Operating Systems

Lecture Course in Autumn Term 2016
University of Birmingham

Erik Tews, David Oswald
based on a course by Eike Ritter

October 24, 2016

Overview

Course Details

- Lecture notes and resources:
- use canvas for discussions
- Recommended Course Books
  - OS Concepts (8th Edition) Silberschatz *et al.*
  - Linux Kernel Development. Robert Love.
  - Linux Programming Interface, Michael Kerrisk
  - The C Programming Language (2nd Edition) Kernighan and Ritchie

Course Details

- Assessment:
  - 80% exam, 20% coursework
  - Description of coursework and assessment criteria will be made available on canvas.
  - There will be 4 pieces of challenging coursework over the term that will put you through your paces, ultimately to give you a firm grasp on the subject.
  - Students on the extended module (check your registration status with the school office if you are unsure of this) will be given an additional piece of coursework.
  - Make use of the labs (two hours per week) to work on the coursework problems.

Prerequisites for this module

- Good C programming skills are essential prerequisite for this module
- Familiarity with pointers and explicit memory management expected
- See `http://www.cs.bham.ac.uk/~exr/teaching/lectures/opsys/13_14/exercises/ex1.pdf` for the level of C programming skills expected.
- Excercise 1 will be used to freshen up your C skills

What is an Operating System?

- A program that acts as an intermediary between a user of a computer and the computer hardware
- Operating system goals:
  - Execute user programs and make solving user problems easier
  - Make the computer system convenient to use
  - Use the computer hardware in an efficient manner

Two Popular Definitions of an OS

OS as a resource allocator: Manages all resources and decides between conflicting requests for efficient and fair resource use (*e.g.* accessing disk or other devices)

OS as a control system: Controls execution of programs to prevent errors and improper use of the computer (*e.g.* protects one user process from crashing another)

Computer System Structure

Computer system can be divided into four components:

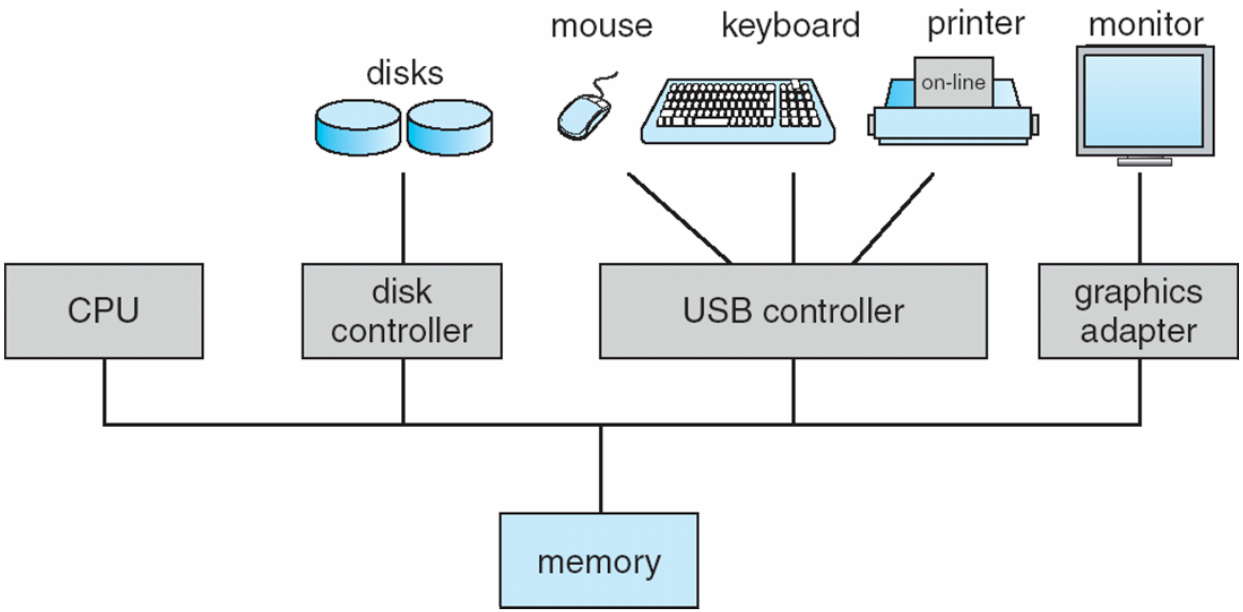
- Hardware: provides basic computing resources CPU, memory, I/O devices
- Operating system: Controls and coordinates use of hardware among various applications and users
- Application programs: define the ways in which the system resources are used to solve the computing problems of the users Word processors, compilers, web browsers, database systems, video games
- Users: People, machines, other computers

Operating System Topics

Bootstrapping of the OS

- Small bootstrap program is loaded at power-up or reboot
  - Typically stored in ROM or EPROM, generally known as firmware (*e.g.* BIOS)
- Initializes all aspects of the system (*e.g.* detects connected devices, checks memory for errors, *etc.*)
- Loads operating system kernel and starts its execution

Computer System Organisation



- One or more CPUs, device controllers connect through common bus providing access to shared memory
- CPU(s) and devices compete for memory cycles (*i.e.* to read and write memory addresses)

Computer System Operation

- I/O devices and the CPU can execute concurrently
- Each device controller (*e.g.* controller chip) is in charge of a particular device type
- Each device controller has a local buffer (*i.e.* memory store for general data and/or control registers)
- CPU moves data from/to main memory to/from controller buffers (*e.g.* write this data to the screen, read coordinates from the mouse, *etc.*)
- I/O is from the device to local buffer of controller
- Device controller informs CPU that it has finished its operation by causing an *interrupt*

Interrupts

- Interrupt transfers control to the interrupt service routine generally, through the interrupt vector, which contains the addresses of all the service routines
- Interrupt architecture must save the address of the interrupted instruction so original processing may be resumed
- Incoming interrupts are disabled while another interrupt is being processed to prevent a lost interrupt
- A trap is a software-generated interrupt caused either by an error or a user request

Storage Structure

- Main memory - only large storage media that the CPU can access directly
- Secondary storage - extension of main memory that provides large non-volatile storage capacity
- Magnetic disks - rigid metal or glass platters covered with magnetic recording material
  - Disk surface is logically divided into tracks, which are subdivided into sectors
  - The disk controller determines the logical interaction between the device and the computer
- Today also often flash memory

Caching

- Important optimisation principle, performed at many levels in a computer (in hardware, operating system, software)
- Information in use copied from slower to faster storage temporarily
- Faster storage (cache) checked first to determine if information is there
  - If it is, information used directly from the cache (fast)
  - If not, data copied to cache and used there
- Cache often smaller than storage being cached
- Cache management is an important design problem
  - Determining cache size and replacement policy

Multiprocessors

- Most systems use a single general-purpose processor
- Multiprocessor systems growing in use and importance
  - Also known as parallel systems, tightly-coupled systems
- Advantages include
  - Increased throughput
  - Economy of scale
  - Increased reliability - graceful degradation or fault tolerance
- Two architectures
  - Asymmetric Multiprocessing - CPUs have different roles, usually one is the master of the others
  - Symmetric Multiprocessing - CPUs have identical roles, sharing process queues to service *ready* processes.

Fault Handling and Protection

- Software error or request creates exception or trap, which are essentially handled as special interrupts
  - Division by zero, request for operating system service
- Other process problems include infinite loop, processes modifying each others' or the operating system's code
- Dual-mode operation allows OS to protect itself and other system components
  - User mode and kernel mode
  - Mode bit provided by hardware
    - Provides ability to distinguish when system is running user code or kernel code
    - Some instructions designated as privileged, only executable in kernel mode
    - System call changes mode to kernel, return from call resets it to user
    - Otherwise, a user process could manipulate hardware directly, leading to chaos.

Process Management

- A process is a program in execution. It is a unit of work within the system. Program is a passive entity (*e.g.* the stored code), process is an active entity.
- Process needs resources to accomplish its task
  - CPU, memory, I/O, files, initialization data
- Process termination requires reclamation of any reusable resources
- Single-threaded process has one program counter, specifying location of next instruction to execute
  - Process executes instructions sequentially, one at a time, until completion
- Multi-threaded process has one program counter per thread
- Concurrency of systems achieved by multiplexing the CPUs among the processes/threads

Process Management Activities

The operating system is responsible for the following activities in connection with process management:

- Creating and deleting both user and system processes
- Suspending and resuming processes
- Providing mechanisms for process synchronization
- Providing mechanisms for process communication
- Providing mechanisms for deadlock handling

Memory Management

- All data must be in memory before and after processing
- All instructions must be in memory in order to be executed
- Memory management determines what is in memory and when
  - Optimizing CPU utilization and computer response to users (*e.g.* avoiding excessive swapping of data between disk and memory)
- Memory management activities
  - Keeping track of which parts of memory are currently being used and by whom
  - Deciding which processes (or parts thereof) and data to move into and out of memory (*e.g.* to and from the disk - virtual memory)
  - Allocating and deallocating memory space as needed

Storage Management

- OS provides uniform, logical view of information storage
  - Abstracts physical properties to logical storage units: file and directories
- Each medium is controlled by device (*e.g.* disk drive, tape drive)
  - Varying properties include access speed, capacity, data-transfer rate, access method (sequential or random)
- File-System management
  - Access control on most systems to determine who can access what
  - OS activities include
    - Creating and deleting files and directories
    - Primitives (*i.e.* standardised functions) to manipulate files and directories
    - Mapping files within memory onto secondary storage

I/O Subsystems

- One purpose of OS is to hide peculiarities of hardware devices from the user
- I/O subsystem responsible for:
  - Memory management of I/O, including buffering (storing data temporarily while it is being transferred), caching (storing parts of data in faster storage for performance), spooling (the overlapping of output of one job with input of other jobs)
  - Abstract device-driver interface, so driver developers know how to interface their code with the OS (*e.g.* network cards, storage devices for different hardware use common interfaces).
  - Drivers for specific hardware devices

Protection and Security

- **Protection** - any mechanism for controlling access of processes or users to resources defined by the OS
- **Security** - defense of the system against internal and external attacks
  - Huge range, including denial-of-service, worms, viruses, identity theft, theft of service
- Systems generally first distinguish among users, to determine who can do what
  - User identities (user IDs, security IDs) include name and associated number, one per user
  - User ID then associated with all files, processes of that user to determine access control
  - Group identifier (group ID) allows set of users to be defined and controls managed, then also associated with each process, file
  - Privilege escalation allows user to change to effective ID with more rights

Structuring Projects

Structuring Projects

- It is infeasible to try to write any useful system as a single source file
  - Especially if we'd like to reuse parts of it in other projects.
  - Or if we'd like someone else to clearly understand its structure.
- So today we will look at how we can structure our source into a project, splitting it into multiple files and possibly libraries.

Object File Compilation and Linking

Object File Compilation and Linking

- So far, we have allowed the compiler to hide from us a crucial step in building software: the **linking step**.
- When we run `gcc hello_world.c -o hello_world`, the compiler first **compiles** all of the files (in this case `hello_world.c`) to *object* files (the actual machine code representing each C file)
- Then it automatically **links** them together into a single file (the final executable), such that references to functions and variables foreign to each object file are resolved as calls (*i.e.* jumps to addresses) into the other object files.
- Additional to our object files, compiled code from the standard libraries is also pulled into the final executable (*e.g.* the machine code of, say, `printf`).
    - In fact, for a small program, most of the code in the executable will be code from such libraries.

Object File Compilation and Linking

- Often we require explicit control over the build process, so we first build the object files for each C file, using the *compile-only* flag, `-c`:
  - `gcc -c hello_world.c -o hello_world.o`
- Then we link the object files into the final executable (in this case `my_app`):
  - `gcc -o my_app hello_world.o linked_list.o ...`
- For the interested, if you run `objdump -d <some_compiled_binary>` on some object file or executable you will see a list of functions defined within the file along with their disassembled code.

Today's Code

Today's Code

- Today we will look at the code of a simple application, `todo_list`, that allows the user to enter a list of items, such that we will get a better understanding of what is involved in putting together a larger project.
  - First we will see how we could implement it as multiple files, looking at how header files my be defined and used.
  - Then we will see how we could split off the linked list code into to a library which can be re-used in other applications.

Building a Project with Make

Building a Project with Make

- We can use the tool `make` to help us to build large software projects.
- `make` allows us to automate the running of long commands, through the definition rules in a special file called `Makefile`.

Structure of simple Makefile :

- Starts with declarations (assignment of values to variables)
- Then have list of targets and commands which re-create the target
  Need to have TAB-character at beginning of line containing the command!
- Can call `make` in directory `<dir>` via
  `make -C <dir>`

- Such a list of targets consists of one line containing target and dependencies
- dependencies are files which need to be present and newer than the target
- commands will be executed if target needs to be re-generated variables may be used

Conventions:

- Have target `all` which makes everything in this directory normally first target
- Have also target `clean` which removes all targets and temporary files created

Client-server architectures

- Have *server* process, which waits for client requests and processes them once they arrive
- Multiple clients may connect and request service
- Standard paradigm for services on the internet
- Have in addition to IP-addresses *port numbers* assigned to particular services
- Examples:
- Port 80 for http
- Port 25 for sending mail (smtp)
- Port 143 for reading mail via imap
- Port assignments listed in `/etc/services`
- Client-Server architecture implemented via *sockets*

Sockets

How to setup a client/server connection via sockets:
Initialisation phase for the server:

- Server creates endpoint via `socket`-system call
- Server specifies port number and protocol in structure `sockaddr_in`
- Server assigns information in `sockaddr_in`-structure to socket via `bind`-system call

Now server waits for incoming connection via `accept`-system call
When connection received, server reads data via `read`-system call and writes data back via `write`-system call
When server is finished with current connection, server closes connection via `close`-system call

Concurrency

- Good handling of concurrency vital for implementing sockets
- Will use `pthread`-library for this
  Library implements kernel-level threads together with synchronisation mechanisms
- Key point: Program may create arbitrary number of threads, which share memory and may run concurrently
- Synchronisation achieved by mutual exclusion:
  A section of code satisfies mutual exclusion if it can executed by only one thread at a time, and in addition no switching between threads happens while this section of code is executed.
- Such a section of code is called critical section

Important operations on threads

- `pthread_create` creates new threads
- `pthread_exit` exits threads
- `pthread_mutex_lock`(mutex) starts a critical section involving those threads using the specified mutex
- `pthread_mutex_unlock`(mutex) ends a critical section involving those threads using the specified mutex
- `pthread_attr_init` sets up default attributes
- `pthread_join` wait for termination for another thread
- `pthread_attr_setdetachstate` cause thread to terminate immediately when `pthread_exit` is called
  $\Rightarrow$ cannot use `pthread_join`

OS Services

How do Users and Processes interact with the Operating System?

- **Users** interact indirectly through a collection of system programs that make up the operating system interface. The interface could be:
  - A GUI, with icons and windows, *etc.*
  - A command-line interface for running processes and scripts, browsing files in directories, *etc.*
  - Or, back in the olden days, a non-interactive batch system that takes a collection of jobs, which it proceeds to churn through (*e.g.* payroll calculations, market predictions, *etc.*)
- **Processes** interact by making *system calls* into the operating system proper (*i.e.* the *kernel*).
  - Though we will see that, for stability, such calls are not direct calls to kernel functions.

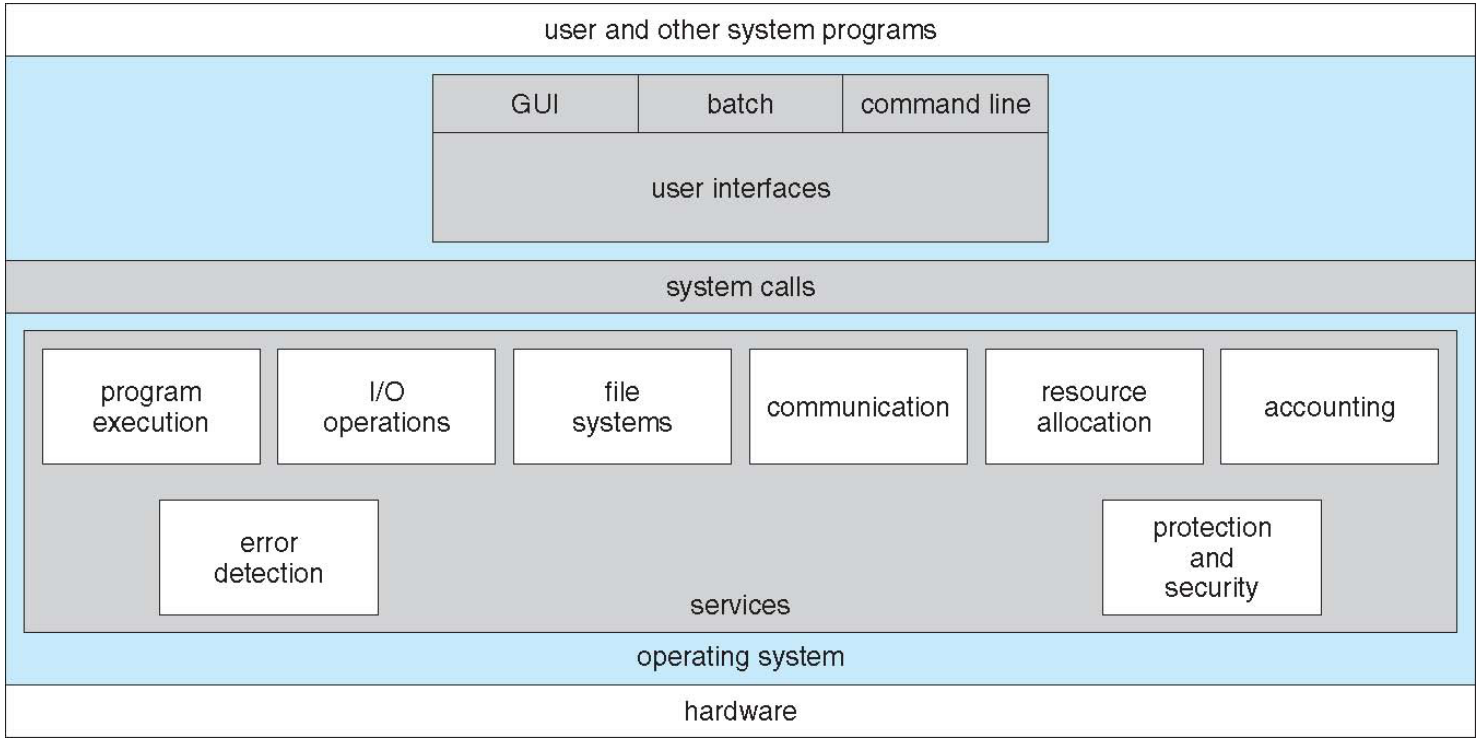
Services for Processes

- Typically, operating systems will offer the following services to processes:
  - **Program execution**: The system must be able to load a program into memory and to run that program, end execution, either normally or abnormally (indicating error)
  - **I/O operations**: A running program may require I/O, which may involve a file or an I/O device
  - **File-system manipulation**: Programs need to read and write files and directories, create and delete them, search them, list file Information, permission management.
  - **Interprocess Communication (IPC)**: Allowing processes to share data through message passing or shared memory

Services for the OS Itself

- Typically, operating systems will offer the following internal services:
  - **Error handling**: what if our process attempts a divide by zero or tries to access a protected region of memory, or if a device fails?
  - **Resource allocation**: Processes may compete for resources such as the CPU, memory, and I/O devices.
  - **Accounting**: *e.g.* how much disk space is this or that user using? how much network bandwidth are we using?
  - **Protection and Security**: The owners of information stored in a multi-user or networked computer system may want to control use of that information, and concurrent processes should not interfere with each other

OS Structure with Services



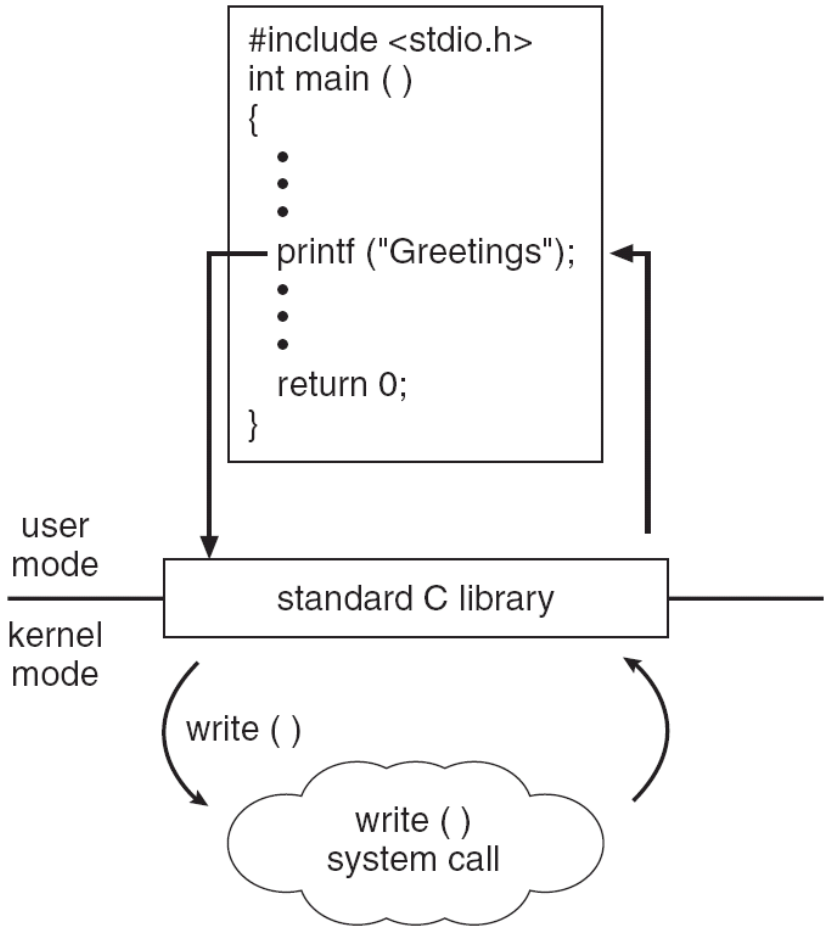
System Calls

- Programming interface to the services provided by the OS (*e.g.* open file, read file, *etc.*)
- Typically written in a high-level language (C or C++)
- Mostly accessed by programs via a high-level Application Program Interface (API) rather than direct system call.
- Three most common APIs are Win32 API for Windows, POSIX API for UNIX-based systems (including virtually all versions of UNIX, Linux, and Mac OS X)
- So why use APIs in user processes rather than system calls directly?
  - Since system calls result in execution of privileged kernel code, and since it would be crazy to let the user process switch the CPU to privileged mode, we must make use of the low-level hardware trap instruction, which is cumbersome for user-land programmers.
  - The user process runs the trap instruction, which will switch CPU to privileged mode and jump to a kernel pre-defined address of a generic system call function, hence the transition is controlled by the kernel.
  - Also, APIs can allow for backward compatibility if system calls change with the release of a OSS

