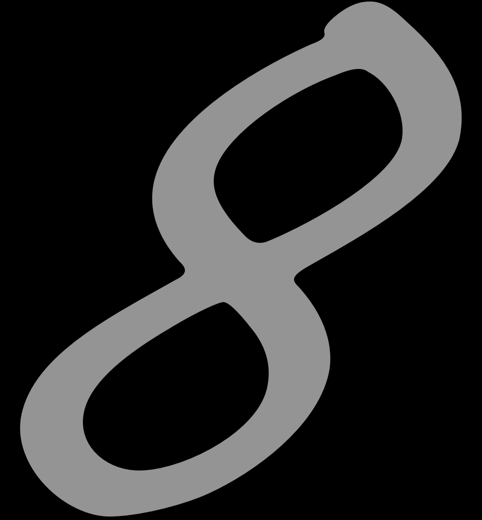
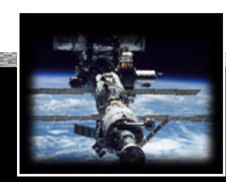


Architectures

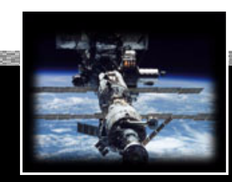
Uwe R. Zimmer
The Australian National University



References for this chapter

[Bacon98]

J. Bacon
Concurrent Systems
1998 (2nd Edition)
Addison Wesley Longman Ltd,
ISBN 0-201-17767-6



Language architectures

(Some workfloor languages are already introduced at this point, so we turn to another style of clean concurrent architectures here)

occam 2.1

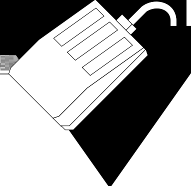
William of Ockham (born at Ockham in Surrey (England) in 1280 and died in Munich in 1349):

- Philosopher and Franciscan monk
- Reasoning in the frame of the school of Nominalism:
    - … science has nothing to do directly with things, but only with concepts of them
    - … leading to the absolute subjectivity of all concepts and universals
- Pioneer of modern Epistemology
  (will also help to develop the concept of Phenomenology 500 years later)

- ‘Occam’s razor’:

“Pluralitas non est ponenda sine neccesitate”
or “plurality should not be posited without necessity”

(a common place in medieval philosophy)



occam 2.1

Origins:

- EPL (Experimental Programming Language) by David May
- CSP (Communicating Sequential Processes) by Tony Hoare
- “Dijkstra-Style” programming

Goals:

- Minimalist approach (☞ Occam’s razor) supplying all means for:

☞ Concurrency & communication,
☞ Distributed systems
☞ Realtime / Predictable systems

occam 2.1

Implementations:

- Transputer networks as an hardware implementation of the occam architecture (inmos, now SGS-Thomson)
- spoc (Southampton Portable occam Compiler)
- KRoC (Kent Retargetable Occam Compiler)

Historical:

- 1982: First conception
- 1992: occam 3 (draft)
- 1994: latest complete version: 2.1

Current state: academic (education)

occam 2.1

Characteristics (... everything is a process):

- Primitive processes are
    - *assignments*
    - *input*, or *output* statements (channel operations)
    - ***SKIP***, or ***STOP*** *(elementary processes)*

- Constructors are:
    - **SEQ** (sequence) + replication
    - **PAR** (parallel) + replication
    - **ALT** (alternation) + replication + priorities

    - **IF** (conditional) + replication
    - **CASE** (selection)
    - **WHILE** (conditional loop)

© 2006 Uwe R. Zimmer, The Australian National University

occam 2.1

Characteristics (… everything is a process and static):

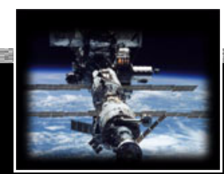
- ☞ no dynamic process creation
- ☞ no unlimited recursion

Syntax structure:

- Indention is used block indication (instead of ‘begin-end brackets’)

Scope of names:

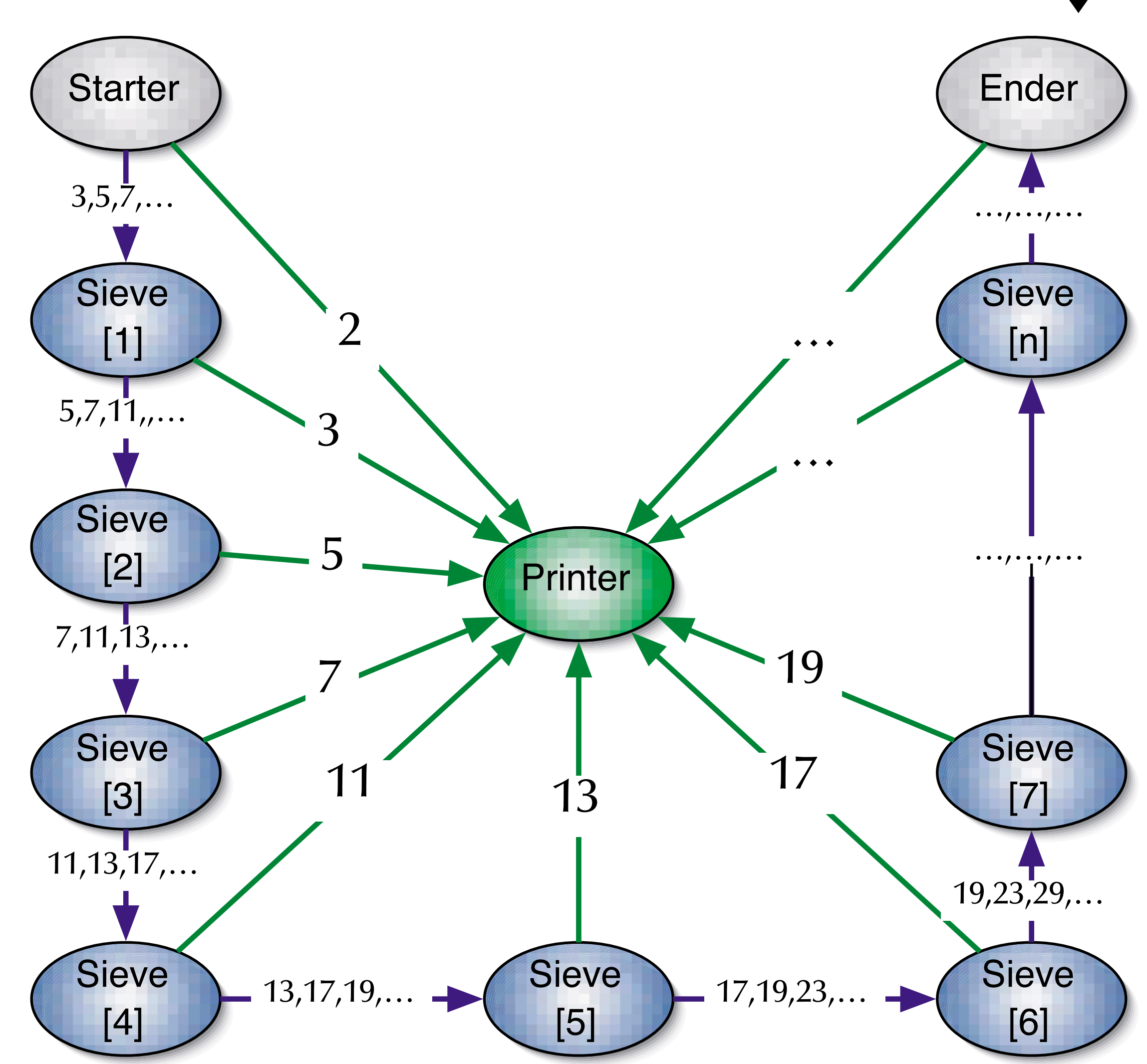
- strictly local, indicated by indention
- no ‘forward declarations’, ‘exports’, ‘global variables’, or ‘shared memories’

