wait until all P_j have finished.
  - When P_j is finished, P_i can obtain needed resources, execute, return allocated resources, and terminate.
  - When P_i terminates, P_{i+1} can obtain its needed resources, and so on.

Safe $\Longrightarrow$ no deadlock; deadlock $\Longrightarrow$ unsafe

Data Structures for the Banker's Algorithm Let `n` = number of processes, and `m` = number of resource types.

- `Available`: Vector of length `m`. If `Available[j]` = `k`, there are `k` instances of resource type `Rj` available.

- `Max` $\texttt{n} \times \texttt{m}$ matrix. If `Allocation[i,j]` = `k`, then P_i is currently allocated `k` instances of $\texttt{R}_\texttt{j}$
- `Allocation`: $\texttt{n} \times \texttt{m}$ matrix. If `Need[i,j]` = `k`, then $\texttt{P}_\texttt{i}$ may need `k` more instances of $\texttt{R}_\texttt{j}$ to complete its task.

$$\texttt{Need[i,j]} = \texttt{Max[i,j]} - \texttt{Allocation[i,j]}$$

Safety Algorithm

1. Let `Work` and `Finish` be vectors of length `m` and `n`, respectively. Initialise:
   `Work = Available`
   `Finish[i] = false for i = 0..n-1`
2. Find an `i` such that both:
   (a) `Finish[i] == false`
   (b) $\texttt{Need}_\texttt{i}$ `<= Work`

   If no such `i` exists, go to step 4
3. `Work = Work + Allocation`
   `Finish[i] = true`
   go to step 2
4. If `Finish[i] == true, for all i,` then the system is in a safe state.

Resource-Request Algorithm for Process $\texttt{P}_\texttt{i}$

$\texttt{Request}_\texttt{i}$ = request vector for process $\texttt{P}_\texttt{i}$. If $\texttt{Request}_\texttt{i}$`[j] == k` then process $\texttt{P}_\texttt{i}$ wants `k` instances of resource type $\texttt{R}_\texttt{j}$.

1. If $\texttt{Request}_\texttt{i}$ `<=` $\texttt{Need}_\texttt{i}$ go to step 2. Otherwise raise error condition, since process has exceeded its maximum claim.
2. If $\texttt{Request}_\texttt{i}$ `<= Available`, go to step 3. Otherwise $\texttt{P}_\texttt{i}$ must wait, since resources are not available.
3. Pretend to allocate requested resources to $\texttt{P}_\texttt{i}$ by modifying the state as follows:
   `Available = Available - Request`
   $\texttt{Allocation}_\texttt{i}$ `=` $\texttt{Allocation}_\texttt{i}$ `+` $\texttt{Request}_\texttt{i}$
   $\texttt{Need}_\texttt{i}$ `=` $\texttt{Need}_\texttt{i}$ `-` $\texttt{Request}_\texttt{i}$

   (a) If safe, then resources allocated to $\texttt{P}_\texttt{i}$

   (b) If unsafe, then $\texttt{P}_\texttt{i}$ must wait, and the old resource-allocation state is restored.

