OS Architecture

Chen Yu
Tsinghua University

Topics

- THE: History of OS Architecture
- UNIX kernel
- Micro Kernel
- ExoKernel
- Extensible Kernel

https://github.com/chyyuu/os_papers/blob/master/readinglist.md#os-architecture

THE: History of OS Architecture

The Structure of THE Multiprogramming System

Outline

- Background
- Contributions
- Platform & Systems goals
- System Components
- Layered structure

Background

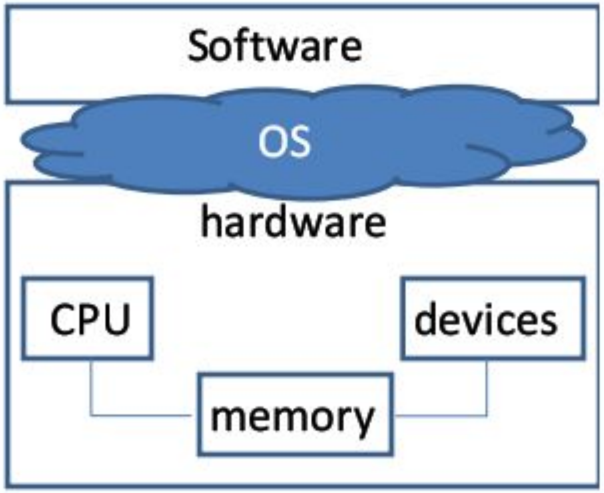
Platform (Hardware)

- Dutch Electrologica EL X8 computer
  - 32K core memory (cycle time *2.5usec*)
  - 512K words drum (1024 words per track, 40msec rev)
  - Low capacity channels supporting peripherals
    - (3 paper tape readers and punches, printer, plotter, and 2 teleprinter)
  - An indirect addressing (suited for stack implementation)
  - A sound control of interrupts and peripherals

Background

Multiprogramming

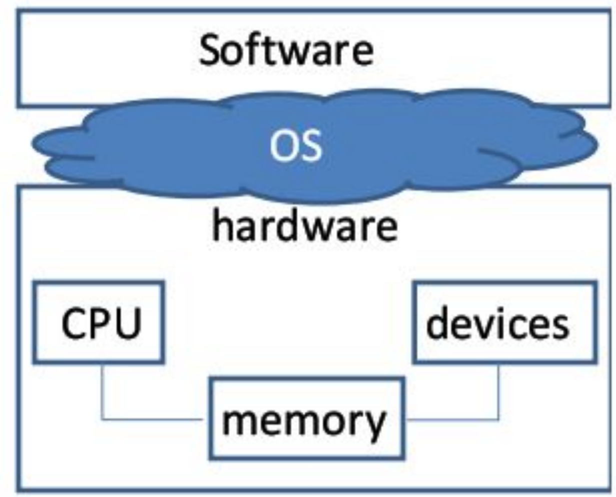
- Multiple programs running concurrently
- Comparison with uni-programming?



Background

Multiprogramming

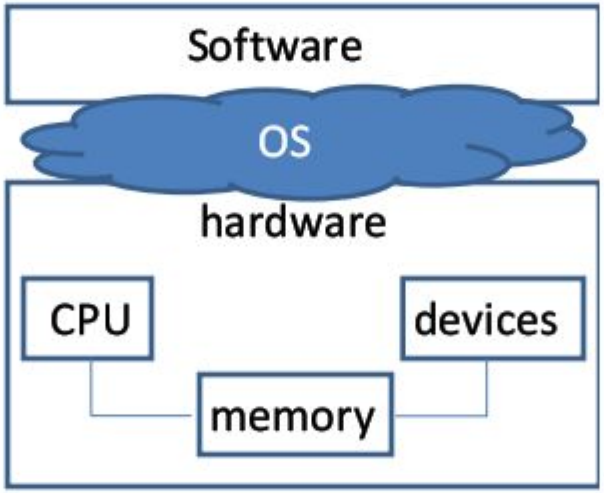
- Multiple programs running concurrently
- Comparison with uni-programming?
  - Better utilization of resource
  - Shorter turn-around time for short job
  - Process
    - Execution unit
    - Resource unit



Background

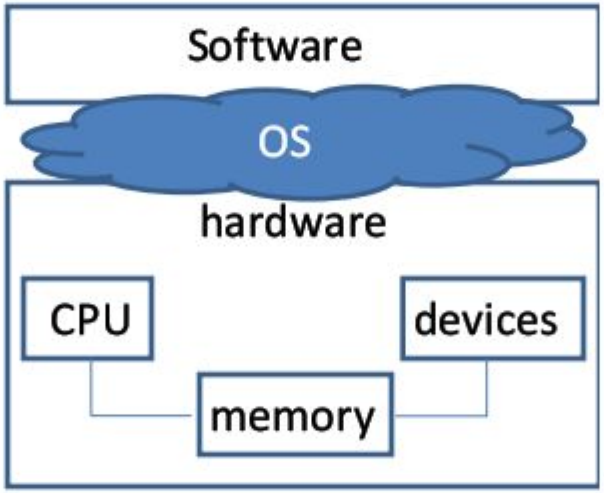
Virtual memory

- Illusion of memory
- Comparison with non-VM ?



