What is an operating system?

1. A virtual machine!

... offering a more comfortable and safer environment

(e.g. memory management and protection, hardware abstraction, process management, inter-process communication, ...)
What is an operating system?

1. A virtual machine!
   ... offering a more comfortable and safer environment

2. A resource manager!
   ... coordinating access to hardware resources

Operating systems deal with:
- processors
- memory
- mass storage
- communication channels
- devices (timers, special purpose processors, peripheral hardware, ...)

and tasks/processes/programs which are applying for access to these resources!

The evolution of operating systems:

- in the beginning: single user, single program, single task, serial processing - no OS
- 50s: System monitors / batch processing
  - the monitor ordered the sequence of jobs and triggered their sequential execution
- 50s-60s: Advanced system monitors / batch processing:
  - the monitor is handling interrupts and timers
  - first support for memory protection
  - first implementations of privileged instructions (accessible by the monitor only).
- early 60s: Multiprogramming systems:
  - employ the long device I/O delays for switches to other, runnable programs
- early 60s: Multiprogramming, time-sharing systems:
  - assign time-slices to each program and switch regularly
- early 70s: Multitasking systems – multiple developments resulting in UNIX (besides others)
- early 80s: single user, single tasking systems, with emphasis on user interface or APIs
  - MS-DOS, CPM, MacOS and others first employed 'small scale' CPUs (personal computers).
- mid-80s: Distributed/multiprocessor operating systems - modern UNIX systems (SYSV, BSD)
Types of current operating systems

Personal computing systems, workstations, and workgroup servers:

- late 70s: Workstations starting by porting UNIX or VMS to ‘smaller’ computers.
- 80s: PCs starting with almost none of the classical OS-features and services, but with a user-interface (MacOS) and simple device drivers (MS-DOS)
- last 20 years: evolving and expanding into current general purpose OSs, like for instance:
  - Solaris (based on SVR4, BSD, and SunOS)
  - LINUX (open source UNIX re-implementation for x86 processors and others)
  - current Windows (used to be partly based on Windows NT, which is ‘related’ to VMS)
  - MacOS (Mach kernel with BSD Unix and a proprietary user-interface)
- Multiprocessing is supported by all these OSs to some extent.
- None of these OSs are suitable for embedded systems, although trials have been performed.
- None of these OSs are suitable for distributed or real-time systems.

Paralleling systems

- support for a large number of processors, either:
  - symmetrical: each CPU has a full copy of the operating system
  - asymmetrical: only one CPU carries the full operating system, the others are operated by small operating system stubs to transfer code or tasks.

Multiprocessing is supported by all these OSs to some extent.
None of these OSs are suitable for embedded systems, although trials have been performed.
None of these OSs are suitable for distributed or real-time systems.

Distributed operating systems

- all CPUs carry a small kernel operating system for communication services.
- all other OS-services are distributed over available CPUs
- services may migrate
- services can be multiplied in order to
  - guarantee availability (hot stand-by)
  - or to increase throughput (heavy duty servers)

Parallel operating systems

- Fast context switches?
- Small size?
- Quick response to external interrupts?
- Multitasking?
- ‘low level’ programming interfaces?
- Interprocess communication tools?
- High processor utilization?
Types of current operating systems

Real-time operating systems

- Fast context switches
  - should be fast anyway
- Small size
  - should be small anyway
- Quick response to external interrupts
  - not ‘quick’, but predictable
- Multitasking
  - often, not always
- "Low level" programming interfaces
  - needed in many operating systems
- Interprocess communication tools
  - needed in almost all operating systems
- High processor utilization
  - fault tolerance builds on redundancy!

Real-time operating systems need to provide...

- the logical correctness of the results as well as
- the correctness of the time, when the results are delivered

Predictability!
(not performance!)

All results are to be delivered just-in-time – not too early, not too late.
Timing constraints are specified in many different ways ...
... often as a response to ‘external’ events

reactive systems

Embedded operating systems

- usually real-time systems, often hard real-time systems
- very small footprint (often a few kBytes)
- none or limited user-interaction
- 90–95% of all processors are working here!
What is an operating system?

Is there a standard set of features for operating systems?

**No:**

the term 'operating system' covers 4kB microkernels, as well as > 1 GB installations of desktop general purpose operating systems.