System calls provided by Windows and Linux

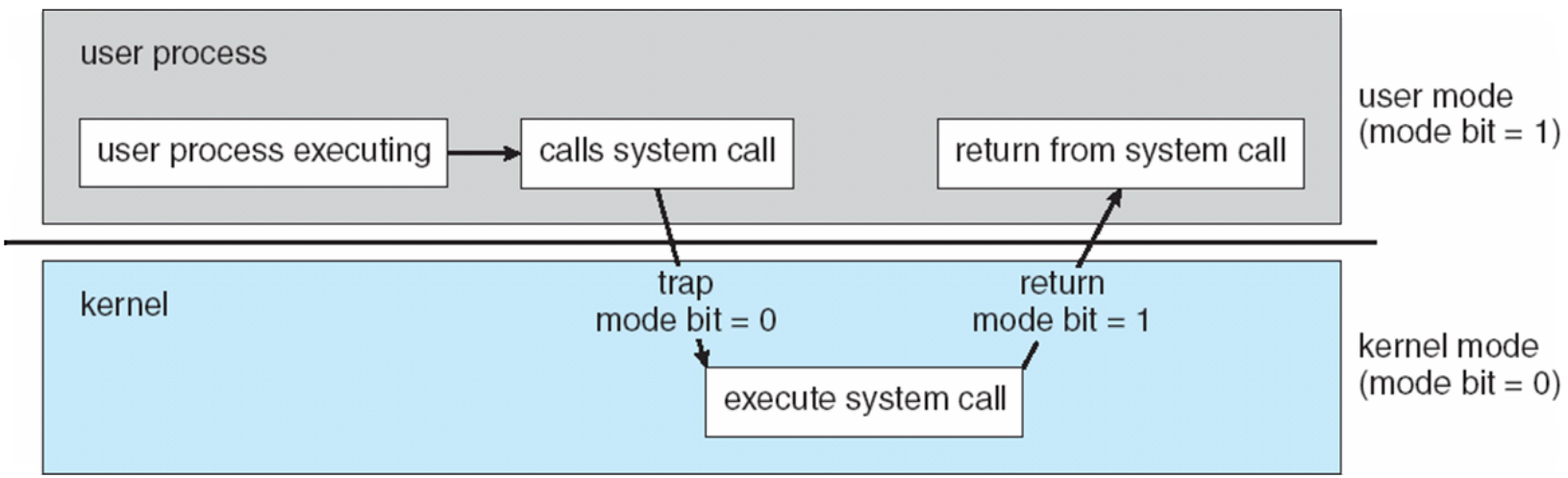
	Windows	Unix
Process Control	CreateProcess() ExitProcess() WaitForSingleObject()	fork() exit() wait()
File Manipulation	CreateFile() ReadFile() WriteFile() CloseHandle()	open() read() write() close()
Device Manipulation	SetConsoleMode() ReadConsole() WriteConsole()	ioctl() read() write()
Information Maintenance	GetCurrentProcessID() SetTimer() Sleep()	getpid() alarm() sleep()
Communication	CreatePipe() CreateFileMapping() MapViewOfFile()	pipe() shmget() mmap()
Protection	SetFileSecurity() InitlializeSecurityDescriptor() SetSecurityDescriptorGroup()	chmod() umask() chown()

An example of a System Call



Trapping to the Kernel

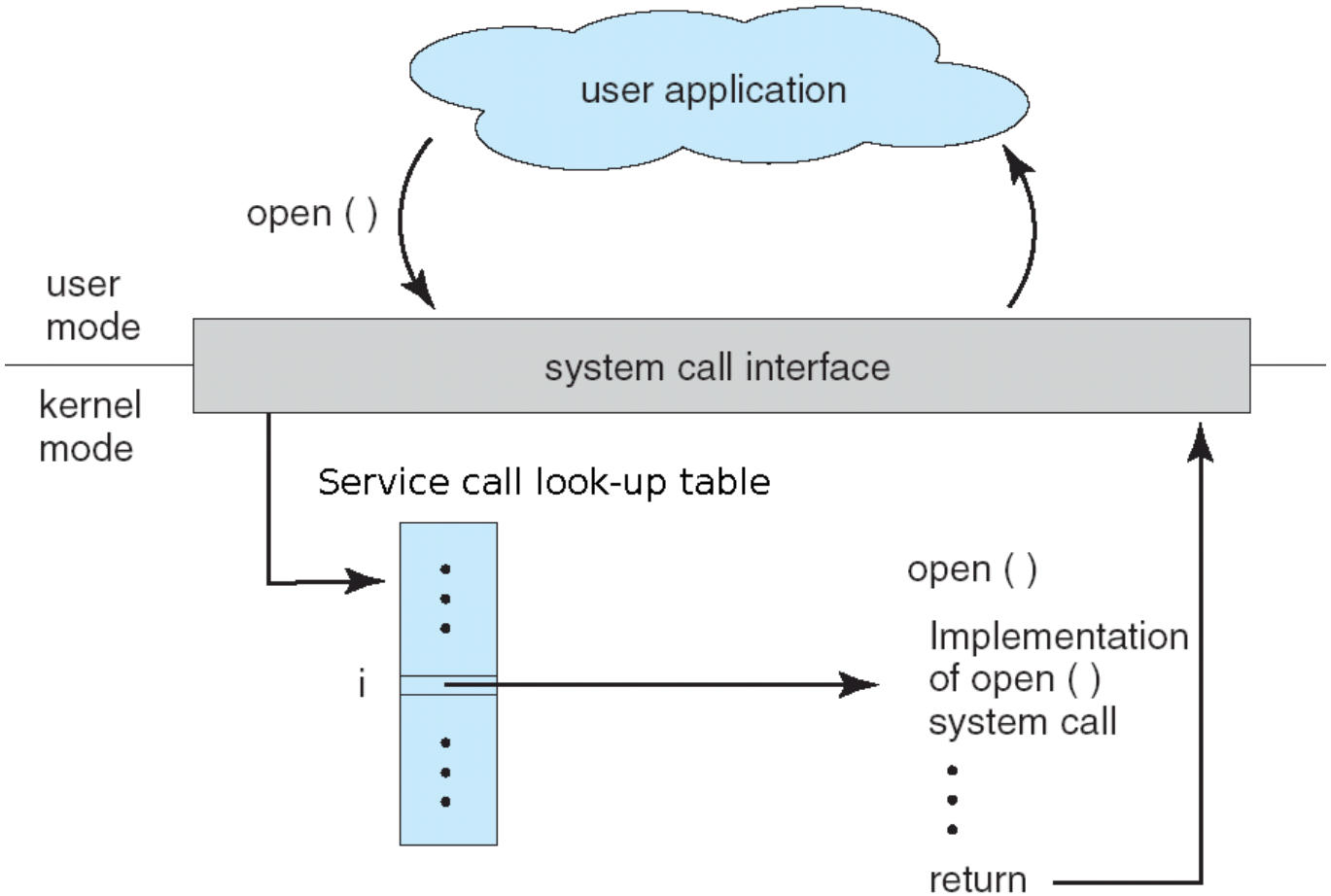
- The user process calls the system call wrapper function from the standard C library
- The wrapper function issues a low-level *trap* instruction (in assembly) to switch from user mode to kernel mode



Trapping to the Kernel

- To get around the problem that no call can directly be made from user space to a specific function in kernel space:
  - Before issuing the trap instruction, an index is stored in a well known location (*e.g.* CPU register, the stack, *etc.*).
  - Then, once switched into kernel space, the index is used to look up the desired kernel service function, which is then called.
- Some function calls may take arguments, which may be passed as pointers to structures via registers.

Trapping to the Kernel



OS Architecture

OS Design

- An OS is possibly the most complex system that a computer will run, and it not yet clear (nor may it ever be) how to design an operating system to best meet the many and varied requirements placed on it.
- The internal structure of OSes can vary widely
- We can start by defining goals and specifications:
  - Affected by choice of hardware, type of system
  - User goals and System goals
    - User goals - operating system should be convenient to use, easy to learn, reliable, safe, and fast
    - System goals - operating system should be easy to design, implement, and maintain, as well as flexible, reliable, error-free, easy to extend, and efficient
- OS architectures have evolved over the years, generally trying to better balance efficiency and stability

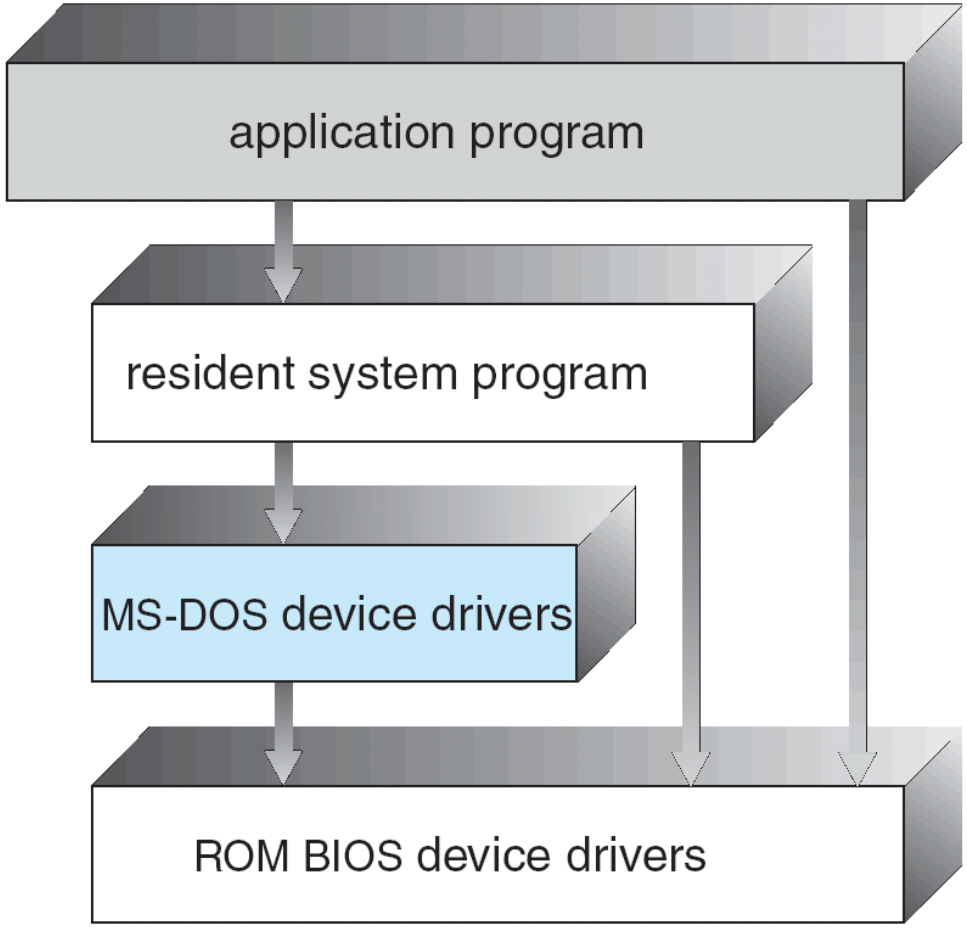
Separation of Policies and Mechanisms

- **Policy**: What will be done?
- **Mechanism**: How to do it?
- The separation of policy from mechanism is a very important principle, it allows maximum flexibility if policy decisions are to be changed later
- An architecture that supports extendible file systems is a good example
    - Rather than hard code a particular file system into the kernel code, create an abstract file-system interface with sufficient flexibility that it can accommodate many file system implementations, eg: NTFS, EXT, FAT.

MS-DOS

- MS-DOS - written to provide the most functionality in the least space
  - Not divided into modules
  - Although MS-DOS has some structure, its interfaces and levels of functionality are not well separated - the highest layer is allowed to call the lowest layer.
  - So, it was easy for a user process (accidentally or on purpose) to de-stabilise the whole system, which is often what happened, even up until MS-DOS based Windows ME.

MS-DOS


application program
resident system program
MS-DOS device drivers
ROM BIOS device drivers

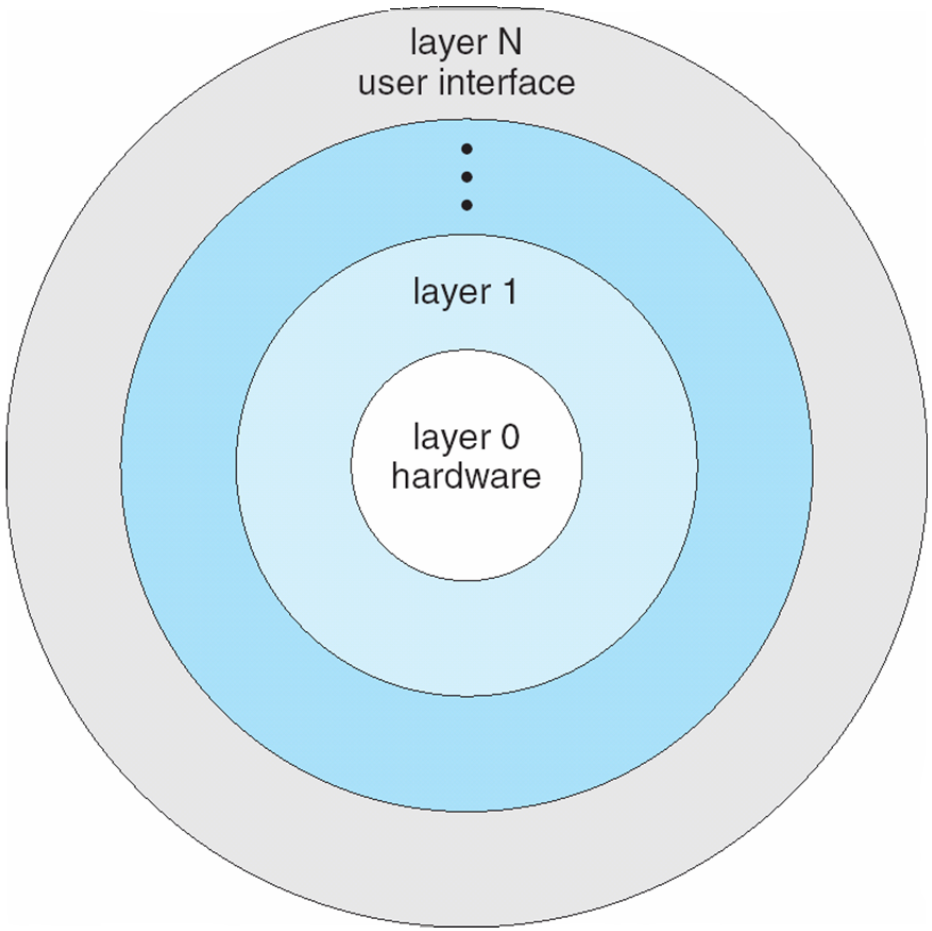
Strict Layered

- The operating system is divided into a number of layers (levels), each built on top of lower layers. The bottom layer (layer 0), is the hardware; the highest (layer N) is the user interface.
- With modularity, layers are selected such that each uses functions (operations) and services of only lower-level layers
  - Importantly for stability, modern CPUs offer protected mode, which means transition between layers is controlled by hardware
  - Attempts of software to run instructions or access memory regions that are higher privilege will result in the CPU raising a hardware exception (*e.g.* divide by zero, attempt to access hardware directly, *etc.*)
- Lends itself to simpler construction, since layers have well-defined functionality and can be tested independently or altered with minimal impact on the rest of the OS (*e.g.* lowest level could be adapted to different CPU architectures with minimal impact on higher layers)

Strict Layered

- Consider a file system. The actual file-system can be implemented in a layer above a layer that reads and writes raw data to a particular disk device, such that the file system will work with any device implemented by the lower layer (*e.g.* USB storage device, floppy disk, hard disk, *etc.*).
- In practice, however, it can be difficult to decide how many layers to have and what to put in each layer to build a general purpose operating system.

Strict Layered

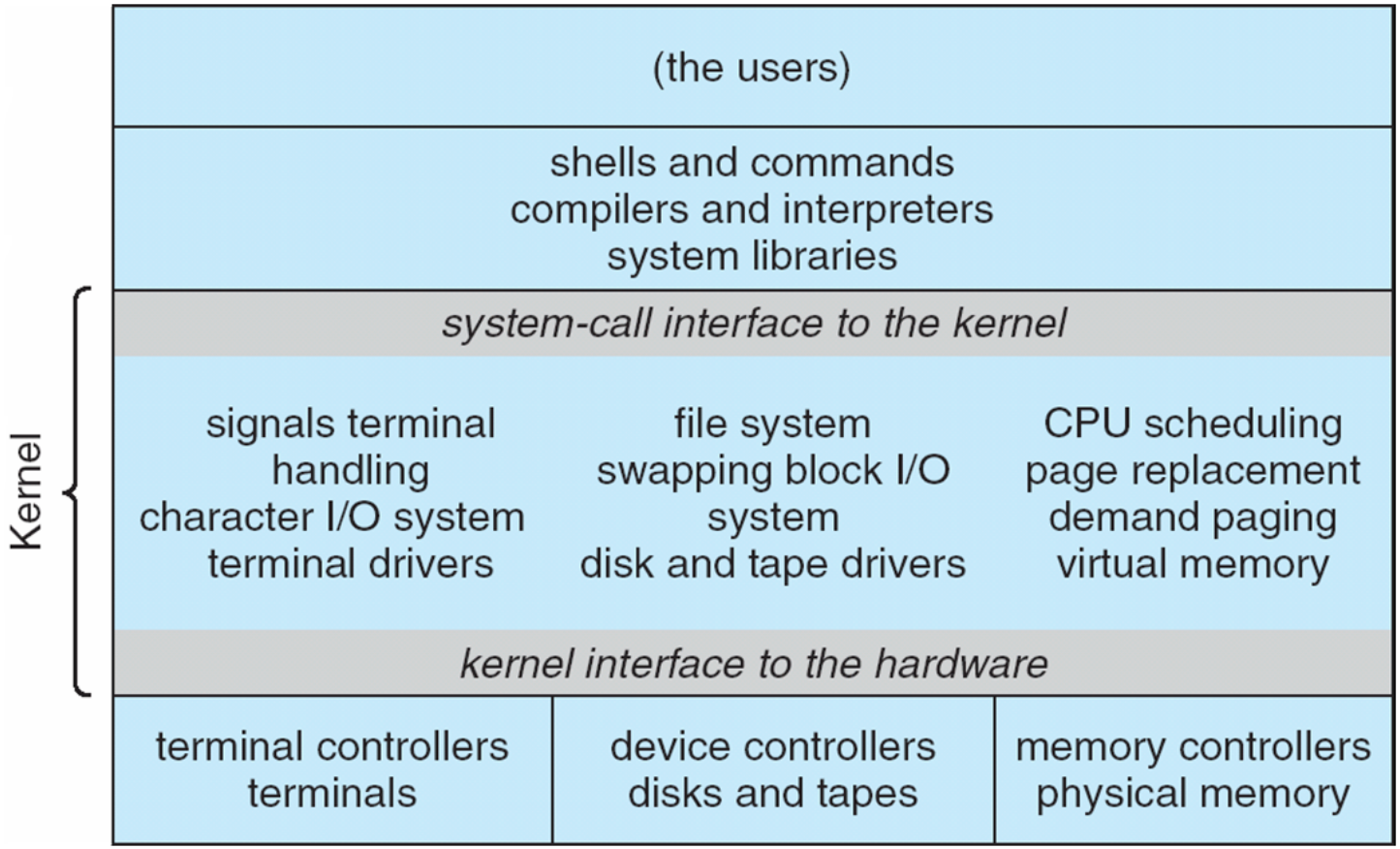


Traditional UNIX

UNIX - one big kernel

- Consists of everything below the system-call interface and above the physical hardware
- Provides the file system, CPU scheduling, memory management, and other operating-system functions; a large number of functions for one level
- Limited to hardware support compiled into the kernel.

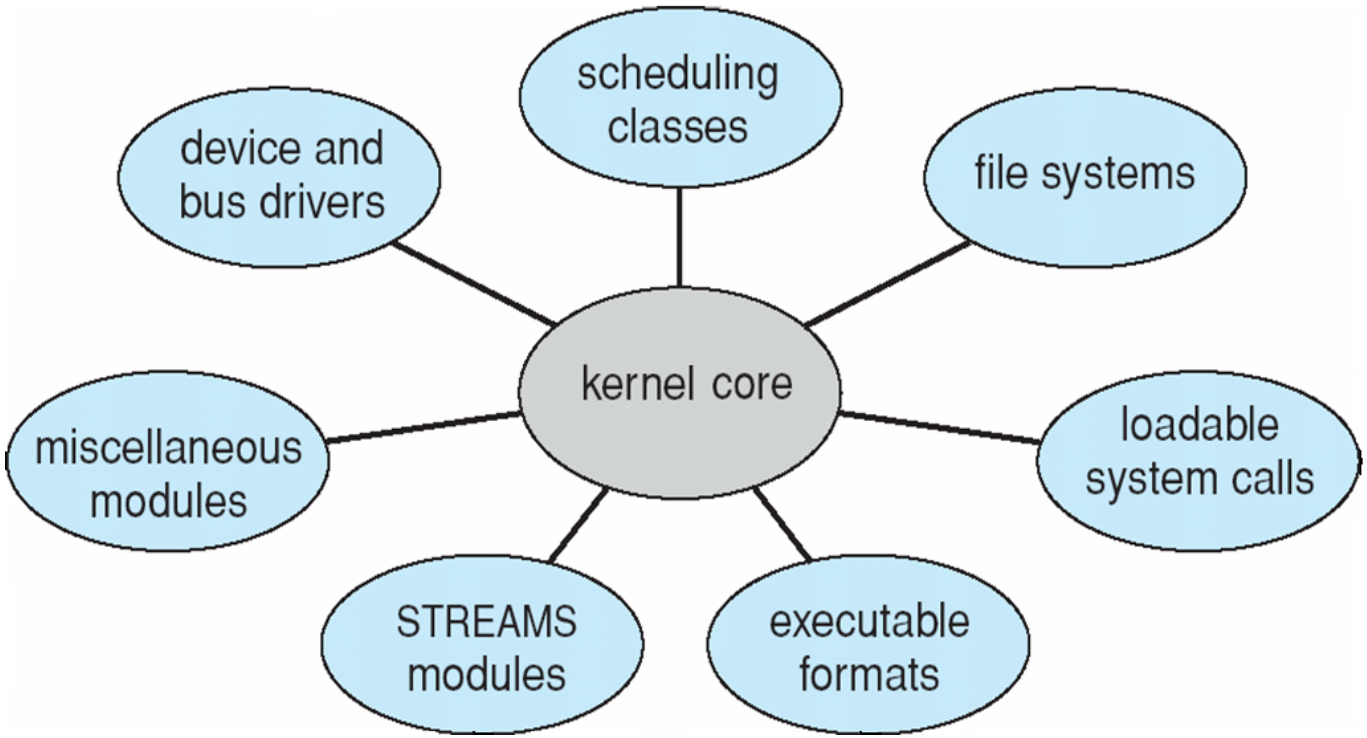
Traditional UNIX



Modular Kernel

- Most modern operating systems implement kernel modules
  - Uses object-oriented–like approach
  - Each core component is separate
  - Each talks to the others over known interfaces
  - Each is loadable as needed within the kernel, so you could download a new device driver for your OS and load it at run-time, or perhaps when a device is plugged in
- Overall, similar to layered architecture but with more flexibility, since all require drivers or kernel functionality need not be compiled into the kernel binary.
- Note that the separation of the modules is still only logical, since all kernel code (including dynamically loaded modules) runs in the same privileged address space (a design now referred to as monolithic), so I could write a module that wipes out the operating system no problem.
  - This leads to the benefits of micro-kernel architecture, which we will look at soon

Modular Kernel



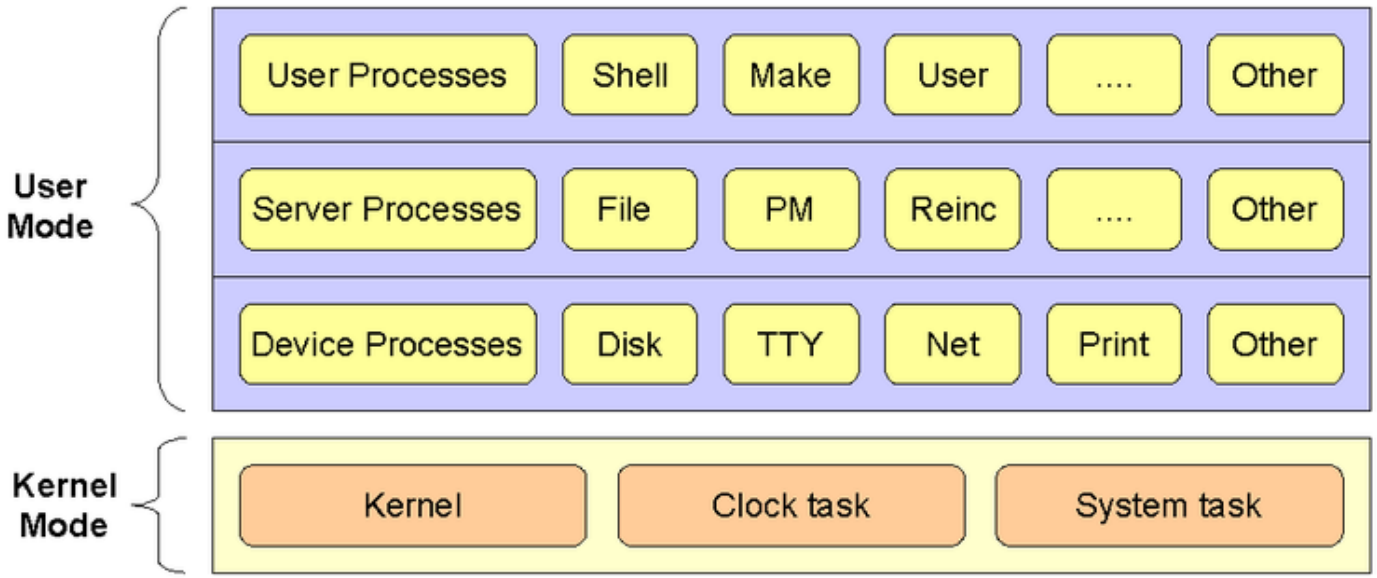
Microkernel

- Moves as much as possible from the kernel into less privileged "user" space (*e.g.* file system, device drivers, *etc.*)
- Communication takes place between user modules using message passing
  - The device driver for, say, a hard disk device can run all logic in user space (*e.g.* decided when to switch on and off the motor, queuing which sectors to read next, *etc.*)
  - But when it needs to talk directly to hardware using privileged I/O port instructions, it must pass a message requesting such to the kernel.