Background

- Virtual memory
  - Illusion of memory
  - Comparison with non-VM ?
    - Faster? No
    - Productivity
    - Protection
    - Portability



Background

- Q: How to design an OS
  - The Basic task of OS?

Background

- Q: How to design an OS
  - The Basic task of OS?

Protection & Management

Background

- Q: How to design an OS
  - The Basic task of OS? Protection & Management
- Q: multi-programming -> VM ?
  - How to protect if no hardware support?
-

Background

- Q: How to design an OS
  - The Basic task of OS?   Protection   &   Management
- Q: multi-programming -> VM ?
  - How to protect if no hardware support?
- Q: VM -> multi-programming?

Background

- Q: How to design an OS
  - The Basic task of OS? Protection & Management
- Q: multi-programming -> VM ?
  - How to protect if no hardware support?
- Q: VM -> multi-programming?
  - Why do we have VM?
  - Paging (THE calls page-segment)
  - Divide program space to fixed-size units (unit) for the illusion
  - Q: VM -> paging?

Contributions

- Layered Structure of OS
  - Upper level calls/invokes lower level
  - User programs are at the highest level
- Concurrent Programming (semaphores)
- Memory segments (virtual addresses)
- Proof of correctness of the system
  - Support for reliability (verification/testing)

Layered Structure

Miss something?

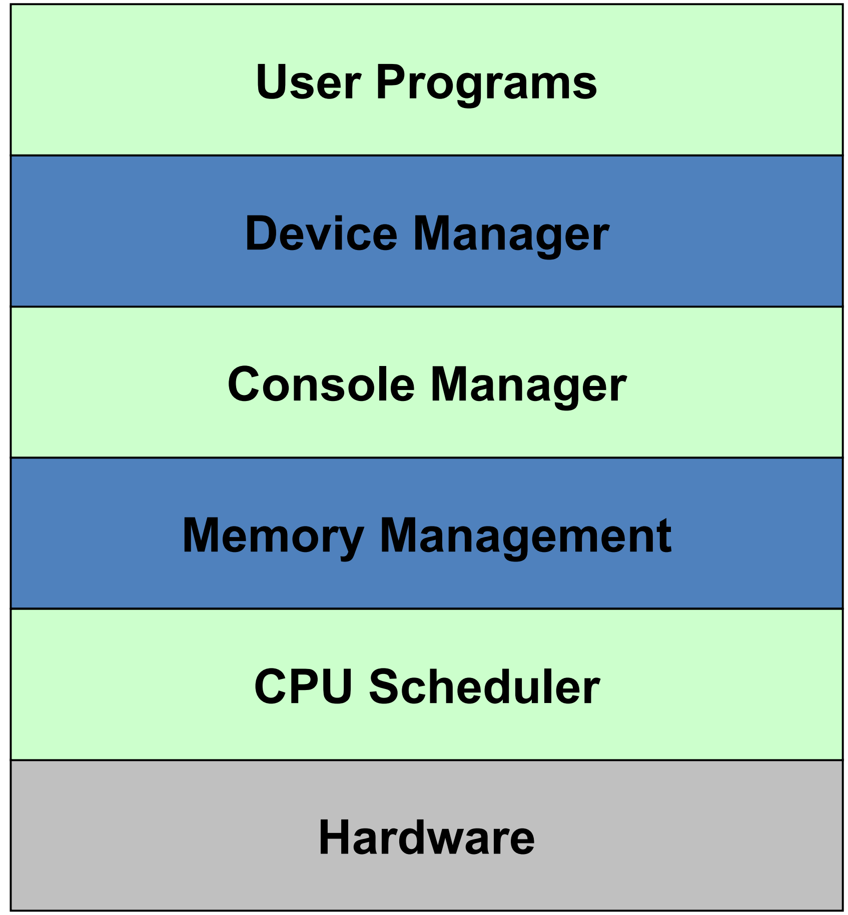
Pros & Cons ?