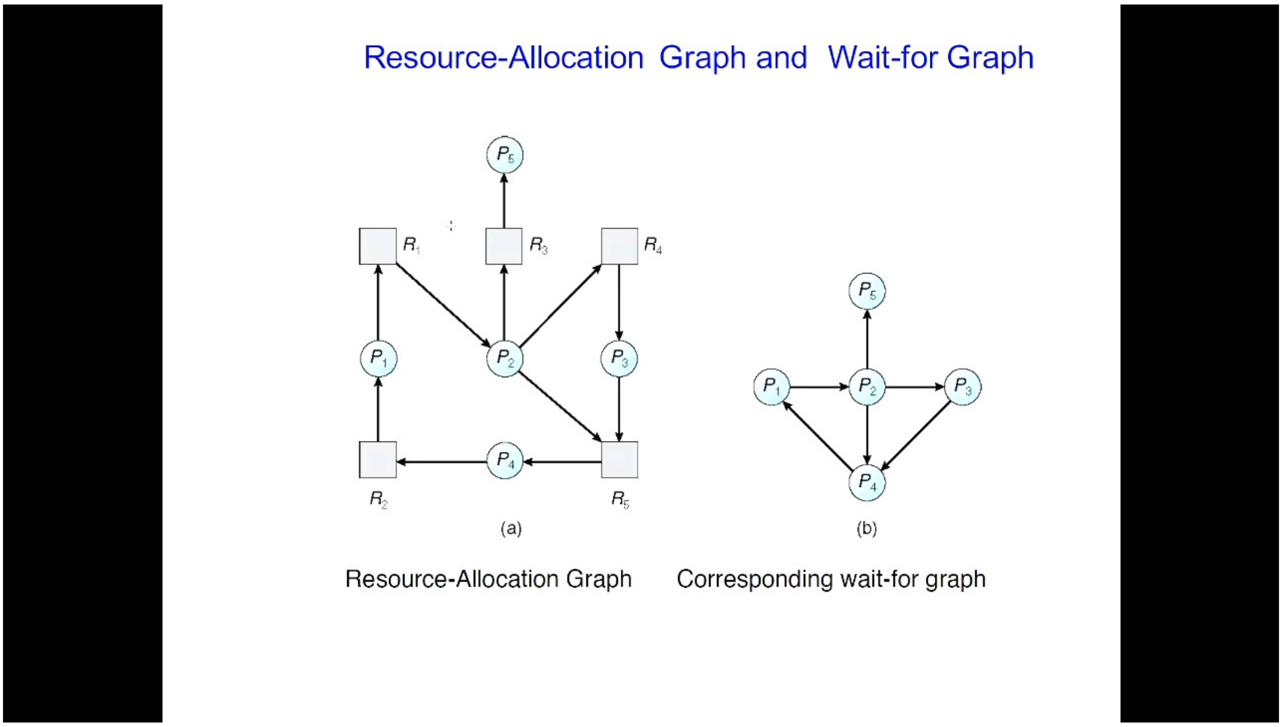
Deadlock Detection

1. Allow system to enter deadlock state
2. Detection algorithm
3. Recovery scheme

Single instance of each resource type

- Maintain a *wait-for* graph
  - Nodes are processes
  - $\texttt{P}_\texttt{i} \rightarrow \texttt{P}_\texttt{j}$ if $\texttt{P}_\texttt{i}$ is waiting for $\texttt{P}_\texttt{j}$
- Periodically invoke an algorithm that searches for a cycle in the graph.
  - If there is a cycle, there exists a deadlock.
- An algorithm to detect a cycle in a graph

  - has runtime complexity $\mathcal{O}(n^2)$ with n being the number of vertices in the graph.



Resource-Allocation Graph and Wait-for Graph

Resource-Allocation Graph

Corresponding wait-for graph

Detection-Algorithm usage

- When, and how often to invoke depends on:
  - How often a deadlock is likely to occur?
  - How many processes will need to be rolled back?
    * One for each disjoint cycle.
- If detection algorithm is invoked arbitrarily:
  - there may be many cycles in the resource graph
  - we would not be able to tell which deadlocked processes “caused” the deadlock.

Recovery from deadlock

- Process termination
  - Abort all deadlocked processes
  - Abort one process at a time until the deadlock cycle is eliminated
  - In which order should we choose to abort?
- Resource pre-emption
  - *Select a victim* — minimise cost
  - *Rollback* — return to some safe state, restart process for that state.
  - *Starvation* — same process may always be picked as victim, include number of rollback in cost factor.

Summary

- Deadlock is bad!
- We can deal with it either statically (prevention) or dynamically (avoidance and/or detection)

- In practice, you'll encounter lock ordering, periodic deadlock detection/correction, and minefields.

8 Scheduling

Scheduling

- We have talked about *context switching*
  - an interrupt occurs (device completion, timer interrupt)
  - a thread causes a trap or execution
  - may need to choose a different thread/process to run
- Glossed over which process or thread to run next
  - "some thread from the ready queue"
- This decision is called *scheduling*
  - scheduling is a *policy*
  - context switching is a *mechanism*

Classes of Schedulers

- Batch
  - Throughput / utilisation oriented
  - Example: audit inter-bank funds transfers each night, Pixar rendering, Hadoop/MapReduce jobs.
- Interactive
  - Response time oriented
- Real time
  - Deadline driven
  - Example: embedded systems (cars, aeroplanes, etc.)
- Parallel
  - Speedup-driven
  - Example: "space-shared" use of a 1000-processor machine for large simulations.