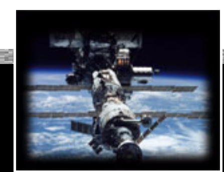


occam 2.1

An example

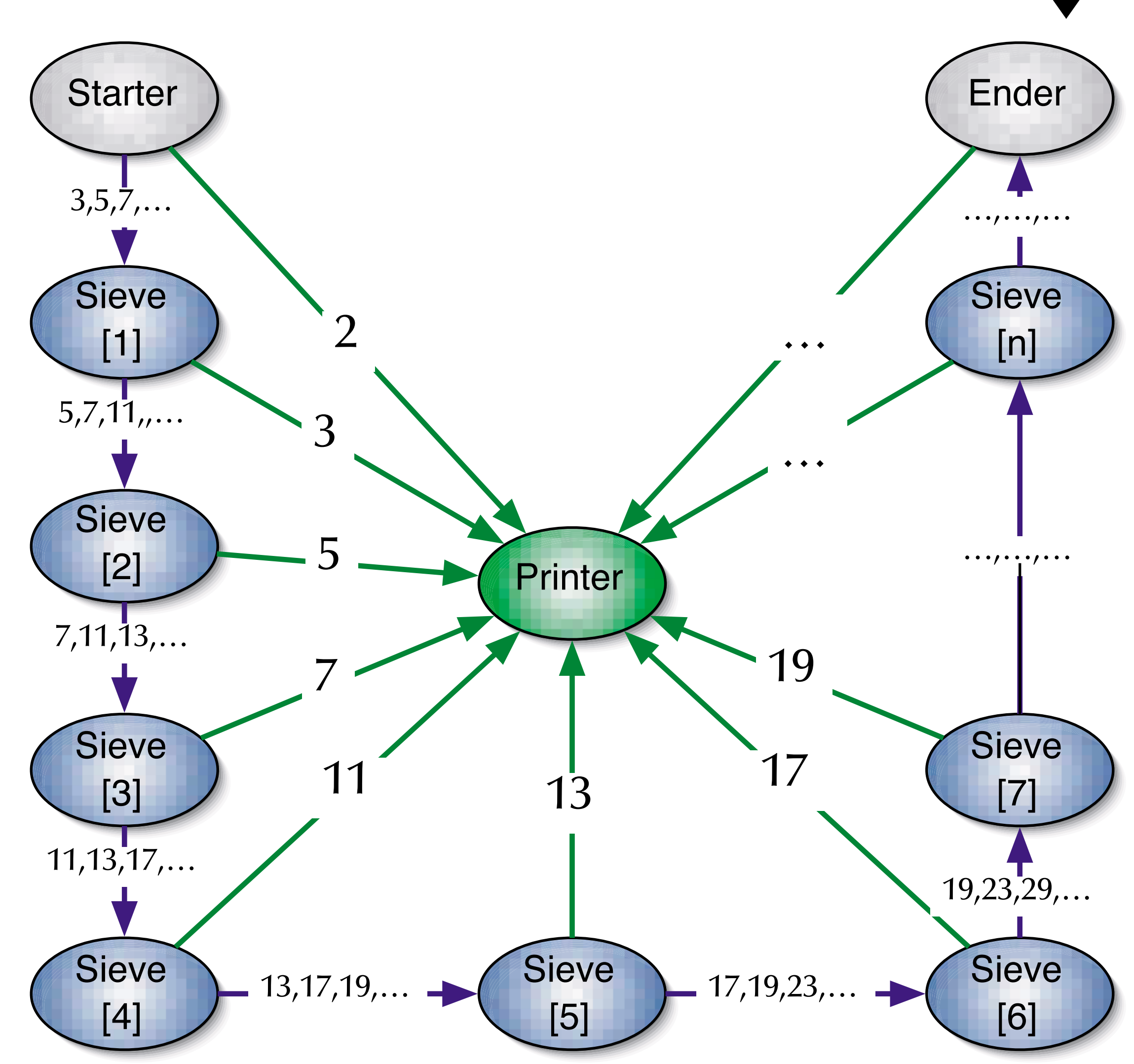
- use processes and channels
  to implement a simple prime sieve





