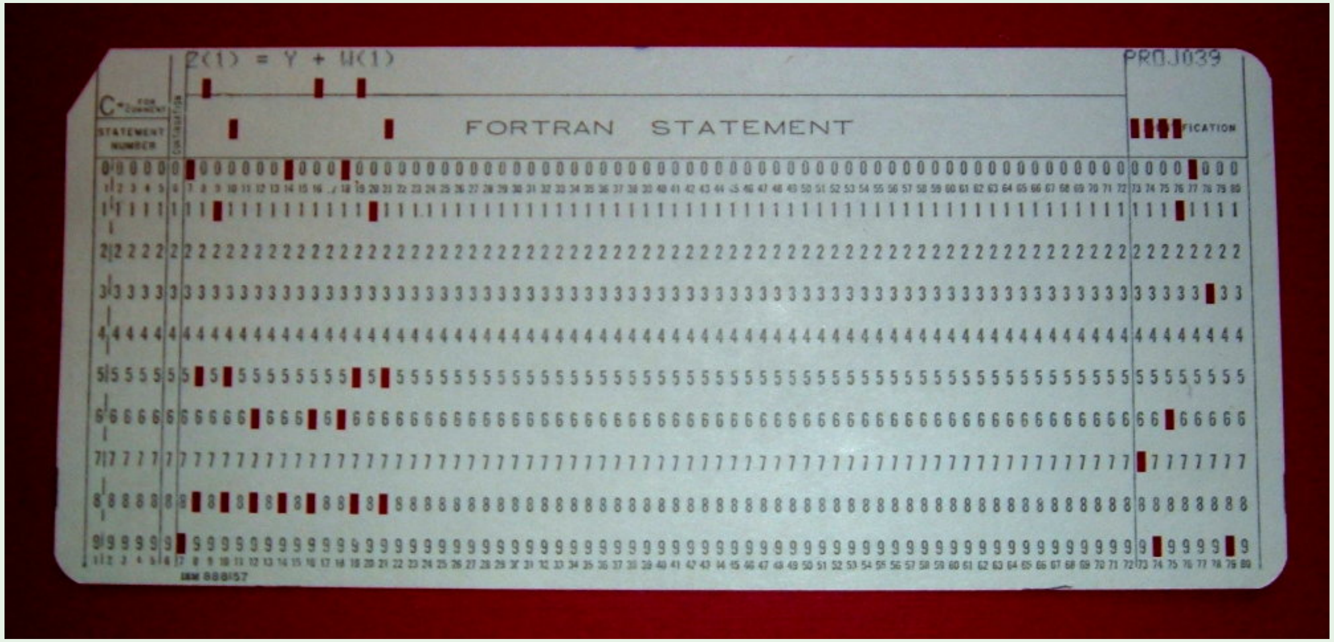
2. A Computer, What For?

- Survey of Operating System Principles

Legacy Systems

Punched Cards



Batch processing

- Interface of *"big iron" mainframes*
- Survives in script languages (UNIX shell, MSDOS `.BAT` files)
- Default paradign for job distribution on shared computation servers
  See e.g., IDRIS: http://www.idris.fr

Modern Systems Without OS

- *Most critical systems do not use an OS at all*
  - Static code generation of a (reactive) scheduler, tailored to a given set of tasks on a given system configuration
  - Synchronous languages: Lustre (Scade), Signal, Esterel
    $\longrightarrow$ main approach for closed systems like flight controllers (Airbus A320–A380)

Is it Enough?

There exist more interactive, complex, dynamic, extensible systems!

They require an *Operating System* (OS)

Operating System Tasks and Principles

Tasks

- Resource management
- Separation
- Communication

$\leftrightarrow$

Principles

- Abstraction
- Security
- Virtualization

The Kernel of the Operating System

Tasks: Resource Management, Separation, Communication

- The kernel is a *process manager*, not a process
- It runs with higher privileges (enforced by the microprocessor)
  - *User mode*: restricted instructions and access to memory
  - *Kernel mode*: no restriction, can execute priviledged operations
- User processes switch to kernel mode when requesting a service provided by the kernel
  - *System call*, asking the kernel to implement a priviledged operation on the behalf of the process
  - *Context switch*, from the kernel's *scheduler*, or due to a system call initiated by the process