Multiple levels of scheduling decisions

- Long term
  - Should a "job" be "initiated", or should it be held?
  - Typical of batch systems.
- Medium term
  - Should a running program be temporarily marked as non-runnable (e.g. swapped out)?
- Short term
  - Which thread should be given to the CPU next? For how long?
  - Which I/O operation should be sent to the disk next?
  - On a multiprocessor:
    - Should we attempt to coordinate the running of threads from the same address space in some way?

      * Should we worry about cache state (processor affinity)?

8.1 Scheduling Goals

Scheduling Goals I: Performance
Many possible metrics / performance goals (which sometimes conflict)

- maximise *CPU utilisation*
- maximise *throughput* (requests completed per second)
- minimise *average response time* (average time from submission of request to completion of response)
- minimise *average waiting time* (average time from submission of request to start of execution)
- minimise *energy* (joules per instruction) subject to some constraint (e.g. frames per second)

Scheduling Goals II: Fairness

- No single, compelling definition of “fair”
  - How to measure fairness?
  - Fair per-user? Per-process? Per-thread?
  - What if one process is CPU bound, and one is I/O bound?
- Sometimes the goal is to be unfair:
  - Explicitly favour some particular class of requests (priority system), but...
  - avoid starvation (be sure everyone gets at least some service).

When to assign?
Pre-emptive v.s. non-pre-emptive schedulers

- Non pre-emptive
  once you give somebody the green light, they’ve got it until they relinquish it
  - an I/O operation
  - allocation of memory in a system without swapping
- Pre-emptive
  - you can re-visit a decision
    * setting the timer allows you to pre-empt the CPU from a thread even if it doesn’t relinquish it voluntarily.
  - Re-assignment always involves some overhead
    * Overhead doesn’t contribute to the goal of any scheduler.

We’ll assume “work conserving” policies

- Never leave a resource idle when someone wants it

8.2 Laws and properties

The Utilisation Law: $U = X \times S$

- U utilisation
- X throughput (requests per second)
- S average service time

- Utilisation is constant, independent of the schedule, so long as the workload can be processed

Little's Law: $N = X \times R$

- N average number in system
- X throughput
- R average response time
- a better average response time implies fewer in system, and vice versa.

Response Time R at a single server under FCFS scheduling:

- $R = \frac{S}{1-U}$
- $N = \frac{U}{1-U}$

8.3 Algorithms

Algorithm 1: First-come first-served (FCFS)

- schedule in the order that they arrive
- "real-world" scheduling of people in (single) lines
- jobs treated equally, no starvation
- Drawbacks:
  - Average response time can be poor: *convoy effect*
  - May lead to poor utilisation of other resources
    - if you send me on my way, I can go keep another resource busy
    - FCFS may result in poor overlap of CPU and I/O activity
  - The more copies of the resource there are to be scheduled
    - the less dramatic the impact of occasional very large jobs (so long as there is a single waiting line)
    - e.g. multiple cores v.s. single core

Algorithm 2: Shortest-job-first (SJF)

- Associate with each process the length of its next CPU burst
  - use these lengths to schedule the process with the shortest time
- SJF is optimal — gives minimum average waiting time for a given set of processes
  - the difficulty is knowing the length of the next CPU request
  - could ask the user.
- Determining the length of next CPU burst
  - Can only estimate the length — should be similar to the previous one
    - then pick process with shortest predicted next CPU burst.
  - Can be done by using the length of previous CPU bursts, using exponential averaging
    1. t_n actual length of nth CPU burst
    2. τ_{n+1} predicted value for the next CPU burst

3. $\alpha, 0 \leq \alpha \leq 1$
4. Define: $\tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n$

- Commonly, set $\alpha = 0.5$
- Pre-emptive version called *shortest-remaining-time-first*

Algorithm 3: Round Robin (RR)