Microkernel

- Benefits:
  - Easier to develop microkernel extensions
  - Easier to port the operating system to new architectures
  - More reliable (less code is running in kernel mode) - if a device driver fails, it can be re-loaded
  - More secure, since kernel is less-complex and therefore less likely to have security holes.
  - The system can recover from a failed device driver, which would usually cause "a blue screen of death" in Windows or a "kernel panic" in linux.
- Drawbacks:
  - Performance overhead of user space to kernel space communication
- The Minix OS and L3/L4 are examples of microkernel architecture

Microkernel: MINIX



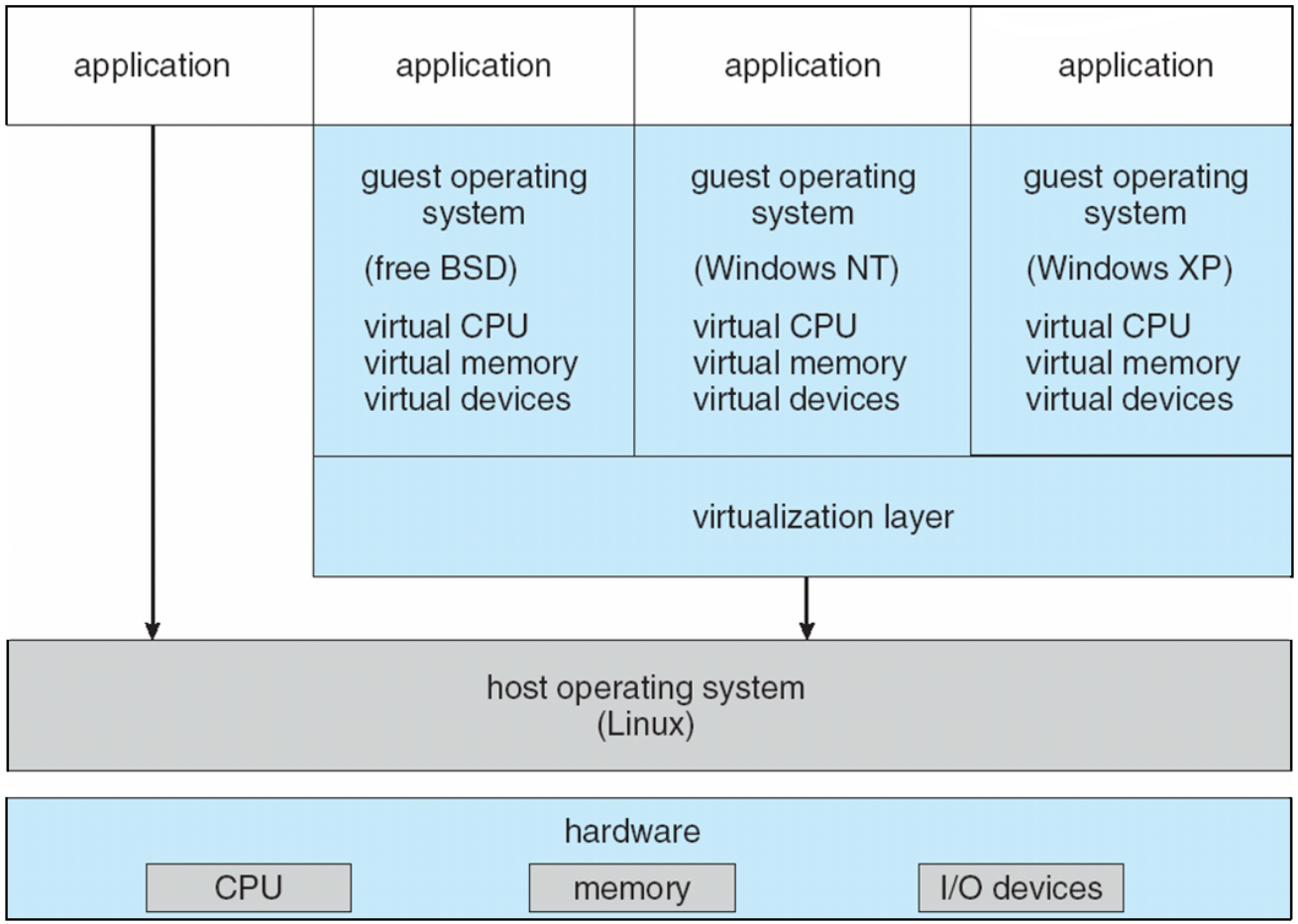
The MINIX 3 Microkernel Architecture

Virtual Machines

Virtual Machines

- A virtual machine takes the layered approach to its logical conclusion. It treats hardware and the operating system kernel as though they were all hardware
- A virtual machine provides an interface identical to the underlying bare hardware
- The operating system host creates the illusion that a process has its own processor and (virtual memory)
- Each guest is provided with a (virtual) copy of underlying computer, so it is possible to install, say, Windows XP as a guest operating system on Linux.

VM Architecture



Virtual Machines: History and Benefits

- First appeared commercially in IBM mainframes in 1972
- Fundamentally, multiple execution environments (different operating systems) can share the same hardware
- Protected from one another, so no interference
  - Some sharing of files can be permitted, controlled
  - Communicate with one another other and with other physical systems via networking
- Useful for development, testing, especially OS development, where it is trivial to revert an accidentally destroyed OS back to a previous stable snapshot.

Virtual Machines: History and Benefits

- Consolidation of many low-resource use systems onto fewer busier systems
- "Open Virtualization Format" (OVF): standard format of virtual machines, allows a VM to run within many different virtual machine (host) platforms.
- Not to be confused with emulation, where guest instructions are run within a process that pretends to be the CPU (*e.g.* Bochs and QEMU). In virtualisation, the goal is to run guest instructions directly on the host CPU, meaning that the guest OS must run on the CPU architecture of the host.

Para-virtualisation

- Presents guest with system similar but not identical to hardware (*e.g.* Xen Hypervisor)
- Guest OS must be modified to run on paravirtualized 'hardware'
  - For example, the kernel is recompiled with all code that uses privileged instructions replaced by hooks into the virtualisation layer
  - After an OS has been successfully modified, para-virtualisation is very efficient, and is often used for providing low-cost rented Internet servers (*e.g.* Amazon EC2, Rackspace)

VMWare Architecture

- VMWare implements full virtualisation, such that guest operating systems do not require modification to run upon the virtualised machine.
- The virtual machine and guest operating system run as a user-mode process on the host operating system

VMWare Architecture

- As such, the virtual machine must get around some tricky problems to convince the guest operating system that it is running in privileged CPU mode when in fact it is not.
  - Consider a scenario where a process of the guest operating system raises a divide-by-zero error.
  - Without special intervention, this would cause the host operating system immediately to halt the virtual machine process rather than the just offending process of the guest OS.
  - So VMWare must look out for troublesome instructions and replace them at run-time with alternatives that achieve the same effect within user space, albeit with less efficiency
  - But since usually these instructions occur only occasionally, many instructions of the guest operating system can run unmodified on the host CPU.

Memory Management

Memory Management

Management of a limited resource:
(Memory hunger of applications increases with capacity!)
$\Rightarrow$ Sophisticated algorithms needed, together with support from HW and from compiler and loader.

Key point: program's view memory is set of memory cells starting at addresss 0x0 and finishing at some value (logical address)

Hardware: have set of memory cells starting at address 0x0 and finishing at some value (physical address)

Want to be able to store memory of several programs in main memory at the same time

need suitable mapping from logical addresses to physical addresses:

- at compile time: absolute references are generated (eg MS-DOS `.com`-files)
- at load time: can be done by special program
- at execution time: needs HW support

Address mapping can be taken one step further:
dynamic linking: use only one copy of system library
$\Rightarrow$ OS has to help: same code accessible to more than one process

Swapping

If memory demand is too high, memory of some processes is transferred to disk

Usually combined with scheduling: low priority processes are swapped out

Problems:

- Big transfer time
- What to do with pending I/O?

First point reason why swapping is not principal memory management technique

Fragmentation

Swapping raises two problems:

- over time, many small holes appear in memory (external fragmentation)
- programs only a little smaller than hole ⇒ leftover too small to qualify as hole (internal fragmentation)

Strategies for choosing holes:

- First-fit: Start from beginning and use first available hole
- Rotating first fit: start after last assigned part of memory
- Best fit: find smallest usable space
- Buddy system: Free holes are administered according to tree structure; smallest possible chunk used

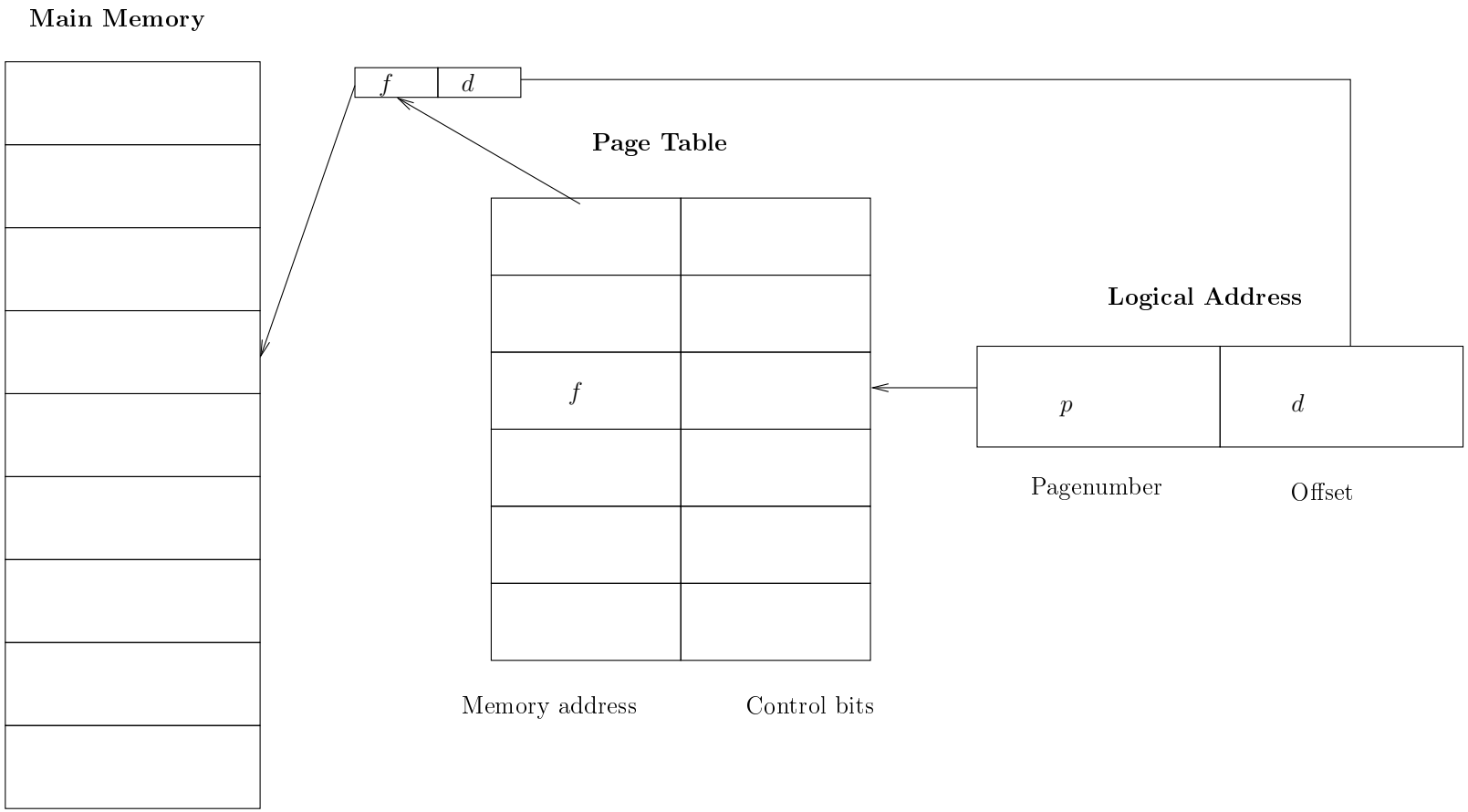
Paging

Alternative approach: Assign memory of a fixed size (page)
$\Rightarrow$ avoids external fragmentation
Translation of logical address to physical address done via page table

Hardware support mandatory for paging:
If page table small, use fast registers. Store large page tables in main memory, but cache most recently used entries

Instance of a general principle:
Whenever large lookup tables are required, use cache (small but fast storage) to store most recently used entries

Memory protection easily added to paging:
protection information stored in page table


Main Memory
f
d
Page Table
f
Logical Address
p
d
Pagenumber
Offset
Memory address
Control bits

Segmentation

Idea: Divide memory according to its usage by programs:

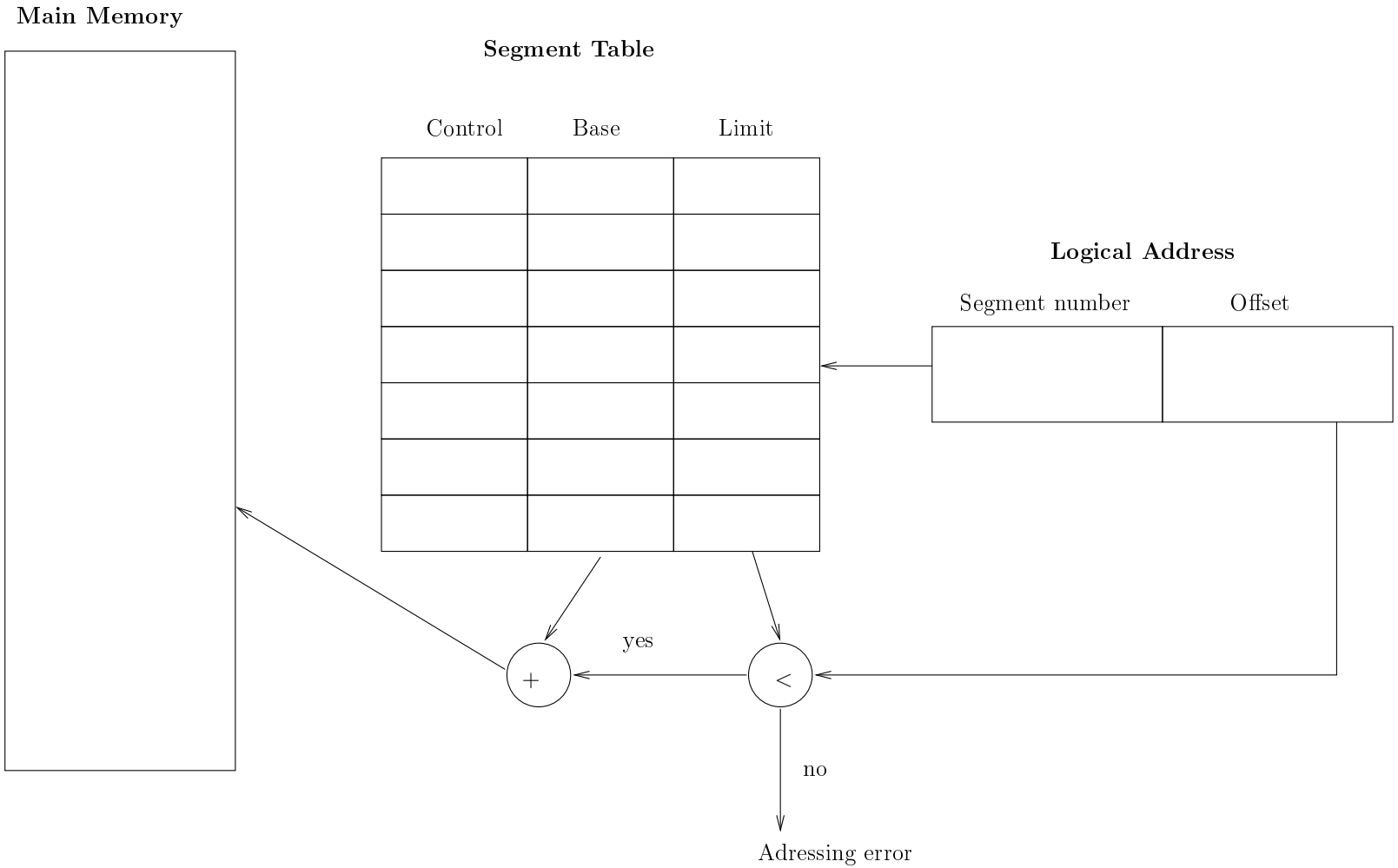
- Data: mutable, different for each instance
- Program Code: immutable, same for each instance
- Symbol Table: immutable, same for each instance, only necessary for debugging

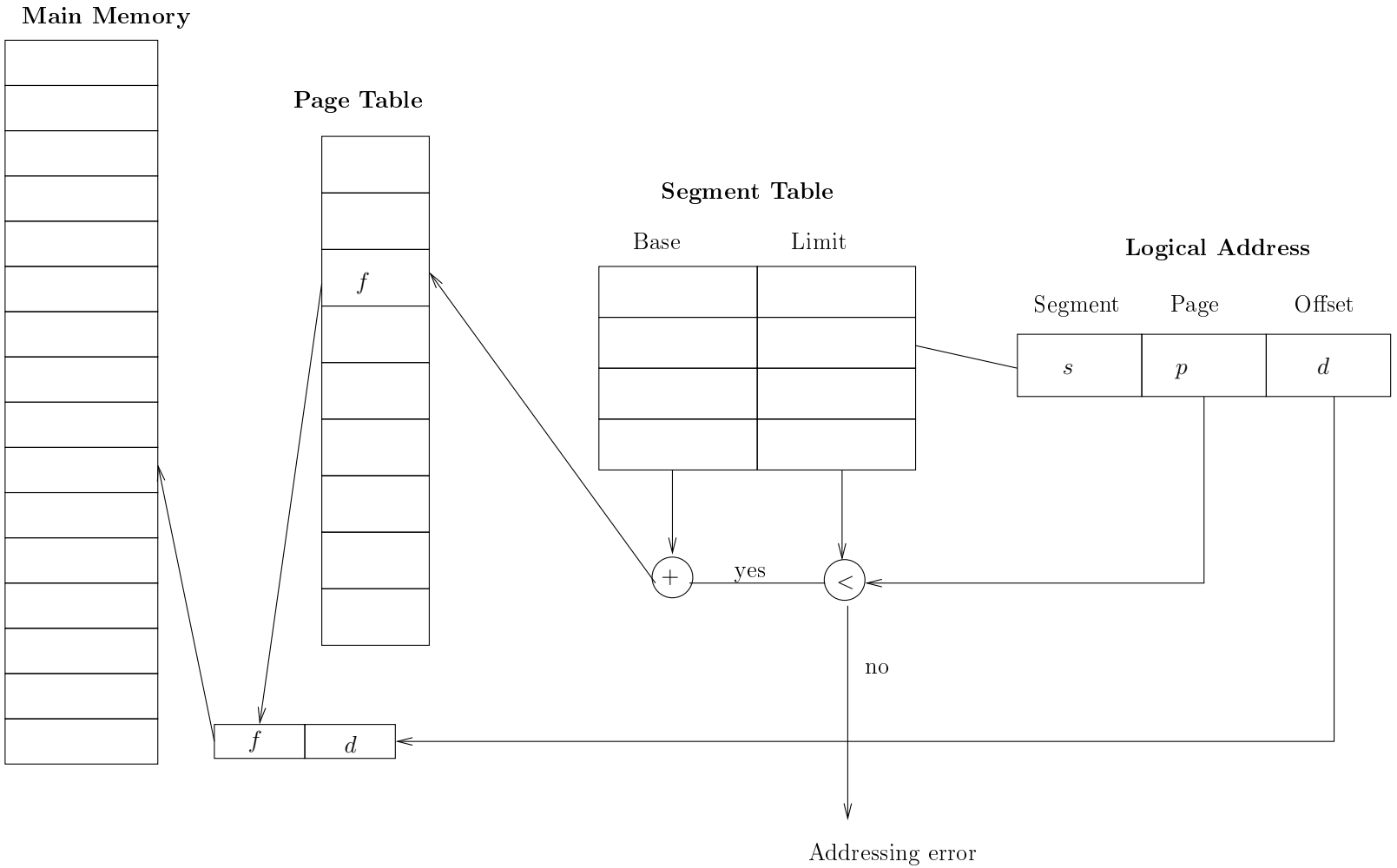
Requires again HW support

can use same principle as for paging, but have to do overflow check

Paging motivated by ease of allocation, segmentation by use of memory

$\Rightarrow$ combination of both works well (eg 80386)


Main Memory
Segment Table
Control
Base
Limit
Logical Address
Segment number
Offset
yes
+
<
no
Adressing error


Main Memory
Page Table
f
Segment Table
Base
Limit
Logical Address
Segment
Page
Offset
s
p
d
+
yes
<
no
f
d
Addressing error

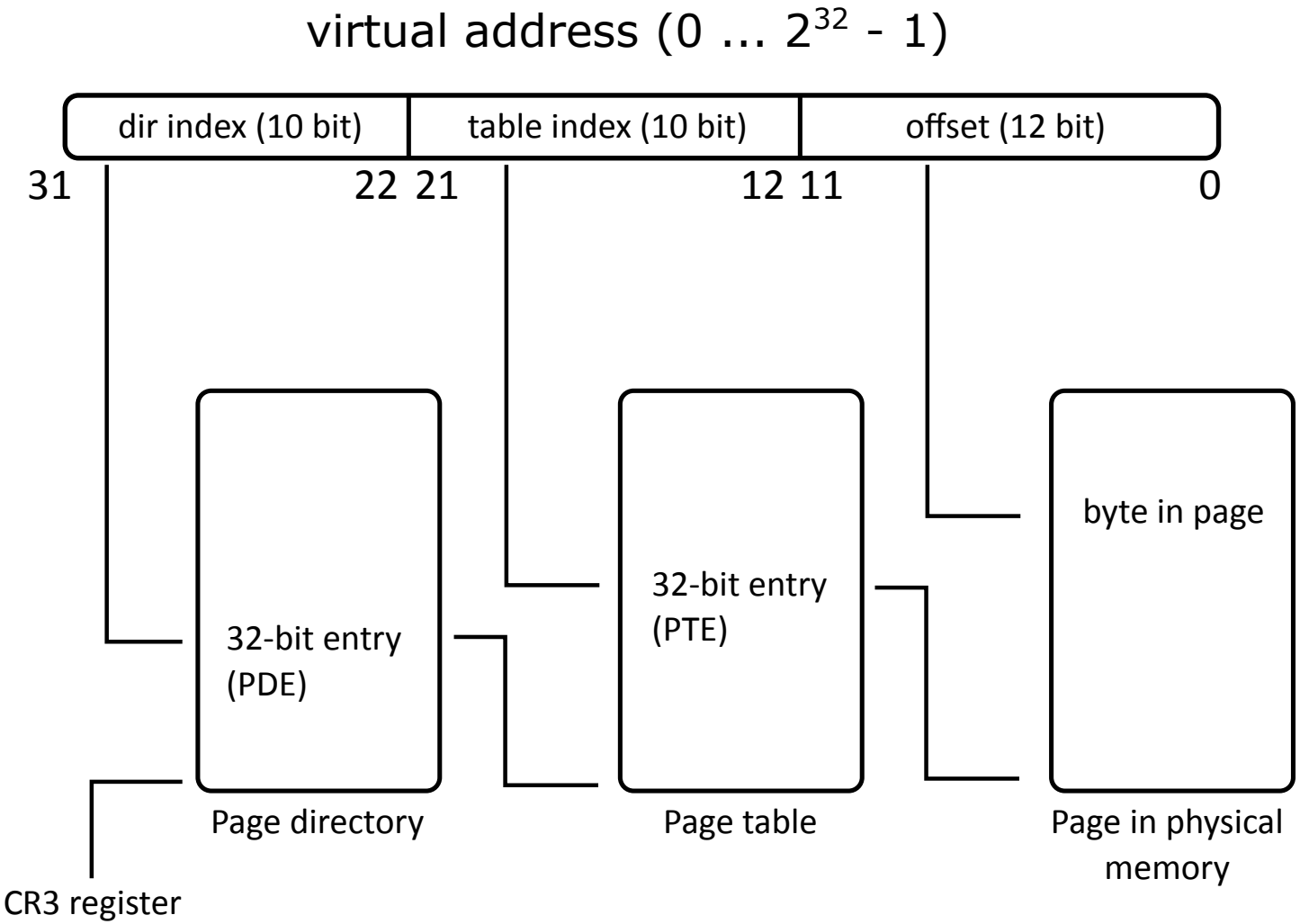
Virtual memory

Idea: complete separation of logical and physical memory
$\Rightarrow$ Program can have extremely large amount of virtual memory

Generalisation of paging and segmentation: multi-level page tables
rationale: most programs use only small fraction of memory intensively.