Virtualized i/o streams

Private console

One huge/unlimited memory space

One processor, never interrupted



Programs written in ALGOL (only in ALGOL)

Mutual Exclusion

- **begin semaphore** *mutex*; *mutex* := 1;

```
parbegin
   begin L1:
      P(mutex);
         critcal section 1;
      V(mutex);
      remainder of cycle 1;
   go to L1 end;
   begin L2:
      P(mutex);
         critical section 2;
      V(mutex); remainder of cycle 2;
   go to L2 end
parend end
```

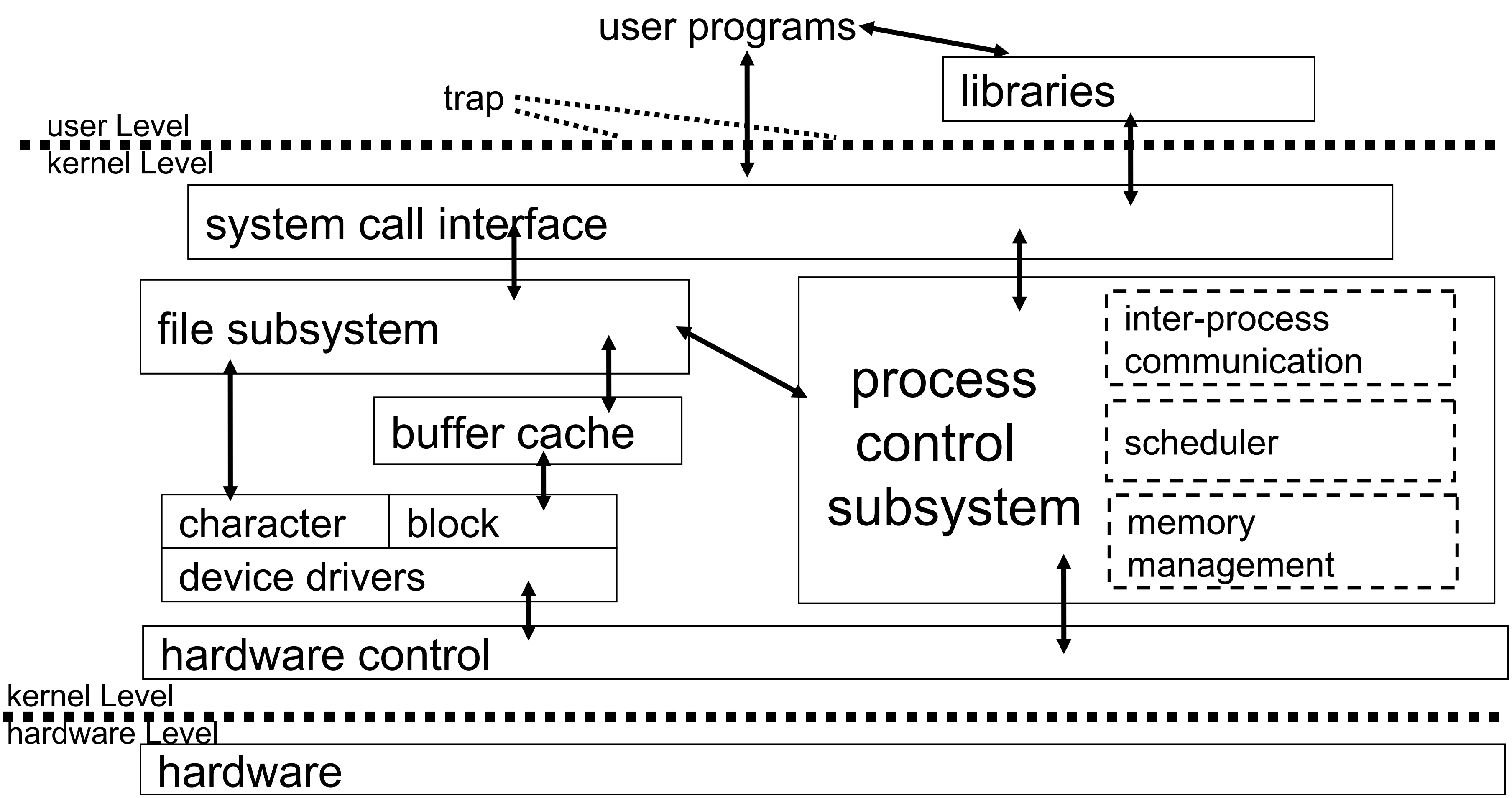
Is THE a good system at that time?