- Each process gets a small unit of CPU time (*time quantum* q), usually 10–100 milliseconds.
  - After this time has elapsed, the process is pre-empted and added to the end of the ready queue.
- If there are n processes in the ready queue and the time quantum is q,
  - then each process gets $\frac{1}{n}$ of the CPU time in chunks of at most q time units at once.
  - No process waits more than $(n - 1)q$ time units.
- Timer interrupts every quantum to schedule next process
- Performance
  - q large $\implies$ FIFO
  - q small $\implies$ q must be large with respect to context switch, otherwise overhead is too high.
- Drawbacks:
  - What if all jobs are exactly the same length?
  - What do you set the quantum to be?
    * no value is "correct"
    * if small, then context switch often, incurring high overhead
    * if large, then the response time degrades.
  - Treats all jobs equally

Algorithm 4: Priority Scheduling

- A priority number (integer) is associated with each process
- The CPU is allocated to the process with the highest priority
- SJF is priority scheduling where priority is the inverse of predicted next CPU burst time.
- Problem: *starvation* — low priority processes may never execute.
- Solution: *ageing* — as time progresses, increase the priority of the process.

Multi-level Feedback Queues (MLFQ)

- It's been observed that workloads tend to have increasing residual life — "if you don't finish quickly, you're probably a lifer"
- This is exploited in practice by using a policy that discriminates against the old.
- MLFQ:
  - there is a hierarchy of queues
  - there is a priority ordering among the queues
  - new requests enter the highest priority queue

  - each queue is scheduling RR
  - requests move between queues based on execution history.

UNIX scheduling

- Canonical scheduler is pretty much MLFQ
  - 3–4 classes spanning ~ 170 priority levels
    - * time-sharing: lowest 60 priorities
    - * system: middle 40 priorities
    - * real-time: highest 60 priorities
  - priority scheduling across queues, RR within
    - * process with highest priority always run first
    - * processes with same priority scheduled RR
  - processes dynamically change priority
    - * increases over time if process blocks before end of quantum
    - * decreases if process uses entire quantum
- Goals:
  - reward interactive behaviour over CPU hogs
    - * interactive jobs typically have short bursts of CPU

8.4 Summary

- Scheduling takes place at many levels
- It can make a huge difference in performance
  - this difference increases with the variability in service requirements
- Multiple goals, sometimes conflicting
- There are many "pure" algorithms, most with some drawbacks in practice — FCFS, SJF, RR, Priority
- Real system use hybrids that exploit observed program behaviour
- Scheduling is important

9 Memory Management

9.1 Background

Goals and Tools of memory management

- Allocate memory resources among competing processes
  - maximising memory utilisation and system throughput
- Provide isolation between processes
  - Addressability and protection: orthogonal
- Convenient abstraction for programming
  - and compilers, etc.
- Tools
  - Base and limit registers
  - Swapping
  - Segmentation
  - Paging, page tables, and TLB
  - Virtual memory

Background

- Program must be brought (from disk) into memory and placed within a process for it to be run.
- Main memory and registers are only storage CPU can access directly.
- Memory unit only sees a stream of address + read requests, or addresses + data and write requests
- Register access in one CPU clock (or less)
- Main memory can take many cycles, causing a *stall*
- *Cache* sits between main memory and CPU registers.
- Protection of memory required to ensure correct operation.

Base and Limit Registers

- A pair of *base* and *limit* registers define the logical address space.
- CPU must check every memory access generated in user mode to be sure it is between base and limit for that user.

9.2 Logical/Virtual address space v.s. Physical address space

Virtual address for multiprogramming

- To make it easier to manage memory of multiple processes, make processes *use logical or virtual address*
  - Logical/virtual addresses are independent of location in physical memory data lives
    - OS determines location in physical memory.
- Instructions issued by CPU reference logical/virtual addresses
  - e.g. pointers, arguments to load/store instructions, PC, etc.

- Logical/virtual addresses are translated by hardware into physical addresses (with some setup from OS).