Efficient implementation tricky
Reason: Enormous difference between
- memory access speed (ca. 60ns)
- disk access speed (ca. 6ms)

Factor 100,000 !!


virtual address (0 ... 2^{32} - 1)
dir index (10 bit)
table index (10 bit)
offset (12 bit)
31
22 21
12 11
0
32-bit entry (PDE)
32-bit entry (PTE)
byte in page
CR3 register
Page directory
Page table
Page in physical memory

Demand Paging

Virtual memory implemented as demand paging: memory divided into units of same length (pages), together with valid/invalid bit

Two strategic decisions to be made:

- Which process to "swap out" (move whole memory to disk and block process): swapper
- which pages to move to disk when additional page is required: pager

Minimisation of rate of page faults (page has to be fetched from memory) crucial

If we want 10% slowdown due to page fault, require fault rate $p < 10^{-6}$!!

Page replacement algorithms

1.) FIFO:
easy to implement, but does not take locality into account
Further problem: Increase in number of frames can cause increase in number of page faults (Belady's anomaly)

2.) Optimal algorithm:
select page which will be re-used at the latest time (or not at all)
$\Rightarrow$ not implementable, but good for comparisons

3.) Least-recently used:
use past as guide for future and replace page which has been unused for the longest time

Problem: Requires a lot of HW support
Possibilities:
-Stack in microcode
-Approximation using reference bit: HW sets bit to 1 when page is referenced.
Now use FIFO algorithm, but skip pages with reference bit 1, resetting it to 0
$\Rightarrow$ Second-chance algorithm

Thrashing

If process lacks frames it uses constantly, page-fault rate very high.
$\Rightarrow$ CPU-throughput decreases dramatically.
$\Rightarrow$ Disastrous effect on performance.

Two solutions:
1.) Working-set model (based on locality):
Define working set as set of pages used in the most recent Δ page references
keep only working set in main memory
$\Rightarrow$ Achieves high CPU-utilisation and prevents thrashing

Difficulty: Determine the working set!
Approximation: use reference bits; copy them each 10,000 references and define working set as pages with reference bit set.

2.) Page-Fault Frequency:
takes direct approach:
- give process additional frames if page frequency rate high
- remove frame from process if page fault rate low

Memory Management in the Linux Kernel

Have only four segments in total:

- Kernel Code
- Kernel Data
- User Code
- User Data

Paging used as described earlier
Have elaborate permission system for pages

Kernel memory and user memory

Have separate logical addresses for kernel and user memory
For 32-bit architectures (4 GB virtual memory):

- kernel space address are the upper 1 GB of address space ($\geq$ 0xC0000000)
- user space addresses are 0x0 to 0xBFFFFFFF (lower 3 GB)

kernel memory always mapped but protected against access by user processes

Kernel memory and user memory

For 64-bit architectures:

- kernel space address are the upper half of address space ($\geq$ 0x8000 0000 0000 0000)
- user space addresses are 0x0 to 0x7fff ffff ffff, starting from above.

kernel memory always mapped but protected against access by user processes

Page caches

Experience shows: have repeated cycles of allocation and freeing same kind of objects (eg inodes, dentries)
can have pool of pages used as cache for these objects (so-called slab cache)
cache maintained by application (eg file system)
`kmalloc` uses slab caches for commonly used sizes

Linux kernel programming

Structure of kernel

Simplified structure of kernel:

```
initialise data structures at boot time;
while (true) {
    while (timer not gone off) {
        assign CPU to suitable process;
        execute process;
    }
    select next suitable process;
}
```

Kernel programming

Kernel has access to all resources
Kernel programs not subject to any constraints for memory acces or hardware access
$\Rightarrow$ faulty kernel programs can cause system crash

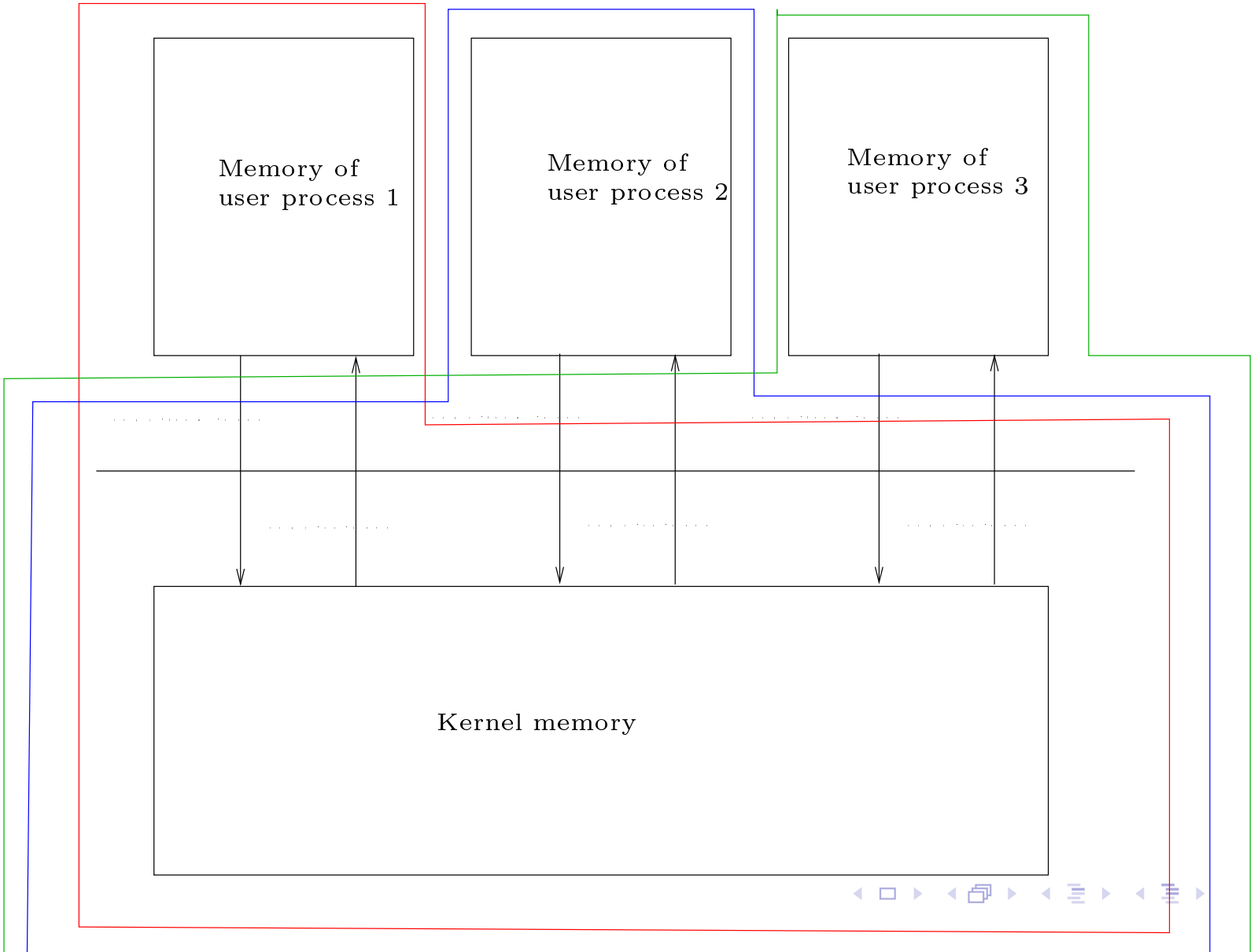
Interaction between kernel and user programs

Kernel provides its functions only via special functions, called system calls

standard C-library provides them

Have strict separation of kernel data and data for user programs

$\Rightarrow$ need explicit copying between user program and kernel (`copy_to_user()`, `copy_from_user()`)


Memory of
user process 1
Memory of
user process 2
Memory of
user process 3
Kernel memory

In addition, have interrupts:
kernel asks HW to perform certain action
HW sends interrupt to kernel which performs desired action
interrupts must be processed quickly
$\Rightarrow$ any code called from interrupts must not sleep

Linux kernel modes

Structure of kernel gives rise to two main modes for kernel code:

- process context: kernel code working for user programs by executing a system call
- interrupt context: kernel code handling an interrupt (eg by a device)

have access to user data only in process context
Any code running in process context may be pre-empted at any time by an interrupt
Interrupts have priority levels
Interrupt of lower priority are pre-empted by interrupts of higher priority

Kernel modules

can add code to running kernel
useful for providing device drivers which are required only if hardware present
`modprobe` inserts module into running kernel
`rmmod` removes module from running kernel (if unused)
`lsmod` lists currently running modules

Concurrency issues in the kernel

Correct handling concurrency in the kernel important:
Manipulation of data structures which are shared between

- code running in process mode and code running in interrupt mode
- code running in interrupt mode

must happen only within critical regions
In multi-processor system even manipulation of data structures shared between code running in process context must happen only within critical sections

Achieving mutual exclusion

Two ways:

- Semaphores/Mutex: when entering critical section fails, current process is put to sleep until critical region is available
  $\Rightarrow$ only usable if all critical regions are in process context
  Functions: `DEFINE_MUTEX()`, `mutex_lock()`, `mutex_unlock()`
- Spinlocks: processor tries repeatedly to enter critical section
  Usable anywhere
  Disadvantage: Have busy waiting
  Functions: `spin_lock_init()`, `spin_lock()`, `spin_unlock()`

Programming data transfer between userspace and kernel

Linux maintains a directory called `proc` as interface between user space and kernel
Files in this directory do not exist on disk
Read-and write-operations on these files translated into kernel operations, together with data transfer between user space and kernel
Useful mechanism for information exchange between kernel and user space

A tour of the Linux kernel

Major parts of the kernel:

- Device drivers: in the subdirectory `drivers`, sorted according to category
- file systems: in the subdirectory `fs`
- scheduling and process management: in the subdirectory `kernel`
- memory management: in the subdirectory `mm`
- networking code: in the subdirectory `net`
- architecture specific low-level code (including assembly code): in the subdirectory `arch`
- include-files: in the subdirectory `include`

Linux Device Drivers

Device drivers

View from user space:
Have special file in `/dev` associated with it, together with five systems calls:

- `open:` make device available
- `read:` read from device
- `write:` write to device
- `ioctl:` Perform operations on device (optional)
- `close:` make device unavailable

Kernel side

Each file may have functions associated with it which are called when corresponding system calls are made
`linux/fs.h` lists all available operations on files
Device driver implements at least functions for `open`, `read`, `write` and `close`.

Categorising devices

Kernel also keeps track of

- Physical dependencies between devices. Example: devices connected to a USB-hub
- Buses: Channels between processor and one or more devices. Can be either physical (eg pci, usb), or logical
- Classes: Sets of devices of the same type, eg keyboards, mice

Handling Interrupts in Device Drivers

Normal cycle of interrupt handling for devices:

- Device sends interrupt
- CPU selects appropriate interrupt handler
- Interrupt handler processes interrupt
  Two important tasks to be done:
  - Data to be transferred to/from device
  - Waking up processes which wait for data transfer to be finished
- Interrupt handler clears interrupt bit of device
  Necessary for next interrupt to arrive

Interrupt processing time must be as short as possible
Data transfer fast, rest of processing slow
⇒ Separate interrupt processing in two halves:

- Top Half is called directly by interrupt handler
  Only transfers data between device and appropriate kernel buffer and schedules software interrupt to start Bottom half
- Bottom half still runs in interrupt context and does the rest of the processing (eg working through the protocol stack, and waking up processes)

sysfs file system

Kernel maintains virtual file system called sysfs mounted at `/sys`
interface between user space and kernel, more advanced than `/proc`
Object with properties and actions represented as directory and files used for reading writing respectively
device model represented in sysfs file system

Subdirectories of `/sys`:

- bus: hardware bus
- class device classes
- devices: physical devices
- fs: file systems
- module: module information
- kernel: linux kernel

Automatic device detection

- /dev-directory contains entries for all available devices
- managed by special daemon (udev) in connection with the kernel
- /dev mounted on temporary file system, maintained by kernel
- relies on probing devices at boot time and notifications about inserting and removing devices
- udevd started at boot time, listens to uevents from driver core
- depmod compiles a list of devices each device driver can handle

Sequence of events when new device inserted:

- The driver core (usb, pci ...) reads device id and other attributes
- Kernel sends event to `udevd`, setting the `MODALIAS` environment variable
- udev handles event and loads module via `modprobe $MODALIAS`.
- `udev` can take further action (eg setting permissions) via rules in `/etc/udev/rules.d`

The Critical-Section Problem

Background

- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes

Classic Problem: Finite buffer shared by producer and consumer

- Producer produces a stream of items and puts them into buffer
- Consumer takes out stream of items

Have to deal with producers waiting for consumers when buffer is full, and consumers waiting for producers when buffer is empty.

Producer Process Code

```
// Produce items until the cows come home.
while (TRUE) {
  // Produce an item.
  Item next_produced = [some new item to add to buffer];

  // Wait one place behind the next item to be consumed - so we don't
  // write to items that have yet to be consumed.
  while (((in + 1) % BUFFER_SIZE) == out); // <- Spin on condition

  // Store the new item and increment the 'in' index.
  buffer[in] = next_produced;
  in = (in + 1) % BUFFER_SIZE;
}
```

Consumer Process Code

```
while (true) {
  // Wait until there is something to consume.
  while (in == out); // <- Spin on condition
  // Get the next item from the buffer
  Item next_item = buffer[out];
  [process next_item]

  // Increment the out index.
  out = (out + 1) % BUFFER_SIZE;
}
```

Improving the Bounded Buffer

- Suppose that we wanted to provide a solution to the consumer-producer problem that fills all the buffer slots, rather than `BUFFER_SIZE-1`.
  - We can do so by having an integer `count` that keeps track of the number of full buffers. Initially, `count` is set to `0`. It is incremented by the producer after it produces a new item and is decremented by the consumer after it consumes an item.

Producer

```
while (true) {
  /* produce an item and put in nextProduced */
  while (count == BUFFER_SIZE); // wait if buffer full
  buffer [in] = nextProduced; // store new item
  in = (in + 1) % BUFFER_SIZE; // increment IN pointer.
  count++; // Increment counter
}
```

Consumer

```
while (true) {
  while (count == 0) ; // do nothing
  nextConsumed = buffer[out];
  out = (out + 1) % BUFFER_SIZE;
  count--;
  /* consume the item in nextConsumed */
}
```

Race Condition

There is something wrong with this code!

- `count++` could be compiled as a sequence of CPU instructions:
  - `register1 = count`
  - `register1 = register1 + 1`
  - `count = register1`
- `count--` could be compiled likewise:
  - `register2 = count`
  - `register2 = register2 - 1`
  - `count = register2`

Race Condition

- Consider that the producer and consumer execute the `count++` and `count--` around the same time, such that the CPU instructions are interleved as follows (with `count = 5` initially):
  - Prod.: `register1 = count` $\{$`register1` $\rightarrow 5\}$
  - Prod.: `register1 = register1 + 1` $\{$`register1` $\rightarrow 6\}$
  - Cons.: `register2 = count` $\{$`register2` $\rightarrow 5\}$
  - Cons.: `register2 = register2 - 1` $\{$`register2` $\rightarrow 4\}$
  - Prod.: `count = register1` $\{$`count` $\rightarrow 6\}$
  - Cons.: `count = register2` $\{$`count` $\rightarrow 4\}$
- With an increment and a decrement, in whatever order, we would expect no change to the original value (5), but here we have ended up with 4?!?!.

How to implement synchronization primitives?

Solution Criteria to Critical-Section Problem

- Solution to protect against concurrent modification of data in the *critical section* with the following criteria:
  - **Mutual Exclusion** - If process P_i is in its critical section (*i.e.* where shared variables could be altered inconsistently), then no other processes can be in this critical section.
  - **Progress** - no process outside of the critical section (*i.e.* in the *remainder section*) should block a process waiting to enter.
  - **Bounded Waiting** - A bound must exist on the number of times that other processes are allowed to enter the critical section after a process has made a request to enter its critical section and before that request is granted (*i.e.* it must be fair, so one poor process is not always waiting behind the others).
- Assume that each process executes at a nonzero speed.
- No assumption concerning relative speed of the N processes.

Peterson's Solution

- Two process solution.
- Assume that the CPU's register LOAD and STORE instructions are atomic (*not realistic* on modern CPUs, educational example).
- The two processes share two variables:
  - `int turn;`
  - `Boolean wants_in[2];`
- The variable `turn` indicates whose turn it is to enter the critical section.
- The `wants_in` array is used to indicate if a process is ready to enter the critical section. `wants_in[i] = true` implies that process P_i is ready.

A Naïve Algorithm for Process P_i

```
is_in[i] = FALSE; // I'm not in the section
do {
  while (is_in[j]); // Wait whilst j is in the critical section
  is_in[i] = TRUE; // I'm in the critical section now.
    [critical section]
  is_in[i] = FALSE; // I've finished in the critical section now.
    [remainder section]
} while (TRUE);
```

What's wrong with this?

Peterson's Algorithm for Process P_i

```
do {
  wants_in[i] = TRUE; // I want access...
  turn = j; // but, please, you go first
  while (wants_in[j] && turn == j); // if you are waiting and it is
                                    // your turn, I will wait.
    [critical section]
  wants_in[i] = FALSE; // I no longer want access.
    [remainder section]
} while (TRUE);
```

When both processes are interested, they achieve fairness through the `turn` variable, which causes their access to alternate.

Peterson's Algorithm for Process P_i

Interesting, but:

- Aside from the question of CPU instruction atomicity, how can we support more than two competing processes?
- Also, if we were being pedantic, we could argue that a process may be left to wait unnecessarily if a context switch occurs only after another process leaves the remainder section but before it politely offers the turn to our first waiting process.

Synchronisation Hardware

Synchronisation Hardware

- Many systems provide hardware support for critical section
- Uniprocessors - could disable interrupts
  - Currently running code would execute without preemption
  - Generally too inefficient on multiprocessor systems
  - Delay in one processor telling others to disable their interrupts
- Modern machines provide the special atomic hardware instructions (Atomic = non-interruptible) `TestAndSet` or `Swap` which achieve the same goal:
  - `TestAndSet`: Test memory address (*i.e.* read it) and set it in one instruction
  - `Swap`: Swap contents of two memory addresses in one instruction
- We can use these to implement simple locks to realise mutual exclusion

Solution to Critical-section Problem Using Locks

- The general pattern for using locks is:

```
do {
  [acquire lock]
    [critical section]
  [release lock]
    [remainder section]
} while (TRUE);
```

TestAndSet Instruction

- High-level definition of the atomic CPU instruction:

```
boolean TestAndSet (boolean *target) {
  boolean original = *target; // Store the original value
  *target = TRUE; // Set the variable to TRUE
  return original: // Return the original value
}
```

- In a nutshell: this single CPU instruction sets a variable to TRUE and returns the original value.
- This is useful because, if it returns FALSE, we know that only our thread has changed the value from FALSE to TRUE; if it returns TRUE, we know we haven't changed the value.

Solution using TestAndSet

- Shared boolean variable `lock`, initialized to false.
- Solution:

```
do {
  while (TestAndSet(&lock)) ; // wait until we successfully
                              // change lock from false to true
    [critical section]
  lock = FALSE; // Release lock
    [remainder section]
} while (TRUE);
```

- Note, though, that this achieves mutual exclusion but not *bounded waiting* - one process could potentially wait for ever due to the unpredictability of context switching.

Bounded-waiting Mutual Exclusion with TestAndSet()

- All data structures are initialised to FALSE.
- `wants_in[]` is an array of waiting flags, one for each process.
- `lock` is a boolean variable used to lock the critical section.

Bounded-waiting Mutual Exclusion with TestAndSet()

```
boolean wants_in[], key = FALSE, lock = FALSE; //all false to begin with

do {
  wants_in[i] = TRUE; // I (process i) am waiting to get in.
  key = TRUE; // Assume another has the key.
  while (wants_in[i] && key) { // Wait until I get the lock
                               // (key will become false)
    key = TestAndSet(&lock); // Try to get the lock
  }
  wants_in[i] = FALSE; // I am no longer waiting: I'm in now

  [*** critical section ***]

  // Next 2 lines: get ID of next waiting process, if any.
  j = (i + 1) % NO_PROCESSES; // Set j to my right-hand process.
  while ((j != i) && !wants_in[j]) { j = (j + 1) % NO_PROCESSES };

  if (j == i) { // If no other process is waiting...
    lock = FALSE; // Release the lock
  } else { // else ....
    wants_in[j] = FALSE; // Allow j to slip in through the 'back door'
  }
  [*** remainder section ***]
} while (TRUE);
```

Inefficient Spinning

But There's a Bit of a Problem...