2. A Computer, What For?

- Survey of Operating System Principles

First OS Principle: Abstraction

Goal

- Simplify, standardize
  - Kernel portability over multiple hardware platforms
  - Uniform interaction with devices
  - Facilitate development of device drivers
  - Stable execution environment for the user programs

Main Abstractions

1. Process
2. File and file system
3. Device
4. Virtual memory
5. Naming
6. Synchronization
7. Communication

Abstraction: Process

Single Execution Flow

- Process: *execution context of a running program*
- Modern OSes support *multiprocessing* with *private address space* for each process
  - Isolation of address spaces enforced by the OS kernel and the processor: *virtual memory*

Abstraction: Process

Multiple Execution Flows

- Within a process, the program "spawns" multiple execution flows operating within the same address space: the *threads*
- Motivation: *finer-grain concurrency* than processes
  - Less information to save/restore with the processor needs to switch from executing one thread to another (see *context switch*)
  - Inter-thread communication is (apparently) easy: plain memory accesses
- Challenge: threads need to *collaborate* when they *concurrently* access data
- Pitfall: looks simpler than distributed computing, but it is hard to keep track of data sharing in large multi-threaded programs, and even harder to get the threads to collaborate correctly (non-deterministic reproducibility problems)

More about threads in the Java language chapter

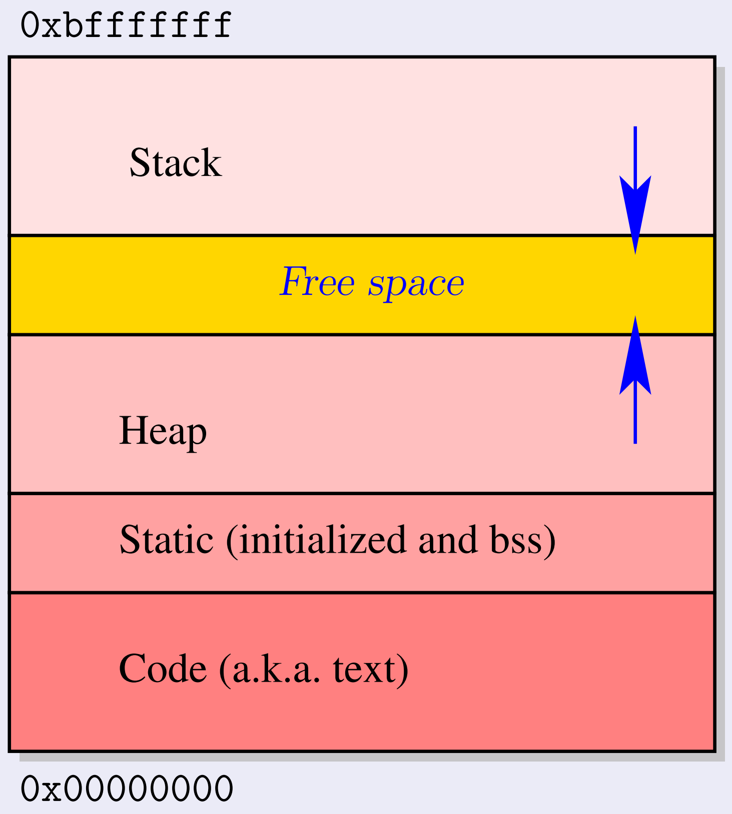
Abstraction: Virtual Memory

- Processes access memory through *virtual addresses*
  - Simulates a large *interval* of memory addresses
  - Address-space protection and separation
  - Hides kernel and other processes' memory
  - Automatic *translation* to *physical addresses* by the processor (MMU/TLB circuits)
- Principle: *paging* mechanism
  - More on this mechanism when exploring the operating system kernel
- *Swap* memory and file system
  - The ability to suspend a process and virtualize its memory allows to store its pages to disk, saving (expensive) RAM for more urgent matters
  - Same mechanism to migrate processes on NUMA multi-processors

Abstraction: Virtual Memory

Segments: Per-Process Virtual Memory Layout

- *Code* (also called *text*) segment
  - Linux: ELF format for object files (`.o` and executable)
- *Static data* segment(s)
  - Global, `static` variables
- *Stack* segment
  - Stack frames for method arguments and local variables
- *Heap* segment
  - Dynamic allocation of objects: `new`