What is an operating system?

Is there a standard set of features for operating systems?

**No:**

the term 'operating system' covers 4kB microkernels, as well as > 1 GB installations of desktop general purpose operating systems.

Is there a minimal set of features?
What is an operating system?

Is there a standard set of features for operating systems?

```
no:
The term 'operating system' covers 4kB microkernels,
as well as > 1 GB installations of desktop general purpose operating systems.
```

Is there a minimal set of features?

```
almost:
Memory management, process management and inter-process communication/synchronisation
will be considered essential in most systems.
```

Is there always an explicit operating system?

```
no:
some languages and development systems operate with standalone runtime environments.
```

Typical features of operating systems

Process management:
- Context switch
- Scheduling
- Book keeping (creation, states, cleanup)

```context switch:
needs to...
'remove' one process from the CPU while preserving its state
choose another process (scheduling)
'insert' the new process into the CPU, restoring the CPU state
```

Some CPUs have hardware support for context switching, otherwise:
- use interrupt mechanism
Typical features of operating systems

Memory management:
- Allocation/Deallocation
- Virtual memory: logical vs. physical addresses, segments, paging, swapping, etc.
- Memory protection (privilege levels, separate virtual memory segments, ...)
- Shared memory

Synchronization/Inter-process communication
- semaphores, mutexes, cond. variables, channels, mailboxes, MPI, etc. (chapter 4)
- tightly coupled to scheduling/task switching!

Hardware abstraction
- Device drivers
- API
- Protocols, file systems, networking, everything else...

Typical structures of operating systems

Monolithic (or 'the big mess...')
- non-portable
- hard to maintain
- lacks reliability
- all services are in the kernel (on the same privilege level)
- but: may reach high efficiency

- e.g. most early UNIX systems,
  MS-DOS (80s), Windows (all non-NT based versions)
  MacOS (until version 9), and many others...

Monolithic & Modular
- Modules can be platform independent
- Easier to maintain and to develop
- Reliability is increased
- all services are still in the kernel (on the same privilege level)
- may reach high efficiency

- e.g. current Linux versions

Monolithic & layered
- easily portable
- significantly easier to maintain
- crashing layers do not necessarily stop the whole OS
- possibly reduced efficiency through many interfaces
- rigorous implementation of the stacked virtual machine perspective on OSs

- e.g. some current UNIX implementations (e.g. Solaris) to a certain degree,
  many research OSs (e.g. 'THE system', Dijkstra '68)
Typical structures of operating systems

µKernels & virtual machines

- µkernel implements essential process, memory, and message handling
- all ‘higher’ services are dealt with outside the kernel → no threat for the kernel stability
- significantly easier to maintain
- multiple OSs can be executed at the same time
- µkernel is highly hardware dependent (only the µkernel needs to be ported.)
- possibly reduced efficiency through increased communications

E.g. wide spread concept: as early as the CP/M, VM/370 (79) or as recent as MacOS X (mach kernel + BSD unix), ...

µkernel, distributed systems

- µkernel implements essential process, memory, and message handling
- all ‘higher’ services are user level servers
- significantly easier to maintain
- kernel ensures reliable message passing between clients and servers
- highly modular and flexible
- servers can be redundant and easily replaced
- possibly reduced efficiency through increased communications

E.g. current research projects, L4, etc.

µKernels & client-server models

- µkernel implements essential process, memory, and message handling
- all ‘higher’ services are user level servers
- significantly easier to maintain
- kernel ensures reliable message passing between clients and servers
- highly modular and flexible
- servers can be redundant and easily replaced
- possibly reduced efficiency through increased communications

E.g. current research projects, L4, etc.

UNIX

UNIX features

- Hierarchical file-system (maintained via ‘mount’ and ‘unmount’)
- Universal file-interface applied to files, devices (I/O), as well as IPC
- Dynamic process creation via duplication
- Choice of shells
- Internal structure as well as all APIs are based on ‘C’
- Relatively high degree of portability

UNICS, UNIX, BSD, XENIX, System V, QNX, IRIX, SunOS, Ultrix, Sinix, Mach, Plan 9, NewtonStep, AIX, HP-UX, Solaris, NetBSD, FreeBSD, Linux, OPEN-STEP, OpenBSD, Darwin, QNX/Neutrino, OS X, QNX RTOS, ...
### Introduction to processes and threads

#### Processes

- **Process**: Address space + Control flow(s)
- Kernel has full knowledge about all processes as well as their states, requirements and currently held resources.