occam 2.1

```
VAL INT n IS 50:
    -- # of primes to be generated

VAL INT limit is 1000:
    -- range to check

[n-2] CHAN of INT link:
    -- links between filters

[n-1] CHAN of INT prime:
    -- channels to Print process

CHAN OF INT display:
PLACE display AT 1:
    -- output display to device 1
```



occam 2.1

```
PROC Starter
  (CHAN OF INT out, print)
    -- feed number into the chain

INT i:
  SEQ
    print ! 2  -- 2 is prime
    i := 3
    WHILE i < limit
      SEQ
        out ! i
        i := i + 2:
          -- generate odd numbers
```

```
PROC Sieve
  (CHAN OF INT in, out, print)
    -- filter out one prime

INT p, next:
  SEQ
    in ? p
    print ! p  -- p is prime
    WHILE TRUE
      SEQ
        in ? next
        IF
          (next\p) <> 0 -- remainder?
            out ! next
          TRUE
            SKIP
```

occam 2.1

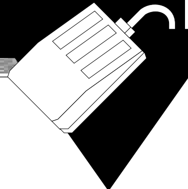
```
PROC Ender
  (CHAN OF INT in, print)
    -- consume rest of numbers

INT p:
  SEQ
    in ? p
    print ! p  -- p is prime
    WHILE TRUE
      in ? p:
```

```
PROC Printer ([] CHAN OF INT value)
    -- print each prime, in order

INT p:
  SEQ i = 0 FOR SIZE value
    SEQ
      value [i] ? p
      display ! p:


PAR -- main program

  Starter (link [0], prime [0])
  PAR i = 1 FOR n-2
    Sieve (link [i-1],
           link [i],
           prime [i])
  Ender (link [n-1], prime [n-1])
  Printer (prime)
```

occam 2.1 versus Ada95

	occam 2.1	Ada95
Addressing:	one-to-one	many-to-one
message formats defined by:	the channels' profiles	the 'accepting' tasks' parameter profiles
synchronization form:	rendezvous	
data-flow:	one way	one way or two ways (extended rendezvous)
selection of open alternatives:	non-deterministic	
Processes:	static	dynamic
shared memory ('monitors'):	-	yes

Operating systems architectures



Operating System based architectures

Hardware environments / configurations:

- stand-alone, universal, single-processor machines
- symmetrical multiprocessor-machines
- local distributed systems
- open, web-based systems
- dedicated/embedded computing

What is the common ground for operating systems?

What is an operating system?

© 2006 Uwe R. Zimmer, The Australian National University

What is an operating system?

1. A virtual machine!

… offering a more comfortable, robust, reliable, flexible … machine

Typ. real-time system

Typ. embedded system

© 2006 Uwe R. Zimmer, The Australian National University

What is an operating system?

2. A resource manager!

… dealing with all sorts of devices and coordinating access

Operating systems deal with

- processors,
- memory
- mass storage
- communication channels
- devices
  (timers, special purpose processors, interfaces, …)

☞ and many tasks/processes/programs, which are applying for access to these resources

© 2006 Uwe R. Zimmer, The Australian National University

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

☞ **no**,
the term 'operating systems' covers 4 KB kernels,
as well as 1 GB installations of general purpose OSs.

Is there a minimal set of features?

☞ **almost**,
memory management, *process management* and *inter-process communication/synchronization* will be considered essential in most systems.

Is there always an explicit operating system?

☞ **no**,
some languages and development systems operate with stand-alone run-time-environments.