- Consider the simple mutual exclusion mechanism we saw in the last lecture:

```
do {
  while (TestAndSet(&lock)) ; // wait until we successfully
                              // change lock from false to true
    [critical section]
  lock = FALSE; // Release lock
    [remainder section]
} while (TRUE);
```

- This guarantees mutual exclusion but with a high cost: that while loop spins constantly (using up CPU cycles) until it manages to enter the critical section.
- This is a huge waste of CPU cycles and is not acceptable, particularly for user processes where the critical section may be occupied for some time.

Sleep and Wakeup

- Rather than having a process spin around and around, checking if it can proceed into the critical section, suppose we implement some mechanism whereby it sends itself to sleep and then is awoken only when there is a chance it can proceed
- Functions such as `sleep()` and `wakeup()` are often available via a threading library or as kernel service calls.
- Let's explore this idea

Mutual Exclusion with sleep(): A first Attempt

- If constant spinning is a problem, suppose a process puts itself to sleep in the body of the spin.
- Then whoever won the competition (and therefore entered the critical section) will wake all processes before releasing the lock - so they can all try again.

```
do {
  while (TestAndSet(&lock)) { // If we can't get the lock, sleep.
    sleep();
  }
    [critical section]
  wake_up(all); // Wake all sleeping threads.
  lock = FALSE; // Release lock
    [remainder section]
} while (TRUE);
```

Mutual Exclusion with sleep(): A first Attempt

- Is this a satisfactory solution?
- Does it achieve mutual exclusion?
- Does it allow progress (*i.e.* stop non-interested processes from blocking those that want to get it)?
- Does it have the property of bounded waiting?

Towards Solving The Missing Wakeup Problem

- Somehow, we need to make sure that when a process decides that it will go to sleep (if it failed to get the lock) it actually goes to sleep without interruption, so the wake-up signal is not missed by the not-yet-sleeping process
- In other words, we need to make the check-if-need-to-sleep and go-to-sleep operations happen atomically with respect to other threads in the process.
- Perhaps we could do this by making use of another lock variable, say `deciding_to_sleep`.
- Since we now have two locks, for clarity, let's rename `lock` to `mutex_lock`.

A Possible Solution?

```
while (True) {
  // Spinning entry loop.
  while (True) {
    // Get this lock, so a wake signal cannot be raised before we actually sleep.
    while(TestAndSet(&deciding_to_sleep));

    // Now decide whether or not to sleep on the lock.
    if (TestAndSet(&mutex_lock)) {
      sleep();
    } else {
      // We are going in to critical section.
      deciding_to_sleep = False;
      break;
    }
    deciding_to_sleep = False; // Release the sleep mutex for next attempt.
  }
    [critical section]
  while(TestAndSet(&deciding_to_sleep)); // Don't issue 'wake' if a thread is
                                         // deciding whether or not to sleep.
  mutex_lock = False; // Open up the lock for the next guy.
  wake_up(all); // Wake all sleeping threads so they may compete for entry.
  deciding_to_sleep = False;
    [remainder section]
}
```

- Have we, perhaps, overlooked something here?

Sleeping with the Lock

- We've encountered an ugly problem — in fact, there are several problems with the previous example that are all related.
  - As we said before, we need to decide to sleep and then sleep in an atomic action (with respect to other threads/processes)
  - But in the previous example, when a thread goes to sleep it keeps hold of the `deciding_to_sleep` lock which sooner or later will result in deadlock!
  - And if we release the `deciding_to_sleep` lock immediately before sleeping, then we have not solved the original problem.
- What to do, what to do...

Sleeping with the Lock

- The solution to this problem implemented in modern operating systems, such as Linux, is to release the lock *during* the kernel service call `sleep()`, such that it is released prior to the context switch, with a guarantee there will be no interruption.
- The lock is then reacquired upon wakeup, prior to the return from the `sleep()` call.
- Since we have seen how this can all get very complicated when we introduce the idea of sleeping (to avoid wasteful spinning), kernels often implement a sleeping lock, called a *semaphore*, which hides the gore from us.
- See `http://tomoyo.sourceforge.jp/cgi-bin/lxr/source/kernel/semaphore.c#L75`

Semaphores

Semaphores

- Synchronisation tool, proposed by E. W. Dijkstra (1965), that
  - Simplifies synchronisation for the programmer
  - Does not require (much) busy waiting
    - We don't busy-wait for the critical section, usually only to achieve atomicity to check if we need to sleep or not, *etc.*
  - Can guarantee *bounded waiting time* and *progress*.
- Consists of:
  - A semaphore type `S`, that records a list of waiting processes and an integer
  - Two standard atomic ($\leftarrow$ very important) operations by which to modify `S`: `wait()` and `signal()`
    - Originally called `P()` and `V()` based on the equivalent Dutch words

Semaphores

- Works like this:
  - The semaphore is initialised with a count value of the maximum number of processes allowed in the critical section at the same time.
  - When a process calls `wait()`, if count is zero, it adds itself to the list of sleepers and blocks, else it decrements count and enters the critical section
  - When a process exits the critical section it calls `signal()`, which increments count and issues a wakeup call to the process at the head of the list, if there is such a process
    - It is the use of ordered wake-ups (*e.g.* FIFO) that makes semaphores support bounded (*i.e.* fair) waiting.

Semaphore as General Synchronisation Tool

- We can describe a particular semaphore as:
  - A Counting semaphore - integer value can range over an unrestricted domain (*e.g.* allow at most N threads to read a database, *etc.*)
  - A Binary semaphore - integer value can range only between 0 and 1
    - Also known as mutex locks, since ensure mutual exclusion.
    - Basically, it is a counting semaphore initialised to 1

Critical Section Solution with Semaphore

```
Semaphore mutex; // Initialized to 1
do {
  wait(mutex); // Unlike the pure spin-lock, we are blocking here.
    [critical section]
  signal(mutex);
    [remainder section]
} while (TRUE);
```

Semaphore Implementation: State and Wait

We can implement a semaphore within the kernel as follows (note that the functions must be atomic, which our kernel must ensure):

```
typedef struct {
  int count;
  process_list; // Hold a list of waiting processes/threads
} Semaphore;

void wait(Semaphore *S) {
  S->count--;
  if (S->count < 0) {
    add process to S->process_list;
    sleep();
  }
}
```

Semaphore Implementation: State and Wait

- Note that, here, we decrement the wait counter before blocking (unlike the previous description).
- This does not alter functionality but has the useful side-effect that the negative count value reveals how many processes are currently blocked on the semaphore.

Semaphore Implementation: Signal

```
void signal(Semaphore *S) {
  S->count++;
  if (S->count <= 0) { // If at least one waiting process, let him in.
    remove next process, P, from S->process_list
    wakeup(P);
  }
}
```

But we have to be Careful: Deadlock and Priority Inversion

- Deadlock: Two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes
- Let `S` and `Q` be two semaphores initialized to 1

Process 1	Process 2
wait(S);	wait(Q);
wait(Q);	wait(S);
.	.
.	.
.	.
signal(S);	signal(Q);
signal(Q);	signal(S);

- Priority Inversion: Scheduling problem when lower-priority process holds a lock needed by higher-priority process
  - Good account of this from NASA programme: `http://research.microsoft.com/en-us/um/people/mbj/mars_pathfinder/authoritative_account.html`

Semaphore Examples

Classical Problems of Synchronisation and Semaphore Solutions

- Bounded-Buffer Problem
- Readers and Writers Problem
- Boring as it may be to visit (and teach about) the same problems over and over again, a solid understanding of these seemingly-insignificant problems will directly help you to spot and solve many related real-life synchronisation issues.

Our Old Friend: The Bounded-Buffer Problem

- The solution set-up:
  - A buffer to hold N items, each can hold one item
  - Semaphore `mutex` initialized to the value 1
  - Semaphore `full_slots` initialized to the value 0
  - Semaphore `empty_slots` initialized to the value N.

Producer

```
while(True) {
  wait(empty_slots); // Wait for, then claim, an empty slot.
  wait(mutex);
  // add item to buffer
  signal(mutex);
  signal(full_slots); // Signal that there is one more full slot.
}
```

Consumer

```
while(True) {
  wait(full_slots); // Wait for, then claim, a full slot
  wait(mutex);
  // consumer item from buffer
  signal(mutex);
  signal(empty_slots); // Signal that now there is one more empty slot.
}
```

Readers-Writers Problem

- A data set is shared among a number of concurrent processes
  - Readers - only read the data set; they do not perform any updates
  - Writers - can both read and write
- Problem:
  - Allow multiple readers to read at the same time if there is no writer in there.
  - Only one writer can access the shared data at the same time

Readers-Writers Problem

- The solution set-up:
  - Semaphore `mutex` initialized to 1
  - Semaphore `wrt` initialized to 1
  - Integer `readcount` initialized to 0

Writer

```
while(True) {
  wait(wrt); // Wait for write lock.
  // perform writing
  signal(wrt); // Release write lock.
}
```

Reader

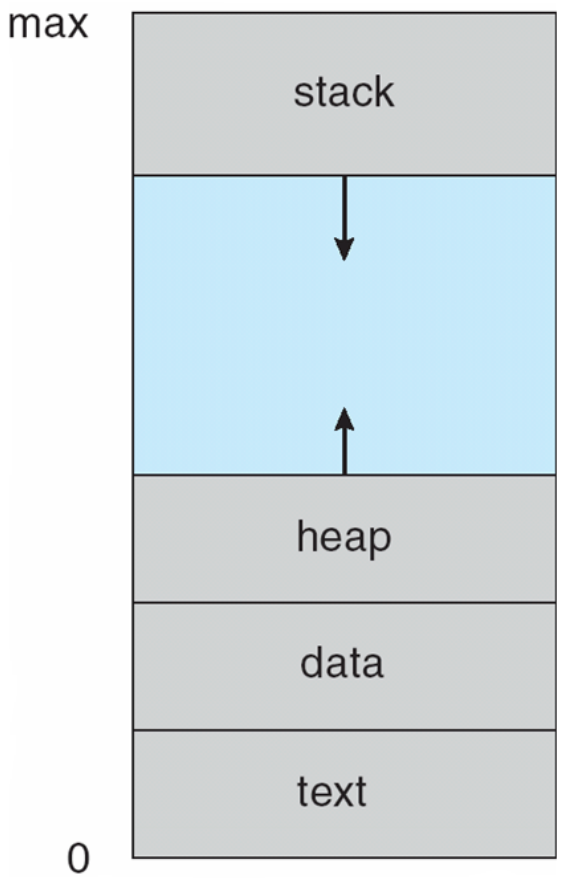
```
while(True) {
  wait(mutex) // Wait for mutex to change read_count.
  read_count++;
  if (read_count == 1) // If we are first reader, lock out writers.
    wait(wrt)
  signal(mutex) // Release mutex so other readers can enter.

  // perform reading

  wait(mutex) // Decrement read_count as we leave.
  read_count--;
  if (read_count == 0)
    signal(wrt) // If we are the last reader to
  signal(mutex) // leave, release write lock
}
```

Process Concept

Process Concept

- An operating system executes a variety of programs:
  - Batch system - jobs
  - Time-shared systems - user programs or tasks
- Process - a program in execution; process execution must progress in sequential fashion
- A process includes:
  - program counter
  - stack
  - data section

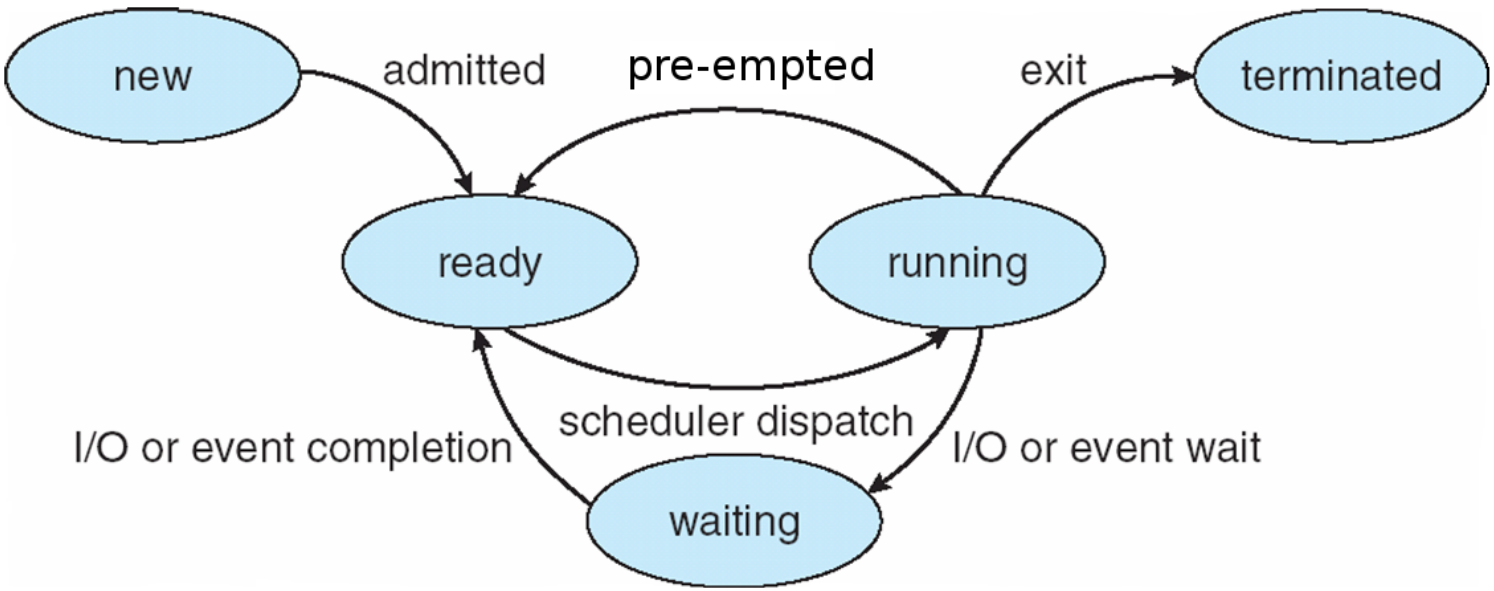
Process in Memory



Process States

- As a process executes, it changes state
  - new: The process is being created
  - running: Instructions are being executed
  - waiting: The process is waiting for some event to occur
  - ready: The process is waiting to be assigned to a processor
  - terminated: The process has finished execution

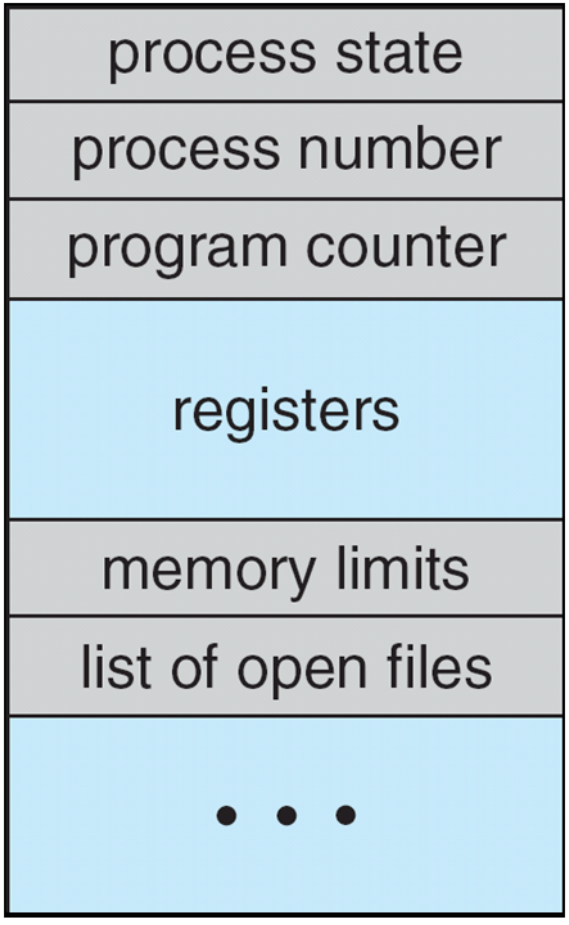
Process States



Process Control Block

- Information associated with each process, which is stored as various fields within a kernel data structure:
  - Process state
  - Program counter
  - CPU registers
  - CPU scheduling information
  - Memory-management information
  - Accounting information
  - I/O status information

Process Control Block



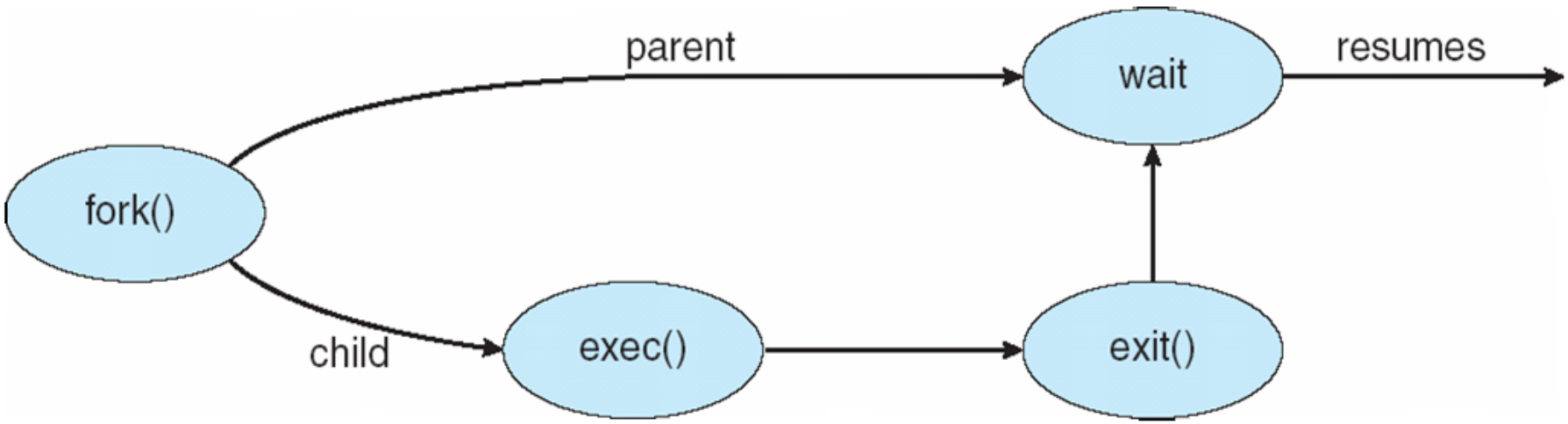
Process Creation

- Parent process create children processes, which, in turn create other processes, forming a tree of processes
- Generally, process identified and managed via a process identifier (pid)
- Resource sharing
  - Parent and children share all resources
  - Children share subset of parent's resources
  - Parent and child share no resources
- Execution
  - Parent and children execute concurrently
  - Parent waits until children terminate

Process Creation

- Address space
  - Child duplicate of parent
  - Child has a program loaded into it
- UNIX examples
  - `fork` system call creates new process
    - will look at fork soon, in one of our practical lectures.
  - exec system call used after a fork to replace the process' memory space with a new program

Process Creation



Process Termination

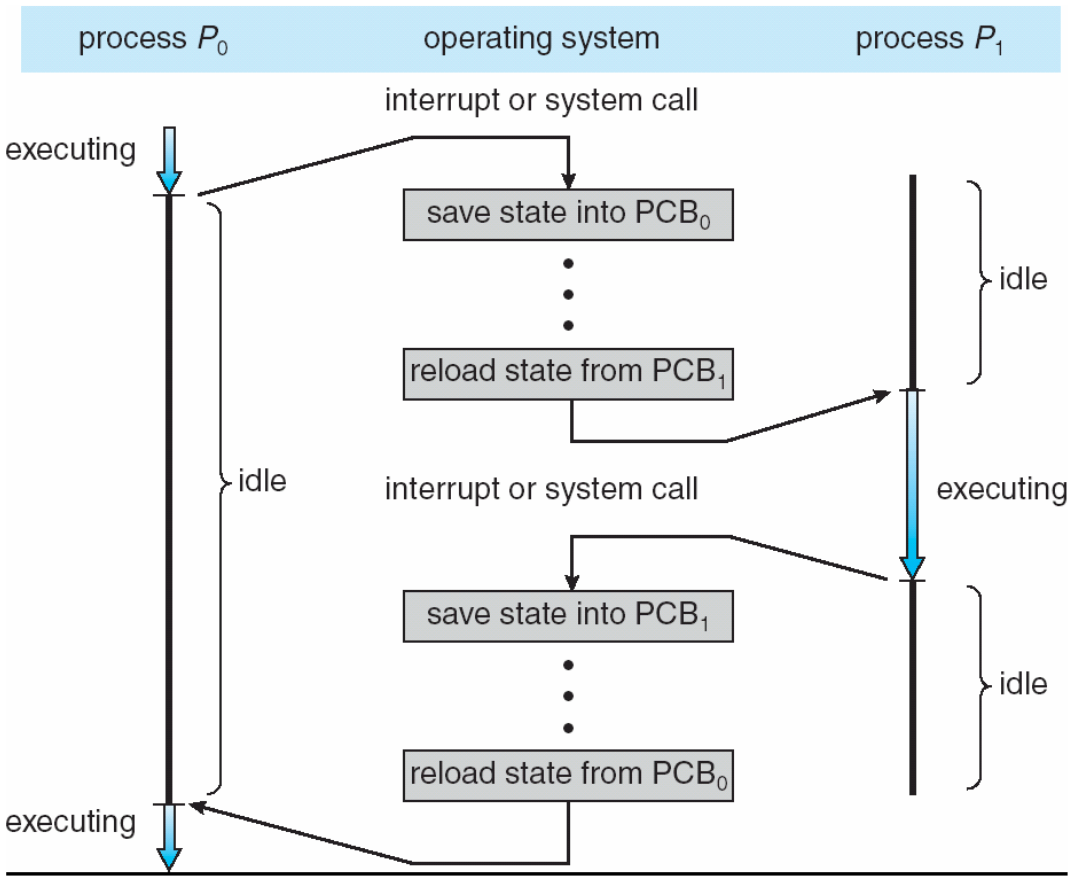
- Process executes last statement and asks the operating system to delete it (exit)
  - Output data from child to parent (via wait)
  - Process' resources are deallocated by operating system
- Parent may terminate execution of children processes (abort)
  - Child has exceeded allocated resources
  - Task assigned to child is no longer required
  - If parent is exiting:
    - Some operating systems do not allow child to continue if its parent terminates — all children terminated (*i.e.* cascading termination)

Concurrency Through Context Switching

Context Switch

- When CPU switches to another process, the system must save the state of the old process and load the saved state for the new process via a context switch
- Context of a process represented in the PCB
- Context-switch time is overhead; the system does no useful work while switching
- Time dependent on hardware support