Yes

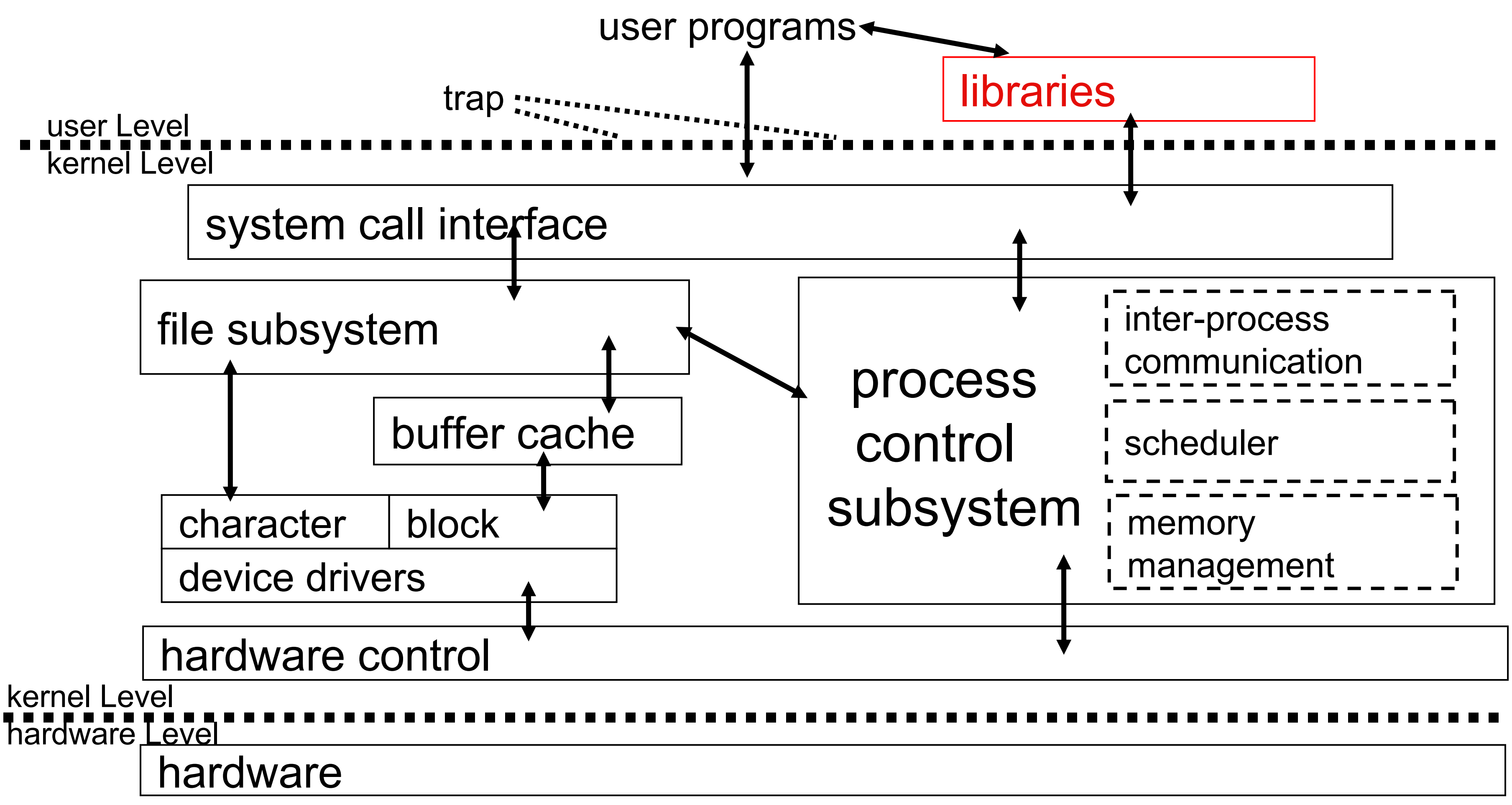
- Performance
  - 20% slower than single machine
  - Short turn-around time (latency) for short jobs
  - Benefit from the multi-programming design choice
- Programmability
  - Big memory
  - Benefit from the VM design choice
- Reliability