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, runable 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 (MacOS) 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)

The evolution of communication systems

- 1901: first wireless data transmission (Morse-code from ships to shore)
- ‘56: first transmission of data through phone-lines
- ‘62: first transmission of data via satellites (Telstar)
- ‘69: ARPA-net (predecessor of the current internet)
- 80s: introduction of fast local networks (LANs): ethernet, token-ring
- 90s: mass introduction of wireless networks (LAN and WAN)

Currently: standard consumer computers come with

- High speed network connectors (e.g. GB-ethernet)
- Wireless LAN (e.g. IEEE802.11g)
- Local device bus-system (e.g. firewire)
- Wireless local device network (e.g. bluetooth)
- Infrared communication (e.g. IrDA)
- Modem/ADSL

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 an user-interface (MacOS) and simple device drivers (MS-DOS)

☞ last 20 years: evolving and expanding into current general purpose OSs:

- Solaris (based on SVR4, BSD, and SunOS)
- LINUX (open source UNIX re-implementation for x86 processors and others)
- current Windows (proprietary, partly based on Windows NT, which is 'related' to VMS)
- MacOS X (Mach kernel with BSD Unix and an proprietary user-interface)

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

© 2006 Uwe R. Zimmer, The Australian National University

Types of current operating systems

Parallel operating systems

- support for a large number of processors, either:
    - symmetrical:
      each CPU has a full copy of the operating system

  or
    - 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)

© 2006 Uwe R. Zimmer, The Australian National University

Types of current operating systems

Real-time operating systems

- Fast context switches?
- Small size?
- Quick responds 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 responds to external interrupts?~~ ☞ not 'quick', but predictable
- ~~Multitasking?~~ ☞ real time systems are often multitasking systems
- ~~'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 requesting …

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

(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

Types of current operating systems

Embedded operating systems

- usually real-time systems, often hard real-time systems
- very small footprint (often a few KBs)
- none or limited user-interaction

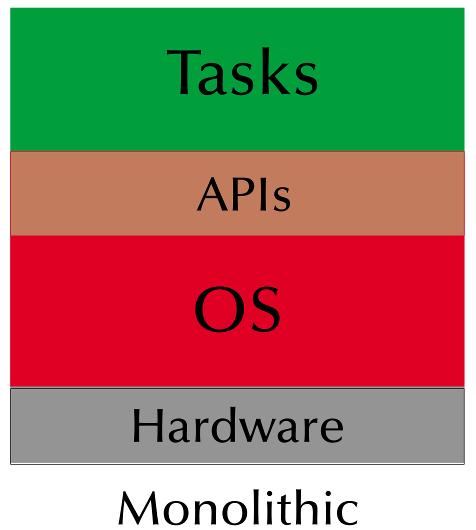
☞ 90-95% of all processors are working here!

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)

☞ may reach very high efficiency



Monolithic

e.g. most early UNIX implementations (70s),
MS-DOS (80s), Windows (basically all versions besides NT and NT-based editions),
MacOS (until version 9), … and many others …

© 2006 Uwe R. Zimmer, The Australian National University

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)