Context Switch

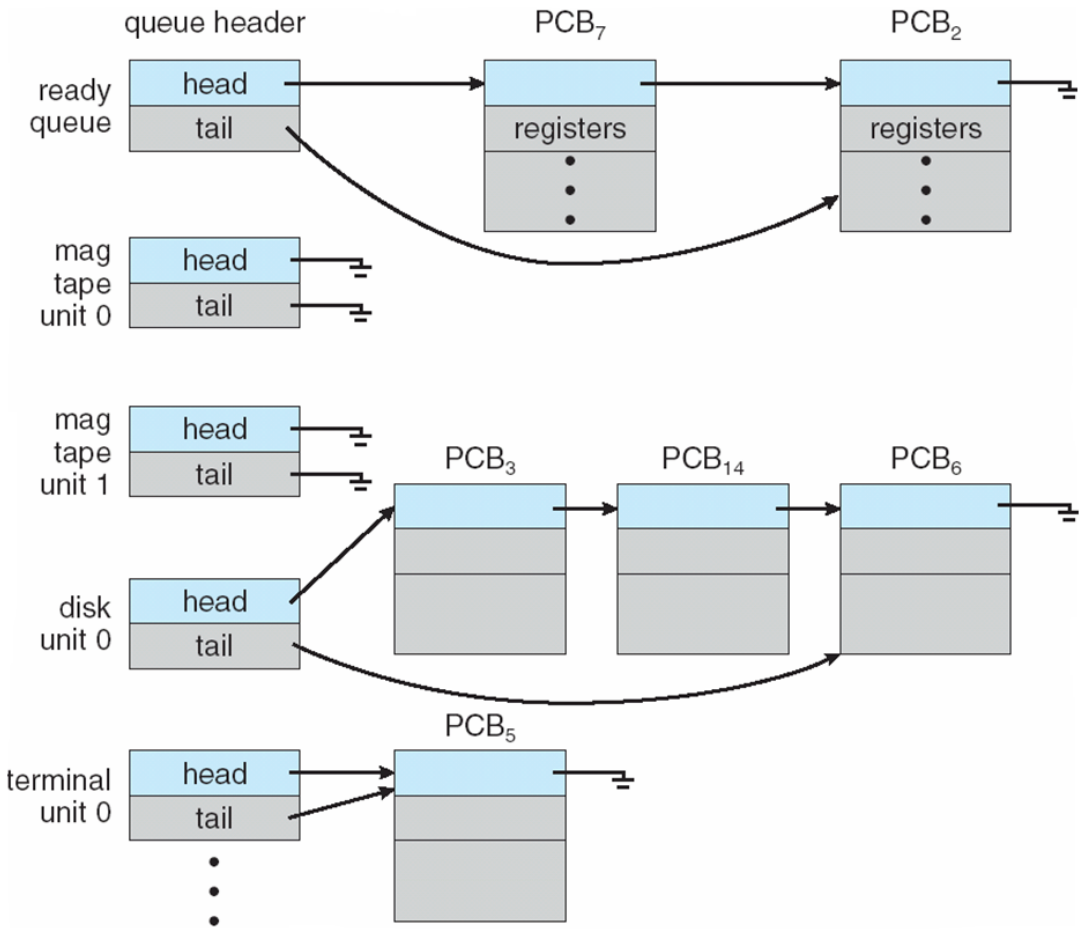


Overview of Process Scheduling

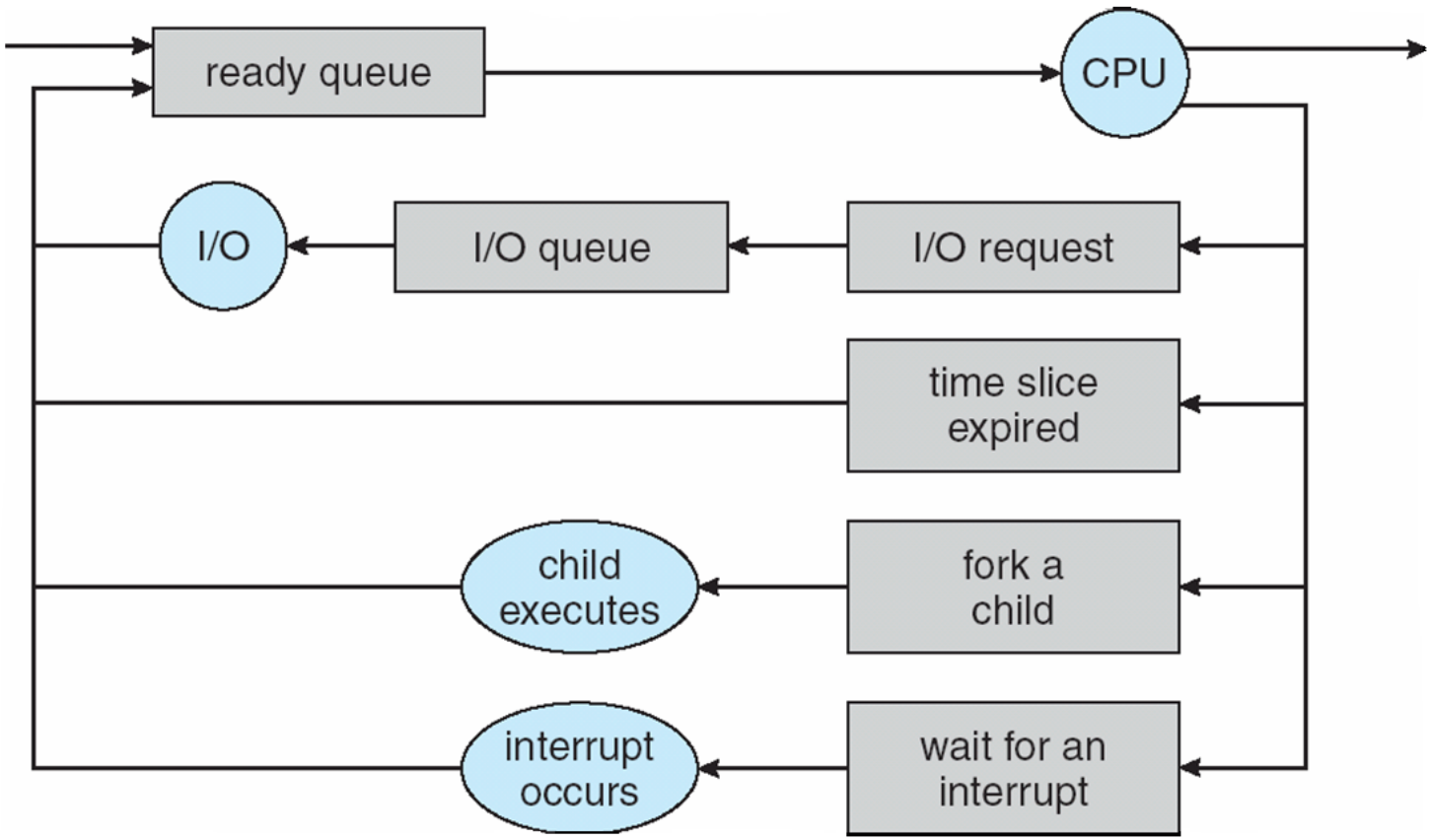
Process Scheduling Queues

- Job queue - set of all processes in the system
- Ready queue - set of all processes residing in main memory, ready and waiting to execute
- Device queues - set of processes waiting for an I/O device
- Processes migrate among the various queues

Process Scheduling Queues



Scheduling Workflow



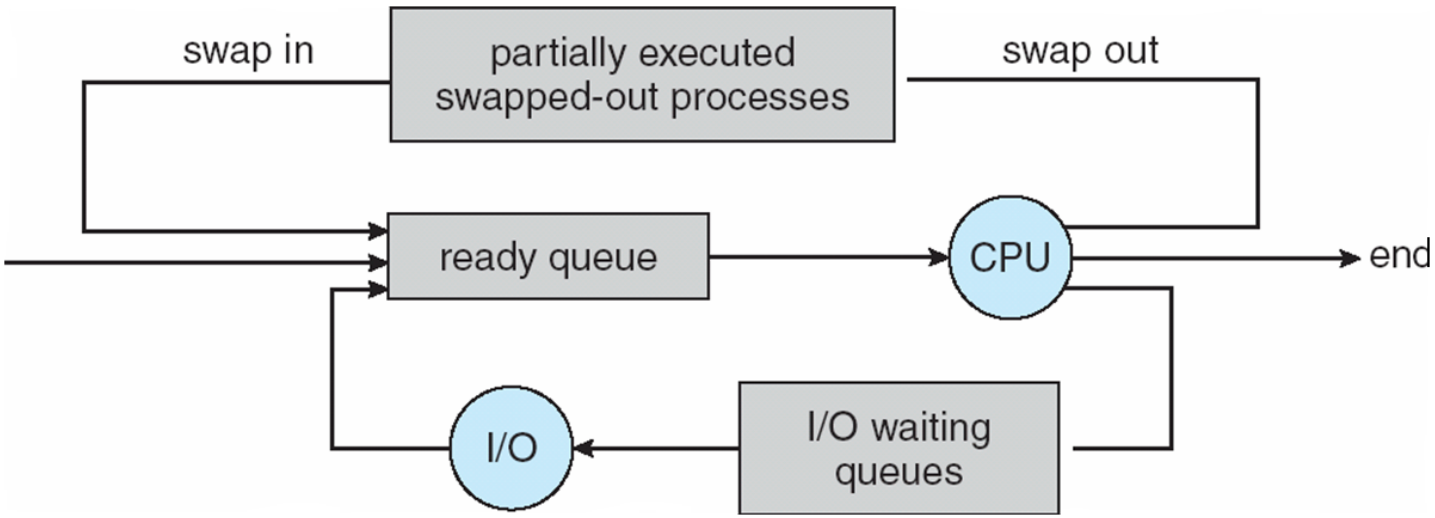
Schedulers

- Long-term scheduler (or job scheduler) - selects which processes should be brought into the ready queue (*e.g.* loaded from the disk into memory)
- Short-term scheduler (or CPU scheduler) - selects which process should be executed next and allocates CPU

Schedulers

- Short-term scheduler is invoked very frequently (milliseconds) (must be fast)
- Long-term scheduler is invoked very infrequently (seconds, minutes) (may be slow)
- The long-term scheduler controls the degree of multiprogramming (*i.e.* how many processes may compete for the CPU)
    - Long-term scheduling is often minimal or absent on mainstream operating systems, such as Windows and Linux.
- Processes can be described as either:
    - I/O-bound process - spends more time doing I/O than computations, many short CPU bursts
    - CPU-bound process - spends more time doing computations; few very long CPU bursts

Addition of Medium Term Scheduling



- To reduce contention among ready processes, in some scheduling designs, medium-term scheduling allows some processes to be temporarily swapped out of memory
  - kind of like they are being told to sit on a bench until things quieten down.

Scheduling

Scheduling

Problem: Which process ready to execute commands gets the CPU?
key function of the operating system

Prerequisites for successful scheduling:

1.) CPU-I/O-Burst Cycle

Experience shows: I/O occurs after fixed amount of time in $\geq 90\%$
$\Rightarrow$ appropriate time for re-scheduling

2.) Preemptive Scheduling: Processes can be forced to relinquish processor

Scheduling Criteria

Have various, often conflicting criteria to measure success of scheduling:

- CPU utilisation
- Throughput: Number of processes completed within a given time
- Turnaround time: Time it takes for each process to be executed
- Waiting time: Amount of time spent in the ready-queue
- Response time: time between submission of request and production of first response

Scheduling algorithms

1.) First-Come, First-Served (FCFS)

Jobs are put in a queue, and served according to arrival time

Easy to implement but CPU-intensive processes can cause long waiting time.

FCFS with preemption is called Round-Robin
standard method in time sharing systems

Problem: get the time quantum (time before preemption) right.
- too short: too many context switches
- too long: Process can monopolise CPU

Shortest Job First

Next job is one with shortest CPU-burst time (shortest CPU-time before next I/O-operation)

Not implementable, but this is algorithm with the smallest average waiting time
$\Rightarrow$ Strategy against which to measure other ones
Approximation: Can we predict the burst-time?
Only hope is extrapolation from previous behaviour
done by weighting recent times more than older ones.

$$\tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n$$

Priority Scheduling

Assumption: A priority is associated with each process
CPU is allocated to process with highest priority
Equal-priority processes scheduled according to FCFS

Two variations:

- With preemption: newly-arrived process with higher priority may gain processor immediately if process with lower priority is running
- Without preemption: newly arrived process always waits

Preemption good for ensuring quick response time for high-priority processes
Disadvantage: Starvation of low-priority processes possible
Solution: Increase priority of processes after a while (Ageing)

Multilevel Queue Scheduling

Applicable when processes can be partitioned into groups (eg interactive and batch processes):

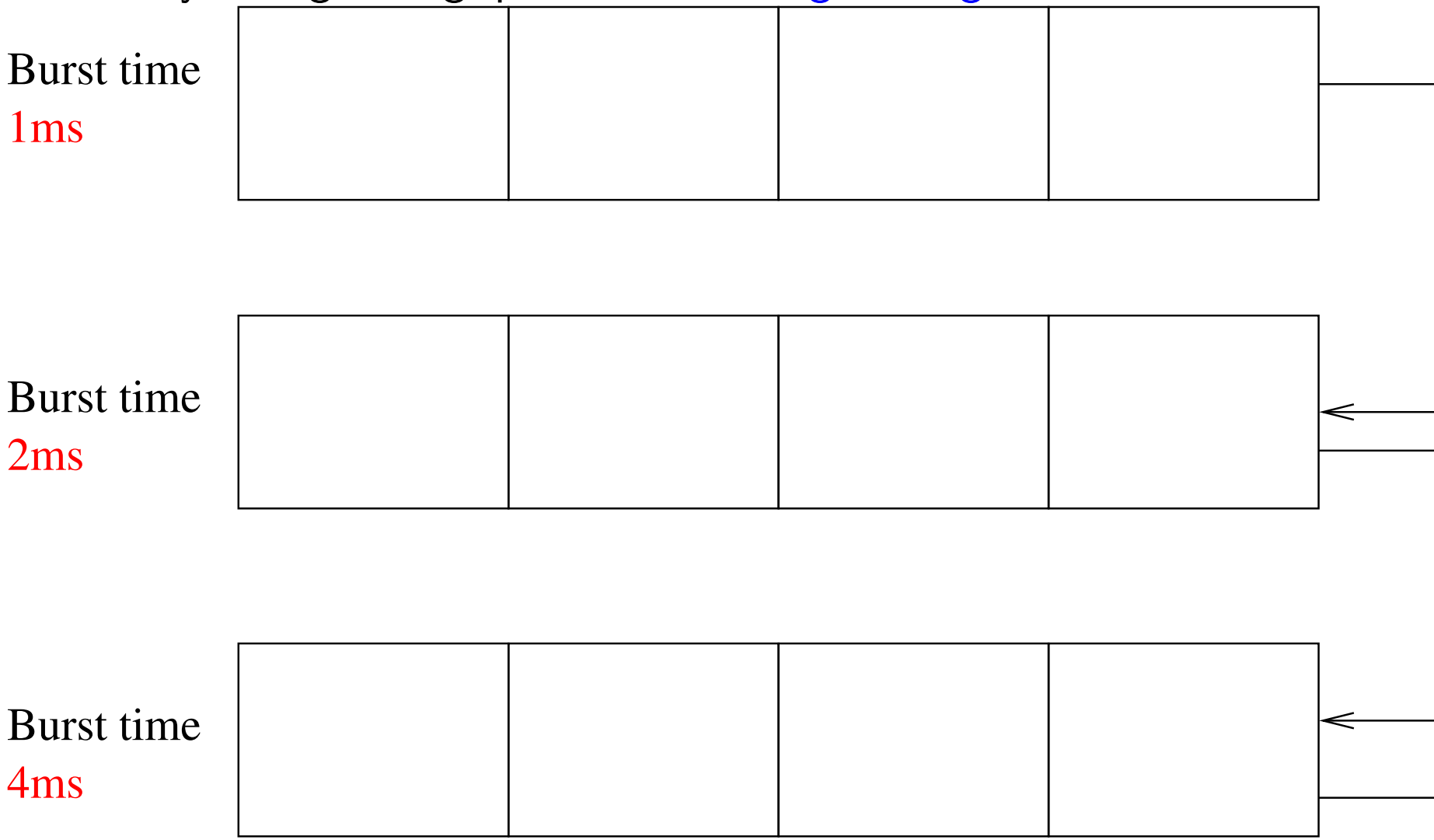
Split ready-queue into several separate queues, with separate scheduling algorithm

Scheduling between queues usually implemented as pre-emptive priority scheduling

Possible setup of queues:

- System processes
- Interactive processes
- Interactive editing processes
- Batch processes

Other way of organising queues: according to length of CPU-burst



Scheduling for Multiprocessor Systems

CPU scheduling more complex when multiple CPU's are available

Most common case: Symmetric multiprocessing (SMP):

- all processors are identical, can be scheduled independently
- have separate ready-queue for each processor (Linux), or shared ready-queue

Processor Affinity

Process affinity for CPU on which it is currently running

- Soft Affinity current CPU only preferred when re-scheduled
- Hard Affinity Process may be bound to specific CPU
  Advantage: caches remain valid, avoiding time-consuming cache invalidation and recovery

Load Balancing

Idea: use all CPU's equally (goes against processor affinity)

- Push migration: periodically check load, and push processes to less loaded CPU's
- Pull migration: idle CPU's pull processes from busy CPU's

Linux Implementation

Several schedulers may co-exist, assign fixed percentage of CPU-time to each scheduler

Important schedulers:

- Round-robin scheduler with priorities (the default scheduler)
- Real-time scheduler (process needs to be assigned explicitly to this one) (typically FIFO)

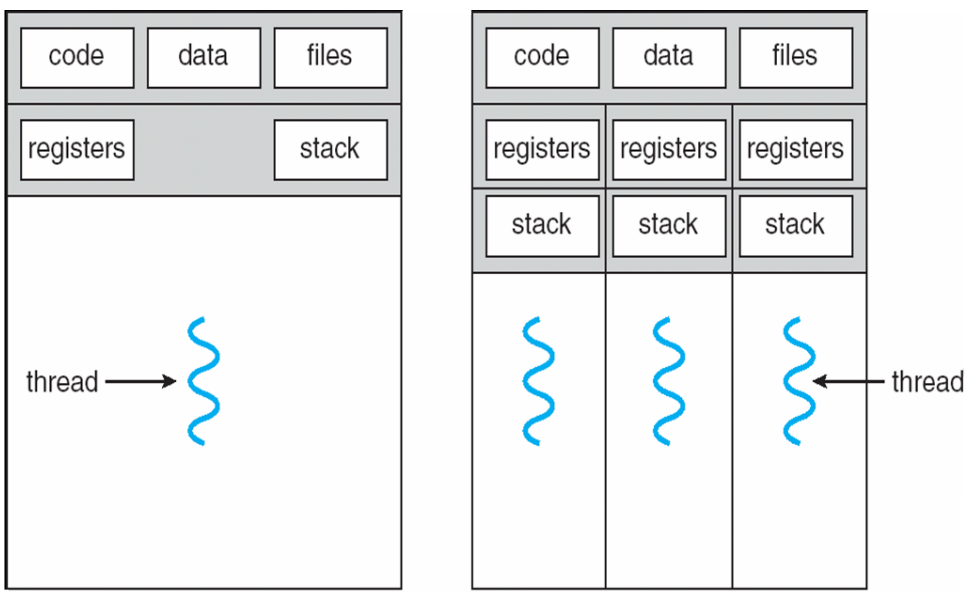
Round-Robin Scheduler with priorities

implemented in an interesting way:
maintain tree of processed ordered by runtime allocated so far
pick next process as one with least runtime allocated so far
insert new process in ready queue at appropriate place in tree
Priorities handled by giving weights to run-times.

Multithreaded Processes

Single and Multithreaded Processes

- A thread is an execution state of a process (*e.g.* the next instruction to execute, the values of CPU registers, the stack to hold local variables, *etc.*)
  - Thread state is separate from global process state, such as the code, open files, global variables (on the heap), *etc.*

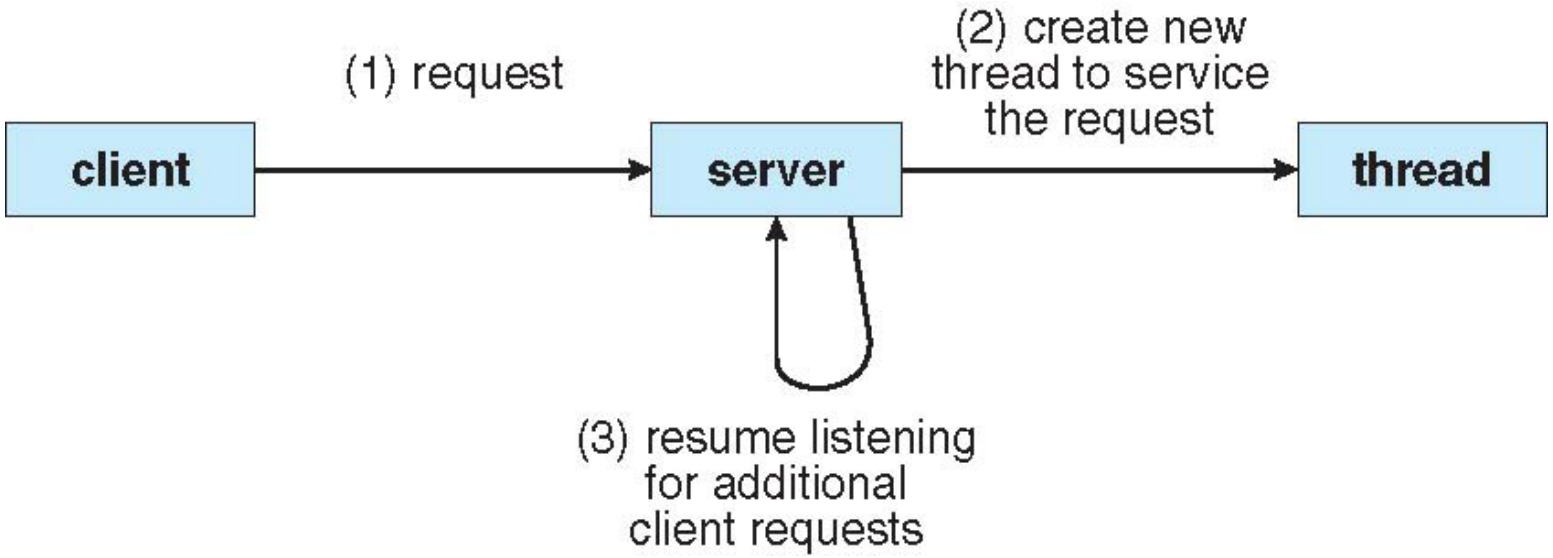


single-threaded process multithreaded process

Benefits of Threads

- **Responsiveness** - user interaction in a GUI can be responded to by a separate thread from that, say, doing long running computation (*e.g.* saving a file, running some algorithm, *etc.*)
- **Resource Sharing** - Threads within a certain process share its address space and can therefore use shared variables to communicate, which is more efficient than passing messages.
- **Economy** - Threads are often referred to as light-weight processes, since running a system with multiple threads has a smaller memory footprint than the equivalent with multiple processes.
- **Scalability** - For multi-threaded processes it is much easier to make use of parallel processing (*e.g.* multi-core processors, and distributed systems)
- **Reduce programming complexity** - Since problems can be broken down into parallel tasks, rather than more complex state machines.