Logical/Virtual Address Space

- The set of logical/virtual addresses a process can reference is its *address space*
  - many different possible mechanisms for translating logical/virtual addresses to physical addresses.
- Program issues addresses in a logical/virtual address space
  - must be *translated* to physical address space;
  - think of the program as having a contiguous logical/virtual address space that starts at 0; and a contiguous physical address space that starts somewhere else.
- *Logical/virtual address space* is the set of all logical addresses generated by a program.
- *Physical address space* is the set of all physical addresses generated by a program.

Memory-Management Unit (MMU)

- Hardware device
  - at runtime maps virtual to physical addresses
- Many methods possible
- Simple scheme: value in relocation register is added to every address generated by a user process at the time it is sent to memory.
  - Base register now called *relocation register*
- The user program deals with *logical* addresses — it never sees the *real* physical addresses.
  - Execution-time binding occurs when reference is made to location in memory.
  - Logical address bound to physical addresses.

9.3 Swapping

Swapping

- What if not enough memory to hold all processes?
- A process can be *swapped* temporarily
  - out of memory to a backing store,
  - brought back into memory for continued execution
  - total physical memory space of processes can exceed physical memory.
- *Backing store* — fast disk
  - large enough to accommodate copies of all memory images for all users;
  - must provide direct access to these memory images.
- *Roll out, roll in* — swapping variant
  - used for priority-based scheduling algorithms;
  - lower-priority process is swapped out so higher-priority processes can be loaded and executed.
- Major part of swap time is transfer time

  - total transfer time is directly proportional to the amount of memory swapped.

- System maintains a *ready queue*
  - ready-to-run processes which have memory images on disk.

Context Switch Time including Swapping

- If next processes to be put on CPU is not in memory
  - need to swap out a process and swap in target process.
- Context switch time can then be very high
- Can reduce cost
  - reduce size of — by knowing how much memory really being used;
  - inform OS of memory use via `request_memory()` and `release_memory()`.
- Other constraints as well on swapping
  - Pending I/O — can't swap out as I/O would occur to wrong process.
- Or always transfer I/O to kernel space, then I/O device
  - known as *double buffering*, adds overhead.
- Standard swapping not used in modern operating systems
  - But modified version common
    - Swap only when free memory extremely low.

9.4 Contiguous Memory Allocation

Contiguous Allocation

- Main memory must support both OS and user processes
- Limited resource, must allocate efficiently
- Contiguous allocation is one early method
- Main memory usually into two *partitions*:
  - Resident OS, usually held in low memory with interrupt vector;
  - User processes then held in high memory;
  - Each process contained in single contiguous section of memory.
- Relocation registers
  - used to protect user processes from each other, and from changing OS code and data;
  - base register contains value of smallest physical address;
  - limit register contains range of logical addresses — each logical address must be less than the limit register.
- MMU maps logical address *dynamically*
  - can then allow actions such as kernel code being *transient* and kernel changing size.

Multiple-partition allocation

- Degree of multiprogramming limited by number of partitions
- 2 approaches

  - Fixed partition
  - Variable partition

Old technique 1: Fixed partitions

- Physical memory is broken up into fixed partitions
  - partitions may have different sizes, but partitioning never changes
  - hardware requirement: *base/relocation register*, *limit register*
    - physical address = logical address + base register
    - base register loaded by OS when it switches to a process
- Advantages:
  - Simple
- Problems
  - *internal fragmentation*: the available partition is larger than what was requested.

Old technique 2: Variable partitions

- Obvious next step: physical memory is broken up into partitions dynamically — partitions are tailored to programs
  - hardware requirements: *base register*, *limit register*
  - physical address = logical address + base register
- Advantages
  - no internal fragmentation
    - simply allocate partition size to be just big enough for process (assuming we know what that is!)
- Problems
  - *external fragmentation*
    - as we load and unload jobs, holes are left scattered throughout physical memory.

Multiple-partition allocation

- *Variable-partition* sizes for efficiency (sized to a given process' needs).
- *Hole* — block of available memory; holes of various sizes are scattered throughout memory.
- When a process arrives, allocated memory from a hole large enough to accommodate it.
- Process exiting frees its partition, adjacent free partitions combined.
- OS maintains information about:
  1. allocated partitions,
  2. free partitions (hole)

Dynamic Storage-Allocation Problem

- *First-fit*: Allocate the *first* hole that is big enough
- *Best-fit*: Allocate the *smallest* hole that is big enough; must search entire list, unless ordered by size
  - produces the smallest leftover hole.