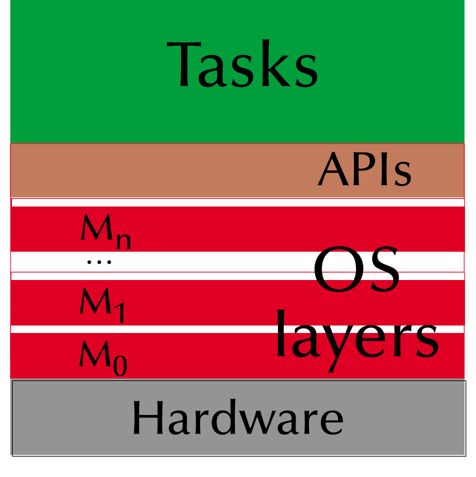
☞ may reach very high efficiency

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



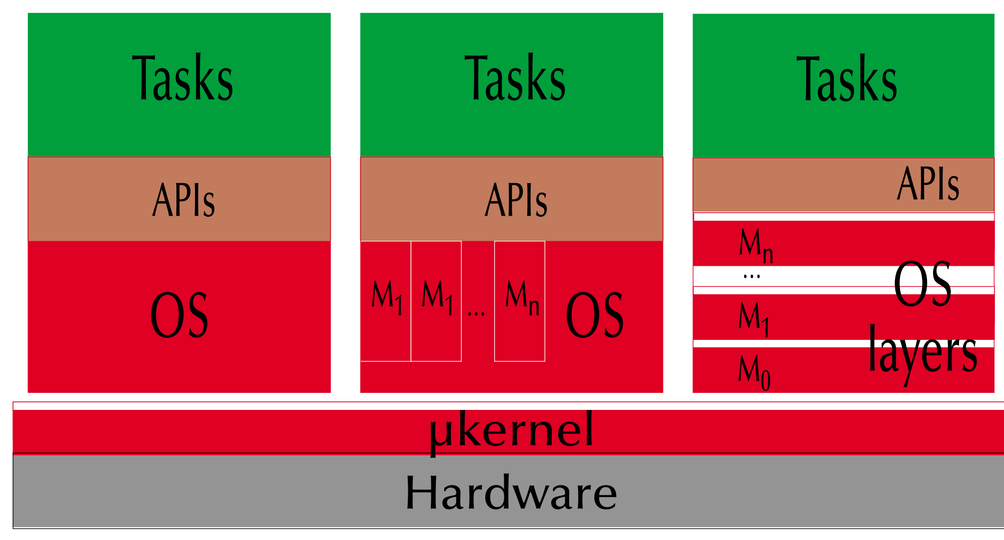
Layered

e.g. some current UNIX implementations (e.g. Solaris) to a certain degree, many research OSs (e.g. 'THE system', Dijkstra '68)

© 2006 Uwe R. Zimmer, The Australian National University

'µkernels and 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 need to be ported.
- possibly reduced efficiency through increased communications



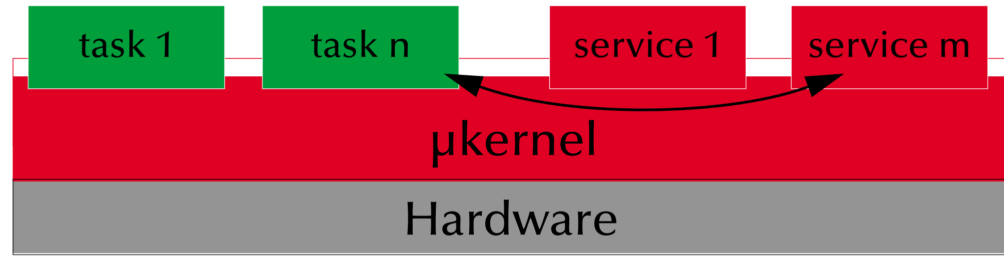
µkernel, virtual machine

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

© 2006 Uwe R. Zimmer, The Australian National University

'µkernels and client-server models'