UNIX Kernel

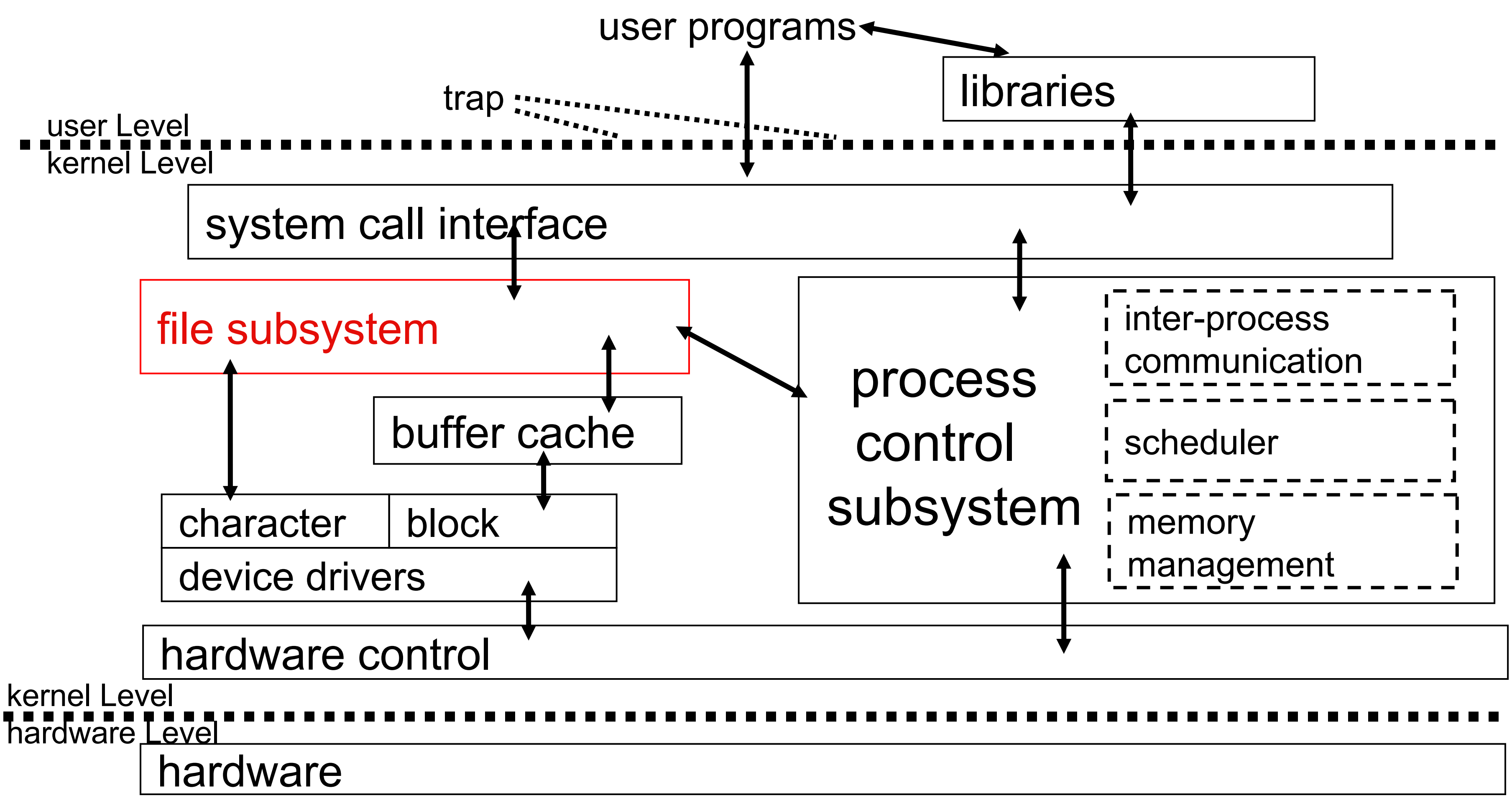
UNIX Architecture



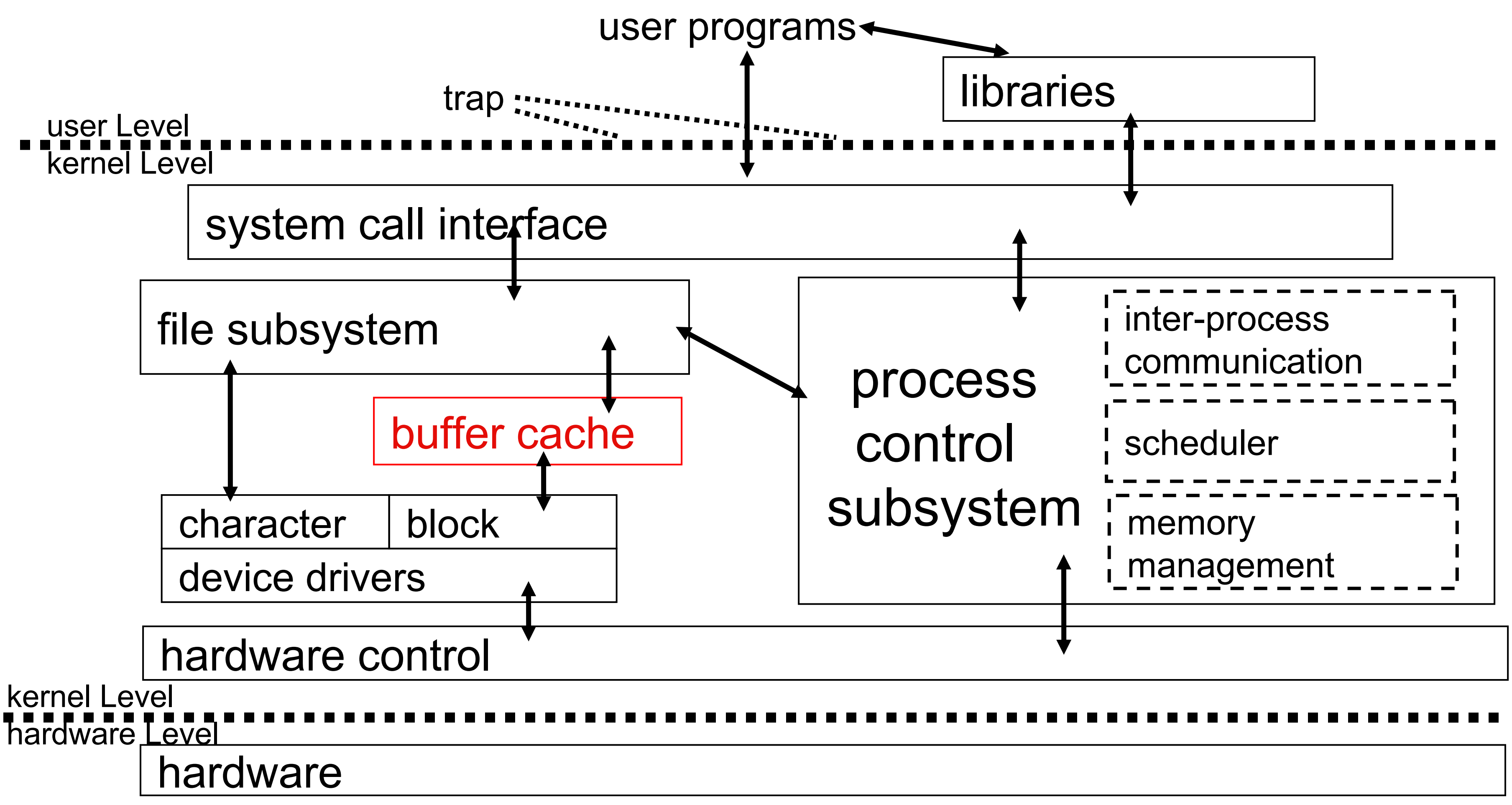
UNIX Architecture