- *Worst-fit*: Allocates the *largest* hole; must also search the entire list
  - produces the largest leftover hole

First-fit and best-fit better than worst-fit in terms of speed and storage utilisation.

Fragmentation

- *External fragmentation*: total memory space exists to satisfy a request, but it is not contiguous;
- *Internal fragmentation*: allocated memory may be slightly larger than requested memory;
- First fit analysis reveals that given N blocks allocated, $0.5N$ blocks lost to fragmentation
- $\frac{1}{3}$ may be unusable $\implies$ *50 percent rule.*

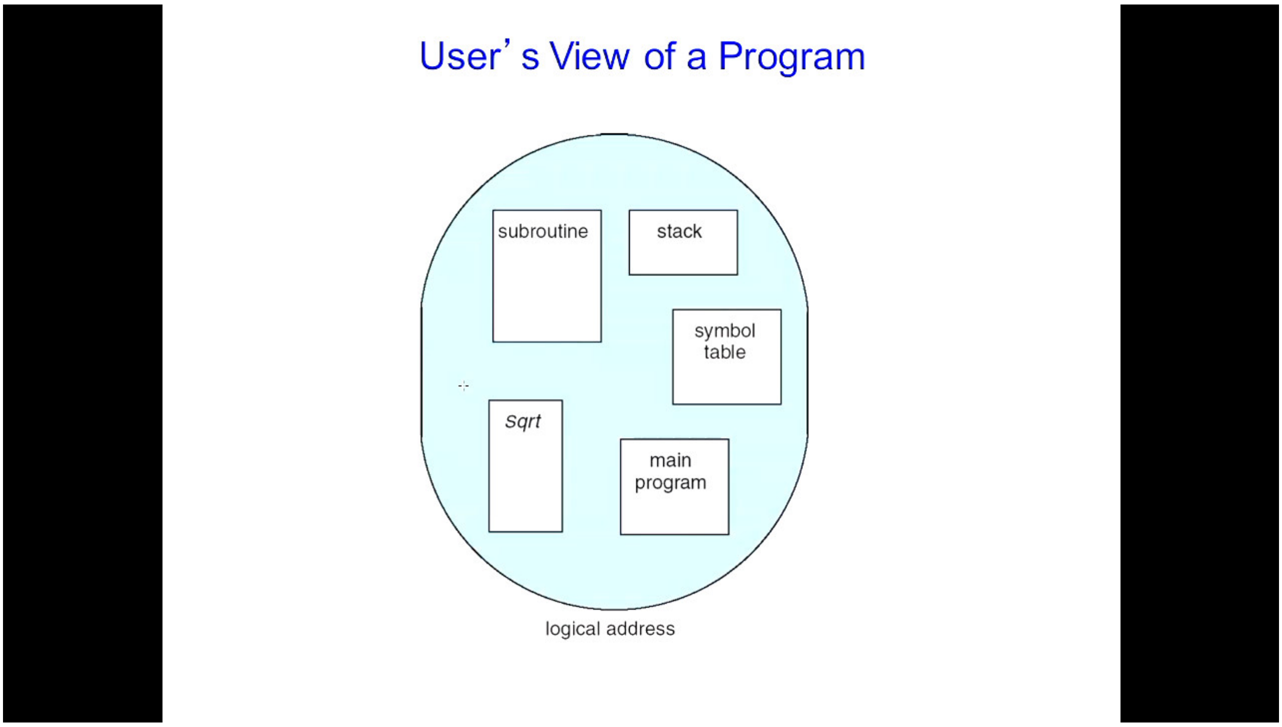
Dealing with fragmentation

- Compact memory by copying
  - Swap a program out
  - Reload it, adjacent to another
  - Adjust its base register
  - Compaction is possible *only* if relocation is dynamic
  - I/O problem:
    - ∗ Latch job in memory while it is involved in I/O
    - ∗ Do I/O only into OS buffers