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



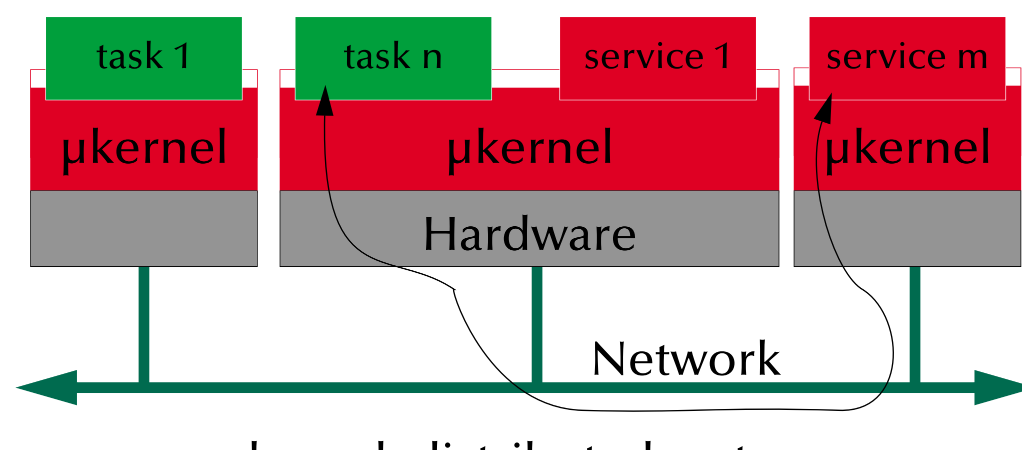
µkernel, client server structure

e.g. current µkernel research projects

© 2006 Uwe R. Zimmer, The Australian National University

‘µkernels and distributed systems’

- µkernel implements essential process, memory, and message handling
- all ‘higher’ services are user-level servers
- kernel ensures reliable message passing between clients and servers: locally and via a communication system
- highly modular and flexible
- servers can be redundant and easily replaced
- possibly reduced efficiency through increased communications



µkernel, distributed systems

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

© 2006 Uwe R. Zimmer, The Australian National University

UNIX features

- **Hierarchical file-system** (maintained via 'mount' and 'demount')
- **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, NeXTSTEP, **AIX**, HP-UX, **Solaris**, **NetBSD**, **FreeBSD**, **Linux**, OPENSTEP, **OpenBSD**, **Darwin**, **QNX/Neutrino**, **OS X**, **QNX RTOS**, … … …

Dynamic process creation

```
pid = fork ();
```

resulting in a *duplication* of the *current* process

- returning **0** to the newly created process (the 'child' process)
- returning the **process id** of the child process to the creating process (the 'parent' process) or **-1** for a failure

Frequent usage:

```
if (fork () == 0) {
  … the child's task …
  … often implemented as: exec ("absolute path to executable file", "args");
  exit (0);                      /* terminate child process */
} else {
  … the parent's task …
  pid = wait ();                 /* wait for the termination of one child process */
}
```

Synchronization in UNIX ☞ Signals

```
#include <unistd.h>
#include <sys/types.h>
#include <signal.h>

pid_t id;

void catch_stop (int sig_num)
{
   /* do something with the signal */
}
```

```
id = fork ();

if (id == 0) {

  signal (SIGSTOP, catch_stop);
  pause ();
  exit (0);

  }

} else {

  kill (id, SIGSTOP);
  pid = wait ();

}
```

Message passing in UNIX ☞ Pipes

```
int data_pipe [2], c, rc;

if (pipe (data_pipe) == -1) {
 perror (“no pipe“); exit (1);
}

if (fork () == 0) {
 close (data_pipe [1]);
 while ((rc = read
   (data_pipe [0], &c, 1)) > 0) {
    putchar (c);
 }
 if (rc == -1) {
  perror (“pipe broken“);
  close (data_pipe [0]);
  exit (1);
 }
 close (data_pipe [0]); exit (0);
} else {

 close (data_pipe [0]);
 while ((c = getchar ()) > 0) {
  if (write
   (data_pipe[1], &c, 1) == -1) {
    perror (“pipe broken“);
    close (data_pipe [1]);
    exit (1);
  };
 }
 close (data_pipe [1]);
 pid = wait ();
}
```

Processes & IPC in UNIX

Processes:

- Process creation results in a duplication of address space ('copy-on-write' becomes necessary)

☞ inefficient, but can generate new tasks out of any user process – no shared memory!

Signals:

- limited information content, no buffering, no timing assurances (signals are *not* interrupts!)