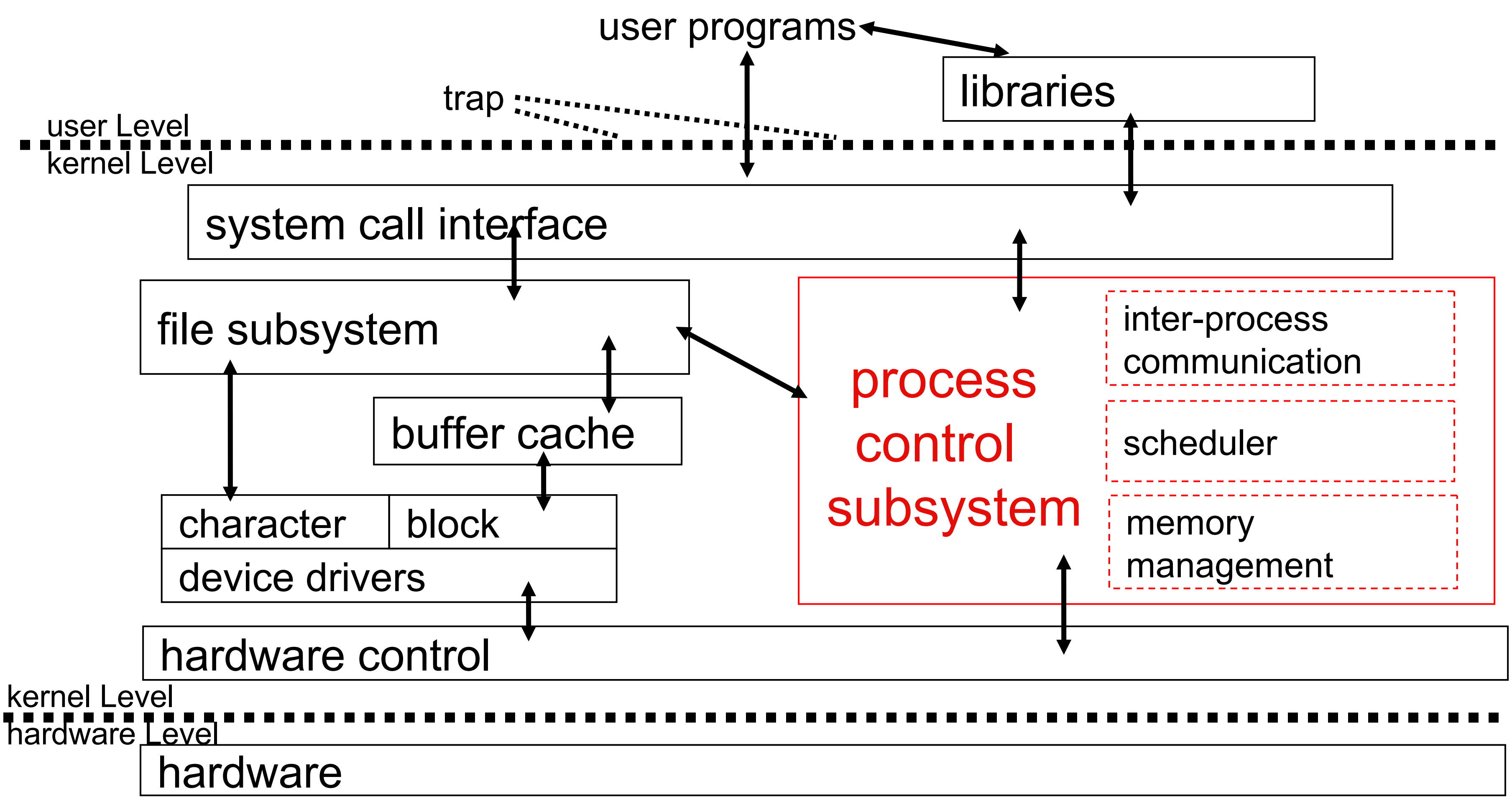
UNIX Architecture



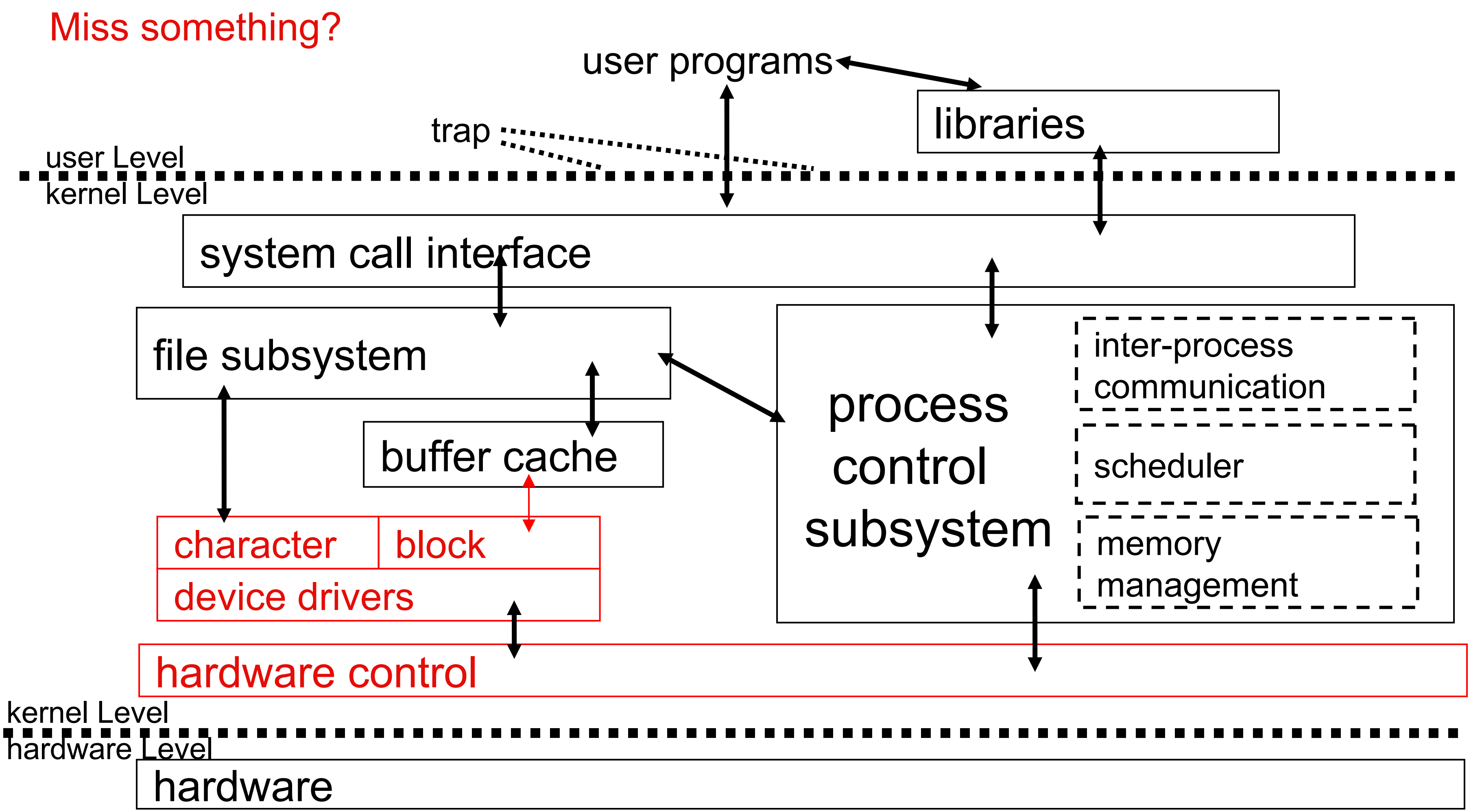
UNIX Architecture



UNIX Architecture



UNIX Architecture



MicroKernel

Outline

1. What is a microkernel?
2. Why use Microkernels
3. Mach
4. L4

What is a Microkernel?