Abstraction: File and File System

- *File*: *storage* and *naming* in UNIX
- *File System* (FS): repository (specialized database) of files
- Directory tree, absolute and relative pathnames
  `/` `.` `..` `/dev/hda1` `/bin/ls` `/etc/passwd`
- File types
  - Regular file or hard link (file name alias within a single file system)
    `$ ln pathname alias_pathname`
  - Soft link: short file containing a pathname
    `$ ln -s pathname alias_pathname`
  - Directory: list of file names (a.k.a. hard links)
  - Pipe (also called FIFO)
  - Socket (networking)
- Assemble multiple file systems through *mount points*
  Typical example: `/home` `/usr/local` `/proc`
- Common set system calls, independent of the target file system

Abstraction: Device

What do a microphone, a hard disk, a Wifi radio module have in common?

They are *devices*, “peripheral” computing or signal processing systems of their own, dedicated to Input/Output (I/O) operations

- Device special files
  - *Block*-oriented device: disks, file systems
    `/dev/hda` `/dev/sdb2` `/dev/md1`
  - *Character*-oriented device: serial ports, console terminals, audio
    `/dev/tty0` `/dev/pts/0` `/dev/usb/hiddev0` `/dev/mixer` `/dev/null`

Abstraction: Name

- Hard problem in operating systems
  - Processes are separated (logically and physically)
  - Need to access *persistent* and/or *foreign* resources
  - Resource *identification* determines large parts of the programming interface
  - Hard to get it right, general and flexible enough
- Good examples: /-separated filenames and pathnames
  - Uniform across complex directory trees
  - Uniform across multiple devices with *mount points*
  - Extensible with *file links* (a.k.a. aliases)
  - Reused for many other naming purposes: e.g., UNIX sockets, POSIX Inter-Process Communication (IPC)
- Could be better
  - INET addresses, e.g., 129.104.247.5, see the never-ending IPv6 story
  - TCP/UDP network ports
- Bad examples
  - Device numbers (UNIX internal tracking of devices)
  - Older UNIX System V IPC
  - MSDOS (and Windows) device letters (the ugly `C:\`)

Abstraction: Concurrency Primitives

Synchronization

- Interprocess (or interthread) synchronization interface
  - Waiting for a process status change
  - Waiting for a signal
  - Semaphores
  - Reading from or writing to a file (e.g., a pipe)

Communication

- Interprocess communication programming interface
  - Synchronous or asynchronous signal notification
  - Pipe (or FIFO), UNIX Socket
  - Message queue
  - Shared memory
- OS interface to network communications
  - INET Socket

Second OS Principle: Security

Basic Mechanisms

- Identification
  /etc/passwd and /etc/shadow, sessions (login)
  UID, GID, effective UID, effective GID
- Isolation of processes, memory pages, file systems
- Encryption, authentication (signature) and key management
- Logging: /var/log and syslogd daemon

Enhanced Security

- SELinux: http://www.nsa.gov/selinux/papers/policy-abs.cfm
- Android security model: http://code.google.com/android/devel/security.html
- Trusted Platform Module (TPM), ARM TrustZone
- Defining a security policy $\neq$ Enforcing a security policy

Third OS Principle: Virtualization

"Every problem can be solved with an additional level of indirection"

Third OS Principle: Virtualization

"Every problem can be solved with an additional level of indirection"

Standardization Purposes

- Common, portable interface
- Software engineering benefits (code reuse)
  - Example: Virtual File System (VFS) in Linux = superset API for the features found in all file systems
  - Another example: drivers with SCSI interface emulation (USB mass storage)
- Security and maintenance benefits
  - Better isolation than processes
  - Upgrade the system transparently, robust to partial failures

Third OS Principle: Virtualization

"Every problem can be solved with an additional level of indirection"

Compatibility Purposes

- Binary-level compatibility
  - Processor and full-system virtualization: emulation, binary translation (*subject of the last chapter*)
  - Protocol virtualization: IPv4 on top of IPv6
- API-level compatibility
  - Java: through its virtual machine and SDK
  - POSIX: even Windows has a POSIX compatibility layer
  - Relative binary compatibility across some UNIX flavors (e.g., FreeBSD)