Multithreaded Server Architecture

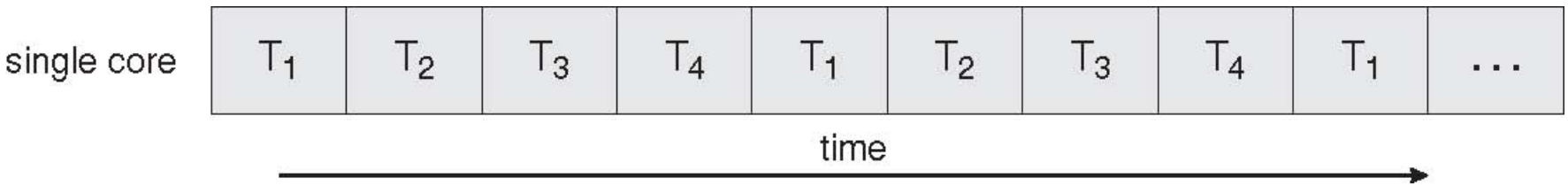


Multicore Programming

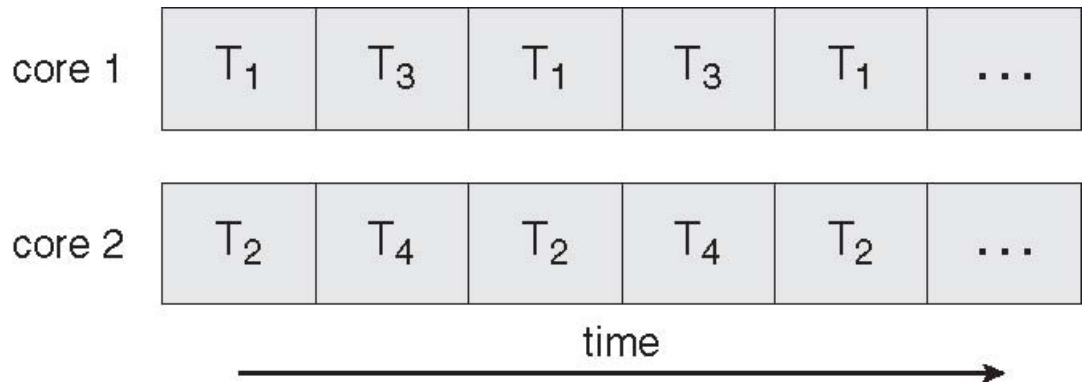
- Multicore systems are putting pressure on programmers, with challenges that include:
  - Dividing activities - How can we make better use of parallel processing?
  - Balance - How should we balance the parallel tasks on the available cores to get maximum efficiency?
  - Data splitting - How can data sets be split for processing in parallel and then rejoined (*e.g.* SETI@home)
  - Data dependency - Some processing must be performed in a certain order, so synchronisation of tasks will be necessary.
  - How to test and debug such systems?

Concurrent and Parallel Execution

Single-core Concurrent Thread Execution



Multicore Parallel Thread Execution



User Threads

- Thread management done by user-level threads library
- Three primary thread libraries:
  - POSIX Pthreads
  - Win32 threads
  - Java threads

Kernel Threads

- Threading is supported by modern OS Kernels
- Examples:
  - Windows XP/2000
  - Solaris
  - Linux
  - Mac OS X

Threading Models

Threading Models

- A particular kernel (*e.g.* on an embedded device, or an older operating system) may not support multi-threaded processes, though it is still possible to implement threading in the user process.
- Therefore many threading models exist for mapping user threads to kernel threads:
  - Many-to-One
  - One-to-One
  - Many-to-Many

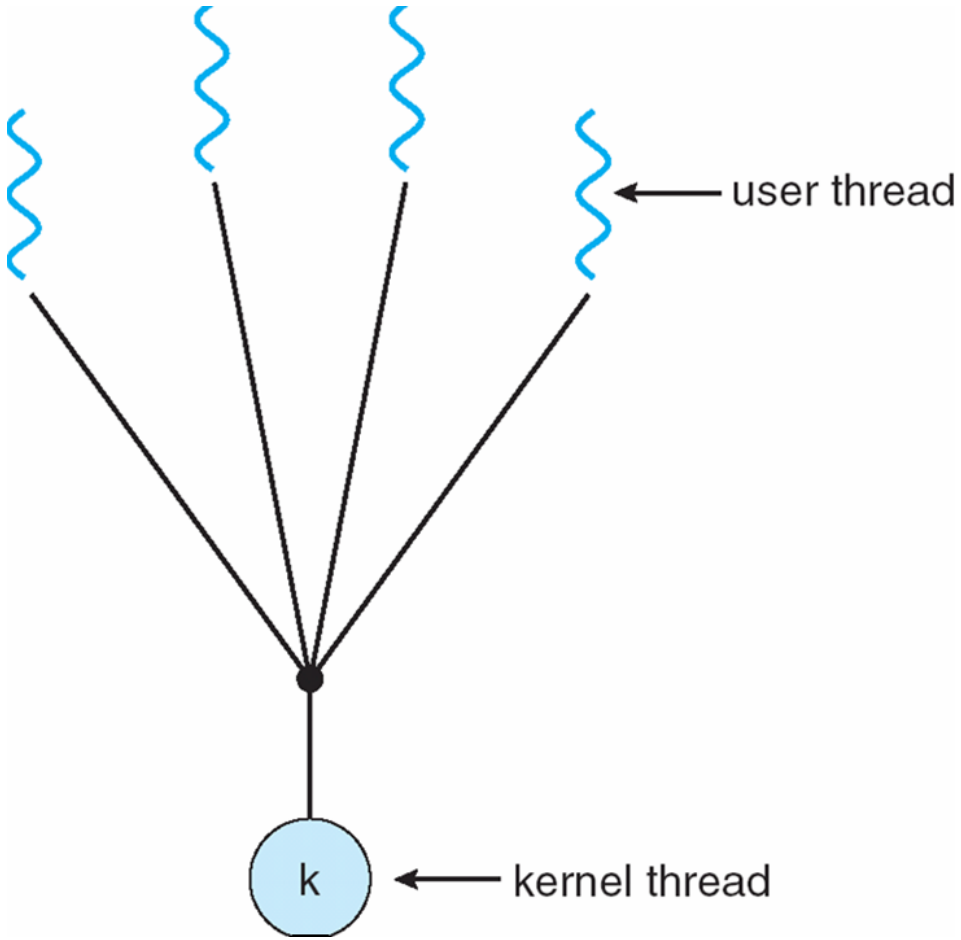
Many-to-One

- Many user-level threads mapped to single kernel thread/process
- Useful if the kernel does not support threads
- But what if one user thread calls a blocking kernel function?
  - This will block the whole process (*i.e.* all the other user threads)
  - Complex solutions exist where the user-mode thread package intercepts blocking calls, changes them to non-blocking and then implements a user-mode blocking mechanism.

Many-to-One

- You could implement something like this yourself, by having a process respond to timer events that cause it to perform a context switch in user space (*e.g.* store current registers, CPU flags, instruction pointer, then load previously stored ones)
  - Since most high-level languages cannot manipulate registers directly, you would have to write a small amount of assembler code to make the switch.
- Examples:
  - Solaris Green Threads: http://bit.ly/qYnKAQ
  - GNU Portable Threads: http://www.gnu.org/software/pth/

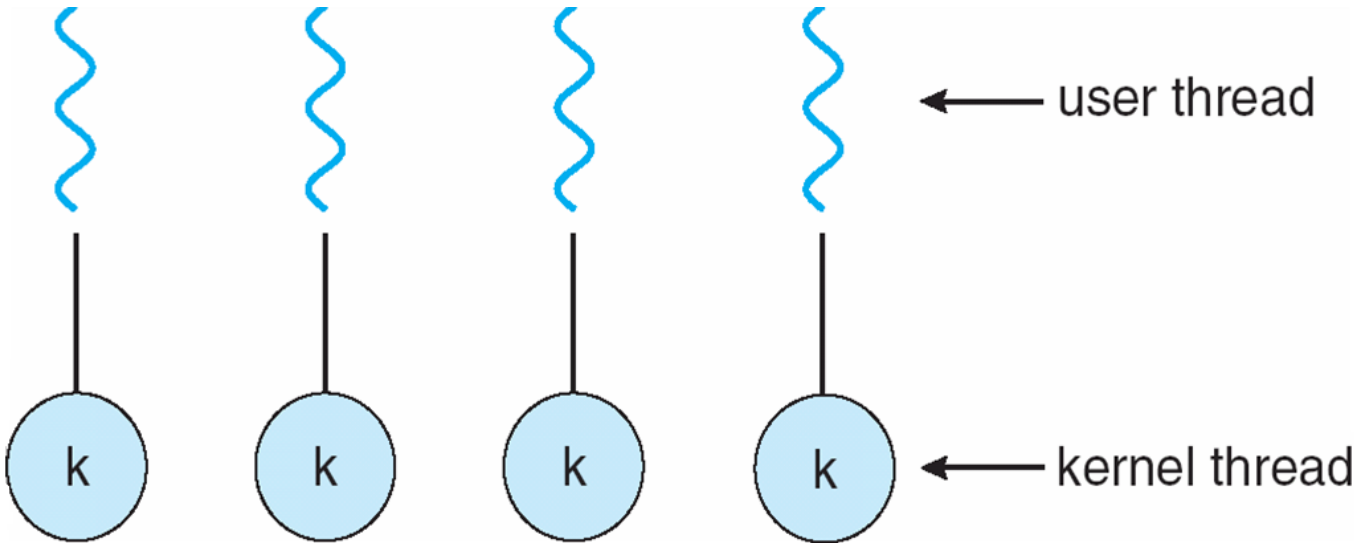
Many-to-One



One-to-One

- Each user-level thread maps to kernel thread
- But, to switch between threads a context switch is required by the kernel.
- Also, the number of kernel threads may be limited by the OS
- Examples:
  - Windows NT/XP/2000
  - Linux
  - Solaris 9 and later

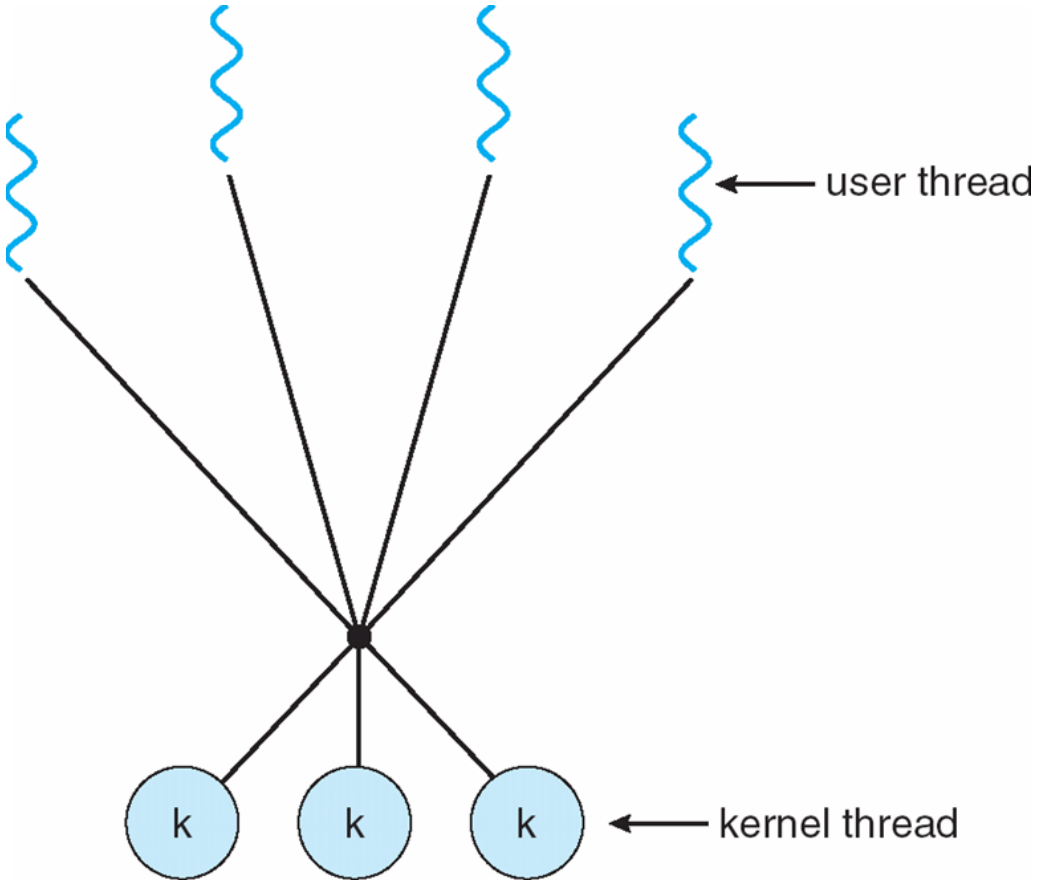
One-to-One



Many-to-Many

- Allows many user level threads to be mapped to many kernel threads
  - Best of both worlds
- Allows the operating system to create a sufficient number of kernel threads
- Solaris prior to version 9
- Windows NT/2000 with the ThreadFiber package

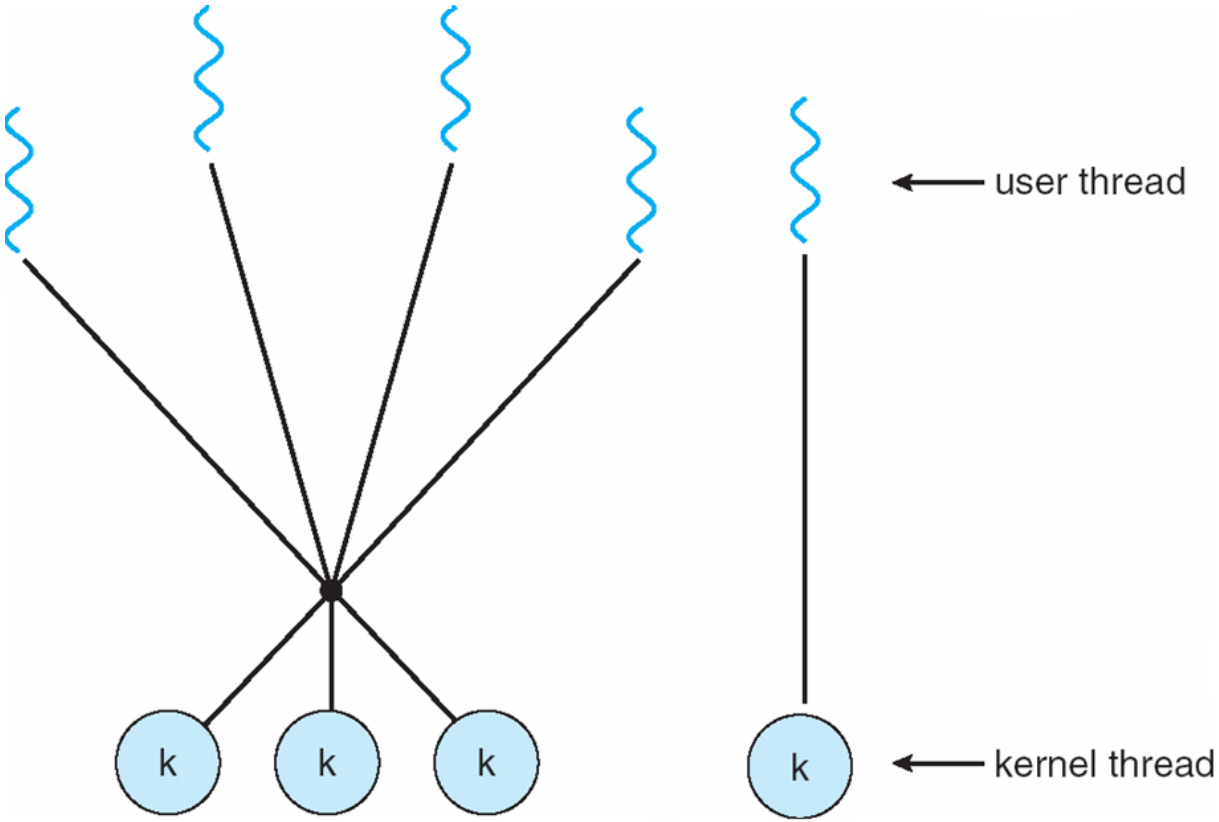
Many-to-Many



Two-Level Model

- Similar to many-to-many, except that it allows a user thread optionally to be bound directly to a kernel thread
- Examples:
  - IRIX
  - HP-UX
  - Tru64 UNIX
  - Solaris 8 and earlier

Two-Level Model



Some Considerations of Threads

Unclear Semantics of UNIX `fork()` system call

- Does `fork()` duplicate only the calling thread or all threads?
- Sometimes we want this, and sometimes we don't, so some UNIX systems provide alternative fork functions.

Thread Cancellation

- How to terminate a thread before it has finished?
- Two general approaches use by programmers:
  - **Asynchronous cancellation** terminates the target thread immediately
    - Useful as a last resort if a thread will not stop (*e.g.* due to a bug, *etc.*)
  - **Deferred cancellation** allows the target thread to periodically check if it should be cancelled
    - This approach is often considered to be much cleaner, since the thread can perform any clean-up processing (*e.g.* close files, update some state, *etc.*)

Signal Handling

- Signals are used in UNIX systems to notify a process that a particular event has occurred
- A signal handler is used to process signals
  - Signal is generated by particular event
  - Signal is delivered to a process
  - Signal is handled by some function
- Not so straightforward for a multi-threaded process. Options are:
  - Deliver the signal to the thread to which the signal applies
  - Deliver the signal to every thread in the process
  - Deliver the signal to certain threads in the process
  - Assign a specific thread to receive all signals for the process
- In most UNIX systems a thread can be configured to receive or block (*i.e.* not handle) certain signals to help overcome these issues.

Thread Pools

- Under a high request-load, multithreaded servers can waste a lot processing time simply creating and destroying threads.
- Solution:
  - Pre-create a number of threads in a pool, where they await work
- Advantages:
  - Usually slightly faster to service a request with an existing thread than create a new thread
  - Allows the number of threads in the application(s) to be bound to the size of the pool, to ensure some level of service for a finite number of clients.

Threading Implementation

Thread Libraries

- Thread library provides programmer with API for creating and managing threads
- Two primary ways of implementing
  - Library entirely in user space
  - Kernel-level library supported by the OS

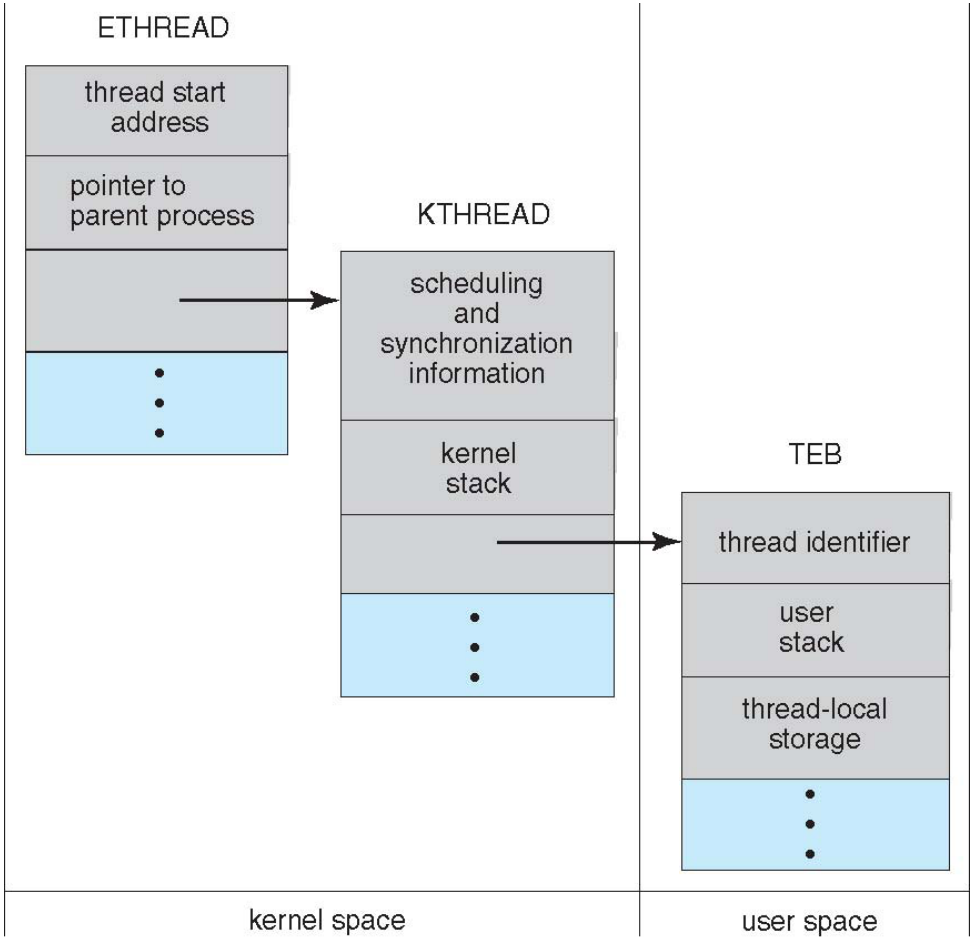
Pthreads

- May be provided either as user-level or kernel-level
- A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization
- API specifies behaviour of the thread library, implementation is up to development of the library
- Common in UNIX operating systems (Solaris, Linux, Mac OS X)
- Example: `threadtest.c`. Note, this is an implementation of *POSIX* Pthreads, so compiles differently!

Windows XP Threads

- Implements the one-to-one mapping (*i.e.* kernel-level threads)
- Each thread contains
  - A thread id
  - Register set
  - Separate user and kernel stacks
  - Private data storage area
- The register set, stacks, and private storage area are known as the context of the threads
- The primary data structures of a thread include:
  - `ETHREAD` (executive thread block) - Stores general info about a thread: its parent process, address of the instruction where the thread starts execution.
  - `KTHREAD` (kernel thread block) - Stores kernel-level state of the thread: kernel stack, *etc.*
  - `TEB` (thread environment block) - Stores user-level state of the thread: user stack, thread-local storage.

Windows XP Threads



Linux Threads

- Linux refers to them as tasks rather than threads
- Thread creation is done through `clone()` system call
- `clone()` allows a child task to share the address space of the parent task (process) and can be passed flags to control exactly what resources are shared.

flag	meaning
CLONE_FS	File-system information is shared.
CLONE_VM	The same memory space is shared.
CLONE_SIGHAND	Signal handlers are shared.
CLONE_FILES	The set of open files is shared.

Java Threads

- Java threads are managed by the JVM
- Typically implemented using the threads model provided by underlying OS
- Java threads may be created by:
  - Extending Thread class
  - Implementing the Runnable interface

File systems

File System

Function: main secondary data storage; also permanent

Extreme speed bottleneck!

Capacity not a problem nowadays: 2 TB disks even for PC.
But backup becoming a problem.

Logical view (view of programmer):
Have a tree structure of files together with read/write operation and creation of directories

Physical view:
Just a sequence of blocks, which can be read and written

OS has to map logical view to physical view

Two possibilities:

- Linked list: Each block contains pointer to next
  $\Rightarrow$ Problem: random access costly: have to go through whole file.
- Indexed allocation: Store pointers in one location: so-called index block. (cf. page table).
  To cope with vastly differing file sizes, may introduce indirect index blocks.

Caching

Disk blocks used for storing directories or recently used files cached in main memory

Blocks periodically written to disk

$\Rightarrow$ Big effiency gain

Inconsistency arises when system crashes

Reason why computers must be shutdown properly

Journaling File Systems

To minimise data loss at system crashes, ideas from databases are used:

- Define Transaction points: Points where cache is written to disk

  $\Rightarrow$ Have consistent state
- Keep log-file for each write-operation

  Log enough information to unravel any changes done after latest transaction point

RAID-Arrays

RAID: Redundant Array of Independent Disks

Main purpose: Increase reliability

- Mirroring: Store same data on different disks
- Parity Schemes Store data on n disks. Use disk $n+1$ to contain parity blocks

  $\Rightarrow$ can recover from single disk failure

  Disadvantage: Parity bit needs to be re-computed for each write operation

Disk access

Disk access contains three parts:

- Seek: head moves to appropriate track
- Latency: correct block is under head
- Transfer: data transfer

Time necessary for Seek and Latency dwarfs transfer time
$\Rightarrow$ Distribution of data and scheduling algorithms have vital impact on performance

Disk scheduling algorithms

Standard algorithms apply, adapted to the special situation:
1.) FCFS: easiest to implement, but: may require lots of head movements

2.) Shortest Seek Time First: Select job with minimal head movement
Problems:
- may cause starvation
- Tracks in the middle of disk preferred

Algorithm does not minimise number of head movements

3.) SCAN-scheduling: Head continuously scans the disk from end to end (lift strategy)
$\Rightarrow$ solves the fairness and starvation problem of SSTF

Improvement: LOOK-scheduling:
head only moved as far as last request (lift strategy).