9.5 Segmentation

Segmentation

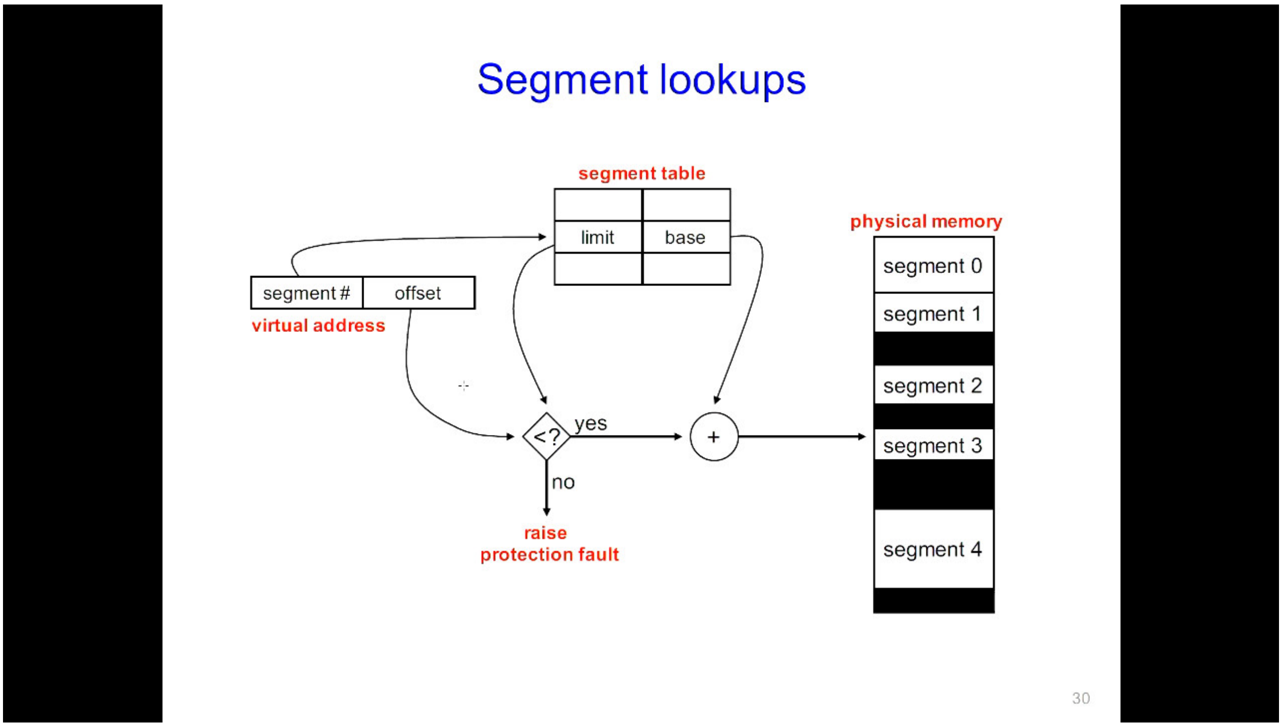
- Dealing with fragmentation
  - Why not remove the need for continuous addresses?
- Segmentation
  - partition an address space into *logical* units
    - ∗ stack, code, heap, subroutines
  - a virtual address is `<segment #, offset>`
- Facilitates sharing and reuse
  - a segment is a natural unit of sharing — a subroutine or function
- A natural extension of variable-sized partitions
  - variable-sized partition = 1 segment per process
  - segmentation = many segments per process.



User’s View of a Program

Hardware support

- Segment table
  - multiple base/limit pairs, *one per segment*
  - segments named by segment #, used as index into table
    - a virtual address is `<segment #, offset>`
  - offset of virtual address added to base address of segment to yield physical address



Segment lookups

Pros and Cons

- Logical and it facilitates sharing and reuse

- Allows non-contiguous physical addresses
  - Helps exploit varying sized holes
- But it has the complexity of a variable partition system
  - except that linking is simpler, and the "chunks" that must be allocated are smaller than a "typical" linear address space.
- Segmentation rarely used alone
  - Paging is the basis for modern memory management

9.6 Summary

- Logical/Virtual Address Space v.s. Physical Address Space
- Swapping
- Contiguous memory allocation
- Fragmentation
- Segmentation
- Paging
  - A better solution