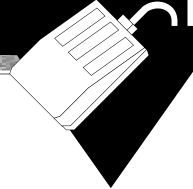
☞ very basic, yet not very powerful form of synchronization

Pipes:

- unstructured byte-stream communication, access is identical to file operations

☞ not sufficient to design client-server architectures or network communications

© 2006 Uwe R. Zimmer, The Australian National University



Sockets in BSD UNIX (also in System V.R4)

Sockets try to keep the paradigm of a universal file interface for everything and introduce:

Connectionsless interfaces (e.g. UDP/IP):

- Server side: `socket` ➠ `bind` ➠ `recvfrom` ➠ `close`
- Client side: `socket` ➠ `sendto` ➠ `close`

Connection oriented interfaces (e.g. TCP/IP):

- Server side: `socket` ➠ `bind` ➠ {`select`} [`connect` | `listen` ➠ `accept`
  ➠ `read` | `write` ➠ [`close` | `shutdown`]
- Client side: `socket` ➠ `bind` ➠ `connect` ➠ `write` | `read` ➠ [`close` | `shutdown`]

© 2006 Uwe R. Zimmer, The Australian National University

POSIX

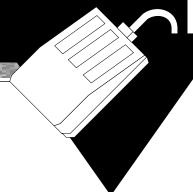
Portable Operating System Interface for Computing Environments

- IEEE/ANSI Std 1003.1 and following
- Program Interface (API) [C Language]
- more than 30 different POSIX standards
  (a system is 'POSIX compliant', if it implements parts of just one of them!)

POSIX – some of the real-time relevant standards

1003.1 12/01	OS Definition	single process, multi process, job control, signals, user groups, file system, file attributes, file device management, file locking, device I/O, device-specific control, system database, pipes, FIFO, …
1003.1b 10/93	Real-time Extensions	real-time signals, priority scheduling, timers, asynchronous I/O, prioritized I/O, synchronized I/O, file sync, mapped files, memory locking, memory protection, message passing, semaphore, …
1003.1c 6/95	Threads	multiple threads within a process; includes support for: thread control, thread attributes, priority scheduling, mutexes, mutex priority inheritance, mutex priority ceiling, and condition variables
1003.1d 10/99	Additional Real-time Extensions	new process create semantics (spawn), sporadic server scheduling, execution time monitoring of processes and threads, I/O advisory information, timeouts on blocking functions, device control, and interrupt control
1003.1j 1/00	Advanced Real-time Extensions	typed memory, nanosleep improvements, barrier synchronization, reader/writer locks, spin locks, and persistent notification for message queues
1003.21 -/-	Distributed Real-time	buffer management, send control blocks, asynchronous and synchronous operations, bounded blocking, message priorities, message labels, and implementation protocols

© 2006 Uwe R. Zimmer, The Australian National University



POSIX – 1003.1b

Frequently employed POSIX features include:

- **Threads**: a common interface to threading - differences to 'classical UNIX processes'
- **Timers**: delivery is accomplished using POSIX signals
- **Priority scheduling**: fixed priority, 32 priority levels
- **Real-time signals**: signals with multiple levels of priority
- **Semaphore**: named semaphore
- **Memory queues**: message passing using named queues
- **Shared memory**: memory regions shared between multiple processes
- **Memory locking**: no virtual memory swapping of physical memory pages

© 2006 Uwe R. Zimmer, The Australian National University

POSIX – other languages

POSIX is a ‘C’ standard …

… but **bindings to other languages** are also (suggested) POSIX standards:

- Ada: 1003.5*, 1003.24 (some PAR approved only, some withdrawn)
- Fortran: 1003.9 (6/92)
- Fortran90: 1003.19 (withdrawn)

… and there are POSIX standards for **task-specific POSIX profiles**, e.g.:

- Super computing: 1003.10 (6/95)
- Realtime: 1003.13, 1003.13b (3/98)
  - profiles 51-54: combinations of the above RT-relevant POSIX standards ☞ RT-Linux
- Embedded Systems: 1003.13a (PAR approved only)

Architectures

- **Academic**
    - occam 2.1, CSP, …
- **Workfloor**
    - Ada95, Java, …
- **Environments / Operating Systems**
    - Operating systems architectures
    - UNIX as a concept and basic UNIX features
    - POSIX