Particular tasks may require different disk access algorithms
Example : Swap space management
Speed absolutely crucial $\Rightarrow$ different treatment:

- Swap space stored on separate partition
- Indirect access methods not used

- Special algorithms used for access of blocks
  Optimised for speed at the cost of space (eg increased internal fragmentation)

Linux Implementation

Interoperability with Windows and Mac requires support of different file systems (eg vfat)
$\Rightarrow$ Linux implements common interface for all filesystems
Common interface called virtual file system
virtual file system maintains

- inodes for files and directories
- caches, in particular for directories
- superblocks for file systems

All system calls (eg `open`, `read`, `write` and `close`) first go to virtual file system
If necessary, virtual file system selects appropriate operation from real file system

Disk Scheduler

Kernel makes it possible to have different schedulers for different file systems
Default scheduler (Completely Fair Queuing) based on lift strategy
have in addition separate queue for disk requests for each process
queues served in Round-Robin fashion
Have in addition No-op scheduler: implements FIFO
Suitable for SSD's where access time for all sectors is equal

OS Security

Security

At least three different categories:

- *Data integrity*: Backup etc., taken care by normal OS functions
- *Protection against user errors*: done by separating users and processes
- *Protection against malicious users*: more complicated, involves trade-offs between ease of use and level of protection

Major problem: *Identification of users*

Most common scheme: Passwords

\+ : Easy to use and understand

\- : All too often too easily guessable

\- : Exposure problem

\- : Where to store passwords

Access rights

Protection mechanisms

Access to shared resources must be controlled

Two aims:
- Protection against users' mistakes
- Increase in reliability

Principles:

- separate policy (*what*) from mechanism (*how*)
  Important for flexibility
- Users should have as much privilege as necessary to get job done

Standard way: each user separate domain of protection
Need *trusted way* of making system privileges available
Disadvantage: primary target for breakins (`setuid`-programs in UNIX)

UNIX access rights

coarse permissions per file (`chmod`, `ls -la`)
For each file, have three categories of possible users

- owner
- group (pre-defined set of users)
- all others

owner can grant permission to

- read (r)
- write (w)
- execute program/find files in directory (x)

Access matrix

abstract view of protections
have row for each domain and column for object
Entry indicates access right
Example:

	F_1	F_2	F_3	Printer
D_1	read		write	
D_2				print

Have capabilities like read, write, copy, owner, control
Unsuitable for implementation because matrix far too large

Implementation Issues

- Access lists for objects (store columns)
- Capability lists for domains (store rows)

Capabilities allow great flexibility
Example: Hydra

- Auxiliary rights: each process can pass access to procedure to other processes
  $\Rightarrow$ dynamic change of access rights
- Rights amplifications: procedure can act on specified type on behalf of any process which is allowed to execute it
  Rights *cannot* be passed on
  $\Rightarrow$ flexible and more secure mechanism for granting higher privileges temporarily

Trusted computing

- OS is large pice of code (Linux ca. 15M LoC)
- security holes give full control to the attacker
- verification of such large programs is infeasible

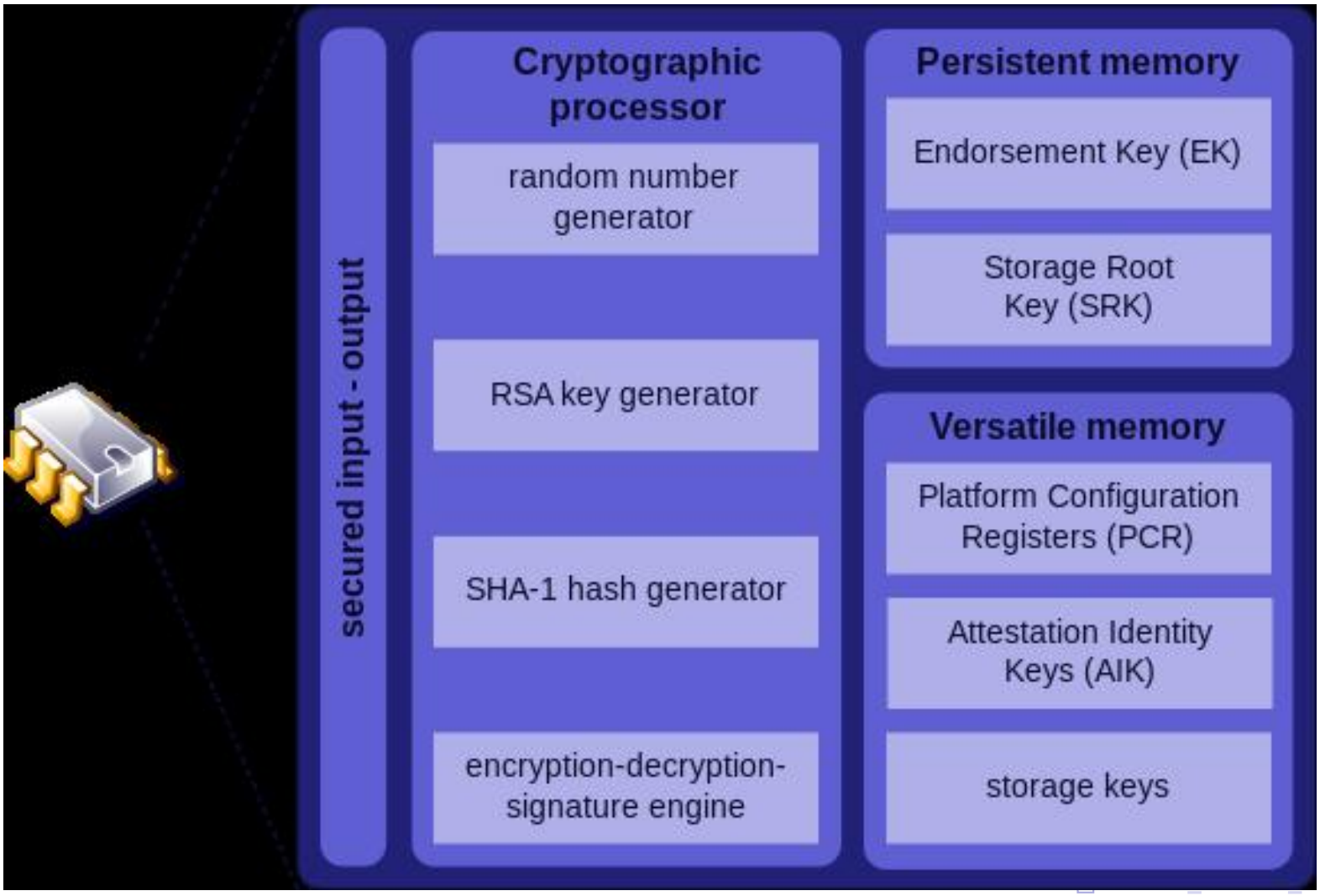
$\Rightarrow$ OS not suitable as part of *trusted computing basis* (the code which must be assumed to be secure in order to guarantee security)

Active area of research: remove OS from trusted computing basis
need to introduce additional HW as anchor for security
Example: Trusted Platform module

Trusted Platform Module

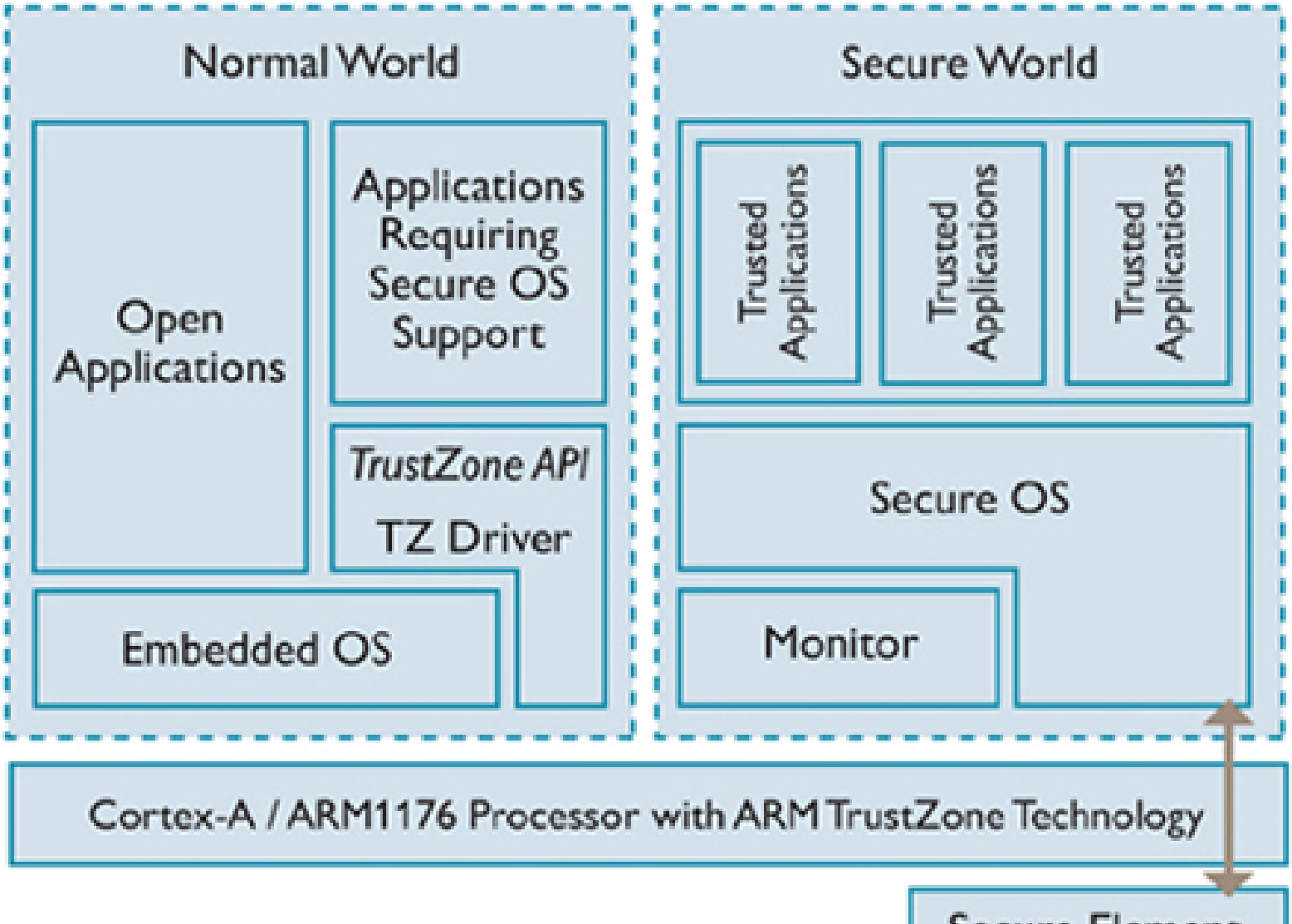
- present in a most modern laptops/desktops except Apple
- provides cryptographic operations which make it possible to
  - certify SW and HW configuration to third parties
  - provide secure boot (all files used for botting have not been corrupted)
  - secure storage
    allow access to data only if system is in known state
    achieved by linking encryption keys to system state

TPM


secured input - output
Cryptographic processor
random number generator
RSA key generator
SHA-1 hash generator
encryption-decryption-signature engine
Persistent memory
Endorsement Key (EK)
Storage Root Key (SRK)
Versatile memory
Platform Configuration Registers (PCR)
Attestation Identity Keys (AIK)
storage keys

Other approach: TrustZone
present in almost all ARM-based phones and tablets
have in addition to normal processor secure part through HW support
Important use case in Android: have private/public key pair where private key is stored within secure memory and not accessible directly
have operations to sign and decrypt using the private key
use public key to encrypt keys held in keystore.
$\Rightarrow$ prevents offline guessing attacks

TrustZone


Normal World
Open Applications
Applications Requiring Secure OS Support
TrustZone API
TZ Driver
Embedded OS
Secure World
Trusted Applications
Trusted Applications
Trusted Applications
Secure OS
Monitor
Cortex-A / ARM1176 Processor with ARM TrustZone Technology

Android specifics

Android-specific changes to the linux kernel

Need to have finer control over which devices are suspended (eg music still playing while screen turned off
Android added wakelocks to kernel

- kernel is set to suspend as soon and as often as possible
- drivers need to actively prevent system from suspending device (and therefore, kernel)
- Have several wakelocks, which keep combination of CPU, screen and keyboard awake

Using wakelocks properly is crucial to achieve long battery life

Binder

- Android uses RPCs very frequently
- Add new RPC mechanism called binder to kernel
- copies directly from user space of writer to user space of reader
- integrates capability mechanism into RPC
- use separate thread pool for incoming RPC for system services

Shared memory

Android kernel provides new shared memory mechanism (ashmem)

- provides reference counting so that kernel can reclaim unused resources
- standard use: process opens shared memory region and share obtained file descriptor through binder

Alarm timers

Standard linux kernel timers not sufficient for advanced power management
Android introduces alarm timers:

- Alarm timers wake up particular process even if system is suspended
- Application will grab wakelock once woken up

Low memory killer

- When system runs out of memory, linux kills greedy processes
- Behaviour unpredictable, could kill important components (eg telephony stack, graphic subsystem)
- Remedy: introduce separate process killer which takes time since application was last used and priority into account
- memory killer takes increasingly severe action

Security enhancements

- Assign a separate user id to each app
- Force all data exchange between apps to go via binder, with fine-grained access control mechanisms
- For system services running as root use selinux-kernel extensions, which specify which file can be accessed by which process.
- $\Rightarrow$ Vulnerabilities in system services do not immediately lead to full access

Network security

- Linux kernel allows any process to open socket and initiate network communication
- Android introduces network permission, filtered by group id
- permission needed to access IP, Bluetooth or raw sockets

Embedded Sytems

Embedded Systems

Chips with power of whole computer systems now in many applications:

- Smartphones and tablets
- Industrial control systems
- Smart cards
- Automotive control units

Characterisation of those systems:

- Fewer resources available: memory, storage space
- Often real-time applications necessary

Limited Resources

Not so much of a problem in general: OS's designed for this case
Only issue: potentially missing MMU
$\Rightarrow$ virtual memory and protection of processes against each other not implementable
Also paging not available

Real-time Operating Systems

Have two different kinds of real-time

1.) Hard real-time: completion required within a guaranteed amount of time

cannot be met by normal time-sharing systems; needs dedicated HW and adaptations to SW

2.) soft real-time: critical processes receive priority. Requires

- pre-emptive priority scheduling (plus sufficient resources to avoid starvation)
- short dispatch latency (time between arrival of process and start of execution)
  Problem: context switch normally only after syscall-completion or when I/O takes place
  way out: make kernel pre-emptible (eg Solaris 2, newer versions of Linux)
- Priority inversion: increase priority of process if resources

Distributed Systems

Aims:

- *Resource Sharing*: costly resources (high-quality printers and expensive programs) can be shared
- *Computation Speedup*: Algorithms can run concurrently
- *Reliability*: Failure of one site does not imply failure of the whole system
  Redundancy prerequisite

Especially last point difficult to fulfill

Design Issues

Several levels of Distribution possible
listed from tightly coupled to loosely coupled

- Shared memory
- Shared file system
- Bus Systems
- Switching Systems

Key Problem Areas:

- Transparency: pretend to be one computer
- Reliability: want availability, fault tolerance
- Performance
- Scalability: avoid centralised tables and algorithms

Communication

Simplest way of communication:
Client/Server Model

Idea: group of processes (servers) used by clients
Advantage: Simple communication
Simple enough to study several problems
First problem: Addressing! Possible Solutions:

- Hardware address into client code: inflexible
- Broadcasting: works only on local networks
- Name Servers: Ask special host
  Example: Domain Name Service, DNS

Second Problem:
Blocking (synchronous) vs. non-blocking primitives
Conceptual ease vs. performance

Third Problem:
Reliable vs. unreliable primitives
Where does the error correction go?
Kernel (once for all) vs. application (possibly more efficient)

Remote Procedure Call

Very simple idea: execute procedure on different host
Goal: total transparency

Basic Schema:

- Client sends arguments to server
- Server executes call
- Server sends results back

Difference to local call hidden in kernel routines
Details complicated:
- Have to transfer parameters
- deal with failures

Problems with parameter passing

- Different representation on different machines: either common format (inefficient), or store format in message
- What to do with call-by-reference parameters? Can copy arrays, but not arbitrary data structures

Failure problems more complicated
Have several cases

- Client cannot locate server: Generate exception
  $\Rightarrow$ transparency lost
- Lost Request Messages: use timer
- Lost Reply Messages: client timer insufficient:
  could execute operation more than once
  Solution: use sequence numbers
- Server Crashes: Sequence numbers not enough: When did crash occur?
  Can guarantee *at least once*, `at most once` semantics, but not *exactly once* semantics $\Rightarrow$ have to have call-specific remedies
- Client Crash: leaves orphans (unwanted computations)
  No general way of getting rid of them

Generating Timestamps

Unique timestamps needed for co-ordination
No problem in monoprocessor system: use system clock
Not possible in distributed systems
One way out:

- Each host maintains logical clock which is advanced with each event
- All message from host contain logical clock
- When message with greater logical clock is received, increase own logical clock to this value
- with two identical timestamps, let host number decide

Mutual Exclusion algorithms

A Distributed Version (Ricart and Agrawala)
Assume reliable messages, unique timestamps
Following steps:

- Process trying to enter critical section:
  sends to all other processes name of section and unique timestamp
- Process receiving such a message:
  - Sends back OK if not interested in critical section
  - Queues message if already in critical section
  - Receiver wants to enter critical section
    $\Rightarrow$ enters critical section if its request has lower timestamp and queues message, otherwise sends OK

Grants mutual exclusion without deadlock or starvation

Problems:

- Requires that everyone knows about all other hosts
- Algorithm fails if one host fails
  $\Rightarrow$ Reconstruction of network necessary in this case

Suitable for networks where configuration is stable

A Token Ring Algorithm

Assumption: Network organised on a (physical or logical) ring, i.e. each node has unique successor in line
Simple algorithm:

- At initialisation, generate token
- Pass token around continuously
- Process wanting to enter a critical region waits for token
- After leaving critical section, process passes token to next neighbour

Properties:

- Detecting lost tokens difficult: Time spent in critical region unbound
- Detecting dead processes easy if sent token is acknowledged

Election Algorithms

Problem: Select new co-ordinator
Assumption:
Know id of *every* host on the network
First example: Bully algorithm

- P sends message to all hosts with higher number
- No response $\Rightarrow$ P wins and is co-ordinator
- Answer received $\Rightarrow$ host with higher number has taken over

Second Example: Ring Algorithm

- any host may send *Election* message
- passed around the net, which each hostid added
- If original host gets message, determines co-ordinator and sends new message around
- After it has gone round, host removes it

Transaction Protocols

Need higher level of abstraction

Example: booking a flight:
Agreement on status of transaction necessary:
either completed or not happened at all

want to model this
Assumptions:

- reliable communication, but hosts may fail
  (lost messages handled by lower levels)
- Have stable storage surviving host crashes and disk failures

Properties of transactions

- *Atomic*: transaction indivisible
- *Consistent*: maintain system invariants
- *Isolated*: Concurrent transaction do not interfere
- *Durable*: Once transaction finished, changes are permanent

Implementation

Two methods used for working on data:

- *Private Workspace*:
  Copy files to host executing the transaction
  Copy results (or original files) back afterwards
- *Writeahead log*:
  modify files locally, but keep log of changes
  makes undo possible later

The Two-Phase Commit Protocol

Aim: Achieve atomicity
Requires central co-ordinator
Protocol works as follows:
(**C** co-ordinator, **S** subordinate)

- **C** writes "Prepare" in the log
- **C** sends "Prepare" message to subordinate
- **S** sends "Ready" message when it is happy to commit
- **C** collects replies
- **C** writes log record
- **C** sends "Commit" message
- **S** writes "Commit" in the log
- **S** Commit
- **S** sends finished message

Concurrency control

Need some way of serialising parallel events
First way: *File Locking*
Each host maintains list who is accessing files at the moment
Only one process allowed to access file
To avoid inconsistency:

- First aquire all locks (Growing Phase)
- Perform operations
- Release locks (Shrinking Phase)

Achieves serialisability
Deadlock possible:
Standard avoidance techniques:

- global order of files
- If lock not available, release all others and wait for random interval

Optimistic concurrency control
Check which files have been written
Any files written twice $\Rightarrow$ Abort transaction
Advantage: Deadlock free, allows maximum parallelism
Problem: Does not work with high load

Refinement: Use timestamps and abort only if transaction with higher timestamp wants to write

Dealing with failures

Distributed systems should cope with failure of one site, loss of messages etc.

Kinds of failures:

- Silent: host does not respond
- Byzantine: host sends false information

Failure detection done by handshaking protocol:

- Site A send "Are-you-up"-message to B
- B answers immediately "I am up"
- If answer not received within certain time, try again, possibly via a different route
- give up after fixed number of attempts

works for silent, not for Byzantine failures

Byzantine failures

Assumption: Processors faulty, communication reliable
Question: Can we achieve agreement between the working processors?
Possible under certain conditions (Lamport)
More than two-thirds of processors working properly
To get idea, consider 1 faulty, 4 processors in total.
Have following steps:

- Every processor sends to other one value of local variable
- All processors collect values received
- Processors pass on all values to all other processors
- Each processor decides by majority voting on values received

Distributed File Systems

Special Problems:

- *Naming:* identify files systemwide
- how are concurrent reads and writes executed?

Naming

Aims:

- *Location Transparency:*
  name does not give any hint on location
- *Location independence:*
  files can be moved without name being changed

standard approach: *Remote mounting*
Make remote file system available under local name
Achieves only location transparency
Latter difficult to achieve (requires name server)

Implementation Issues

Main issue: stateless vs. stateful servers
(Should server keep information about requests?)
Properties of stateless servers

- Fault tolerance
- No `open`/`close`-requests needed
- No problems with client crashes

Advantage of stateful servers

- Read ahead possible
- Idempotency easier
- File locking possible

Main problem: Cache consistency, especially for stateless servers

NFS

Idea: make file systems available on other hosts
Works on different architectures
$\Rightarrow$ Need well-defined protocol
RPC's used for this purpose

Stateless system
$\Rightarrow$ no `open`/`close` RPC's
Each RPC contains absolute address in file
Caching employed:

- Server does normal caching (no ill-effects)
- Client caches reads and writes
  $\Rightarrow$ obtain inconsistency
  Data sent to server only when
  - $>$ 8k written
  - file closed on client
  - timeout reached
  `open` on client checks server

Have possibly several layers of file systems (from bottom up):

- Partition on disk
- full disk encryption
- logical volumes
- file systems on logical volume

Kernel support for layering provided by device mapper
provides mapping from virtual block device provided by device mapper to other block device
works transparently: upper layer unaware of existence of lower layers
Example: PC with full disk encryption

- Partition disk into separate boot partition and partition for rest of data
- Encrypt data partition
- On encrypted data partition, create logical volumes as necessary (at least one for root and one for swap).