- **Kernel has full knowledge about all processes as well as their states, requirements and currently held resources**.

#### Threads

- **Threads** (individual control-flows) can be handled:
  - **Inside the OS**:
    - Kernel scheduling.
    - Thread can easily be connected to external events (I/O).
  - **Outside the OS**:
    - User-level scheduling.
    - Threads may need to go through their parent process to access I/O.
### Process Control Blocks

- **Process Id**
- **Process state:**
  - created, ready, executing, blocked, suspended, bored ...
- **Scheduling attributes:**
  - Priorities, deadlines, consumed CPU-time, ...
- **CPU state:**
  - Saved/restored information while context switches (incl. the program counter, stack pointer, ...)
- **Memory attributes / privileges:**
  - Memory base, limits, shared areas, ...
- **Allocated resources / privileges:**
  - Open and requested devices and files, ...

... PCBs (links thereof) are commonly enqueued at a certain state or condition (awaiting access or change in state)

### Symmetric Multiprocessing (SMP)

All CPUs share the same physical address space (and access to resources).

- Any process/thread can be executed on any available CPU.

### Processes ↔ Threads

Also processes can share memory and the specific definition of threads is different in different operating systems and contexts:

- Threads can be regarded as a group of processes, which share some resources (process-hierarchy).
- Due to the overlap in resources, the attributes attached to threads are less than for 'first-class-citizen-processes'.
- Thread switching and inter-thread communication can be more efficient than switching on process level.
- Scheduling of threads depends on the actual thread implementations:
  - e.g. user-level control-flows, which the kernel has no knowledge about at all.
  - e.g. kernel-level control-flows, which are handled as processes with some restrictions.
Process states

- Created: the task is ready to run, but not yet considered by any dispatcher or waiting for admission
- Ready: ready to run
  - Waiting for a free CPU
- Running: holds a CPU and executes
- Blocked: not ready to run
  - Waiting for a resource
- Suspended: states: swapped out of main memory
  - (None time critical processes)
  - Waiting for main memory space (and other resources)

Definition of terms

Time scales of scheduling
Performance scheduling

**First come, first served (FCFS)**

<table>
<thead>
<tr>
<th>Task</th>
<th>Arrival Time</th>
<th>Duration</th>
<th>Waiting Time</th>
<th>Turnaround Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>(4, 1)</td>
<td>0</td>
<td>8</td>
<td>5.9</td>
<td>12.9</td>
</tr>
<tr>
<td>(12, 3)</td>
<td>1</td>
<td>3</td>
<td>4.1</td>
<td>8.4</td>
</tr>
<tr>
<td>(16, 8)</td>
<td>5</td>
<td>8</td>
<td>2.1</td>
<td>10.8</td>
</tr>
</tbody>
</table>

Tasks have an average time between instantiations of \( T_j \) and a constant computation time of \( C_j \).

As tasks apply concurrently for resources, the actual sequence of arrival is non-deterministic. Hence, even a deterministic scheduling schema like FCFS can lead to different outcomes.

**Round Robin (RR)**

<table>
<thead>
<tr>
<th>Task</th>
<th>Arrival Time</th>
<th>Duration</th>
<th>Waiting Time</th>
<th>Turnaround Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>(4, 1)</td>
<td>0</td>
<td>8</td>
<td>1.2</td>
<td>5.8</td>
</tr>
<tr>
<td>(12, 3)</td>
<td>1</td>
<td>3</td>
<td>3.2</td>
<td>7.4</td>
</tr>
<tr>
<td>(16, 8)</td>
<td>5</td>
<td>8</td>
<td>2.8</td>
<td>11.6</td>
</tr>
</tbody>
</table>

Waiting time: 0.5, average: 1.2 – Turnaround time: 1.2, average: 5.8

- Optimized for swift initial responses.
- "Stretches out" long tasks.
- **Bound maximal waiting time**! (depended only on the number of tasks)
Summary

Operating Systems

- Operating Systems
  - Concept
  - Categories
  - Architectures

- Processes
  - Definition
  - Relation to architectures
  - Scheduling