References for this chapter

[Patterson17]
David A. Patterson & John L. Hennessy
Computer Organization and Design – The Hardware/Software Interface
Chapter 4 “The Processor”,
Chapter 6 “Parallel Processors from Client to Cloud”
ARM edition, Morgan Kaufmann 2017
What is an operating system?
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, ...)

© 2021 Uwe R. Zimmer, The Australian National University
What is an operating system?

1. A virtual machine!

... offering a more comfortable and safer environment
What is an operating system?

2. A resource manager!

... coordinating access to hardware resources
What is an operating system?

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, CP/M, 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.
Parallel operating 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.
Types of current operating 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)
Types of current operating systems

Real-time 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!
Types of current operating systems

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
  
  (not performance!)
Types of current operating 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!
Types of current operating systems

- Entertainment system
- Tail gate
- Black Box
- Window control
- Interior lights
- Mirror dimming
- Seat adjustments
- Key identification
- Cross traffic detection
- Hill start assist
- Traction control
- Blindspot detection
- Power regeneration
- Tire pressure sensors
- Image processing
- Night vision
- Speech recognition
- Dashboard
- Driver monitoring
- Engine/motor management
- Cylinder deactivation
- Start-stop system
- Transmission control
- Automated Wipers
- Displays
- A/C
- Airbags
- Seat heating
- Adaptive cruise control
- Radar/Lidar sensing
- Adaptive cruise control
- Emergency brakes
- Navigation system
- Automated parking
- Alarm system
- Lane holding
- Emergency services call

Artwork by Q. Mehdi (cc attribution license)
Types of current operating 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!

Often over 100 MPUs per car
(and some of them quite high performant)
What is an operating system?

Is there a standard set of features for operating systems?
What is an operating system?

Is there a standard set of features for operating systems?

no:

the term ‘operating system’ covers 4 kB 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?

\[\textbf{no:}\]

the term ‘operating system’ covers 4kB microkernels, as well as >1GB 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 > 1GB 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
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 > 1GB 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?
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 > 1GB 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)

 nues 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

Synchronisation / 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...
Typical structures of operating systems

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
Typical structures of operating systems

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), ...
Typical structures of operating systems

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.
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: locally and through a network
- highly modular and flexible
- servers can be redundant and easily replaced
- possibly reduced efficiency through increased communications

e.g. Java engines, distributed real-time operating systems, current distributed OSs research projects
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

UNIX, BSD, XENIX, System V, QNX, IRIX, SunOS, Ultrix, Sinix, Mach, Plan 9, NeXTSTEP, AIX, HP-UX, Solaris, NetBSD, FreeBSD, Linux, OPEN-STEP, OpenBSD, Darwin, QNX/Neutrino, OS X, QNX RTOS, ... ... ...
1 CPU per control-flow

Specific configurations only, e.g.:

- Distributed μcontrollers.
- Physical process control systems:
  1 cpu per task, connected via a bus-system.

Process management (scheduling) not required.

Shared memory access need to be coordinated.
Introduction to processes and threads

1 CPU for all control-flows

- OS: emulate one CPU for every control-flow:
  Multi-tasking operating system
- Support for memory protection essential.
- Process management (scheduling) required.
- Shared memory access need to be coordinated.
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.
**Introduction to processes and threads**

**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.
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.
Introduction to processes and threads

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.
Introduction to processes and threads

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)
Process states

- **created**: the task is ready to run, but not yet considered by any dispatcher 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
Process states

- **created**: the task is ready to run, but not yet considered by any dispatcher (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))
Process states

- **created**: the task is ready to run, but not yet considered by any dispatcher
  - 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)
  - dispatching and suspending can now be independent modules
### Process states

- **Creation**
- **Admitted**
- **Ready**
  - Ready, suspended
  - Blocked, suspended
- **Blocked**
- **Executing**
- **Suspension**
  - Swap-out
  - Block or synchronize
- **Termination**

**Pre-emption or cycle done**
**Definition of terms**

**Time scales of scheduling**

- **Long-term**
  - creation
  - admission
  - batch

- **Short-term**
  - ready
  - dispatch
  - executing

- **Medium-term**
  - ready, suspended
  - suspend (swap-out)
  - suspend (swap-out)

- **Blocked**
  - suspended
  - swap-out
  - block or synchronize

- **Termination**
  - terminate.
Tasks have an **average time between instantiations** of $T_i$
and a constant **computation time** of $C_i$.
Performance scheduling

First come, first served (FCFS)

Waiting time: 0..11, average: 5.9 – Turnaround time: 3..12, average: 8.4

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.
Performance scheduling

First come, first served (FCFS)

In this example:
- the average waiting times vary between 5.4 and 5.9
- the average turnaround times vary between 8.0 and 8.4

Shortest possible maximal turnaround time!
Performance scheduling

Round Robin (RR)

- Waiting time: 0.5, average: 1.2
- Turnaround time: 1.20, 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