Kernel with minimal features

- Address spaces
- Interprocess communication (IPC)
- Scheduling

What is a Microkernel?

Kernel with minimal features

- Address spaces
- Interprocess communication (IPC)
- Scheduling

Other OS features run as user-space

- servers.
- Device drivers
- Filesystem
- Pager

Why use Microkernels?

Flexibility: can implement competing versions of key OS features, like filesystem or paging, for best performance with applications.

Why use Microkernels?

Flexibility: can implement competing versions of key OS features, like filesystem or paging, for best performance with applications.

Safety: server malfunction restricted to that server (even drivers), not affecting rest of OS.

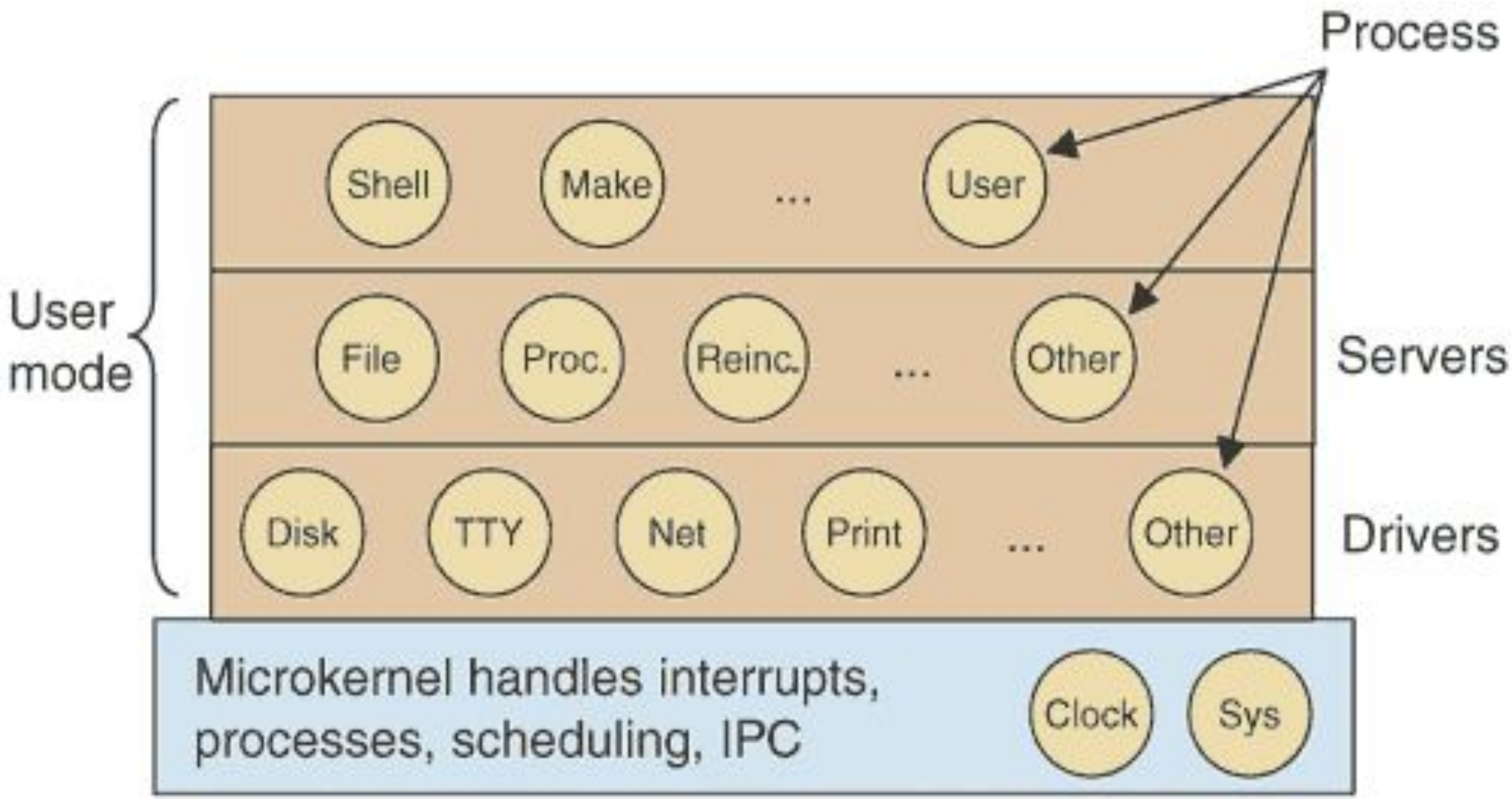
Why use Microkernels?

Flexibility: can implement competing versions of key OS features, like filesystem or paging, for best performance with applications.

Safety: server malfunction restricted to that server (even drivers), not affecting rest of OS.

Modularity: fewer interdepencies and a smaller trusted computing base (TCB).

Example Microkernel Architecture: MINIX 3



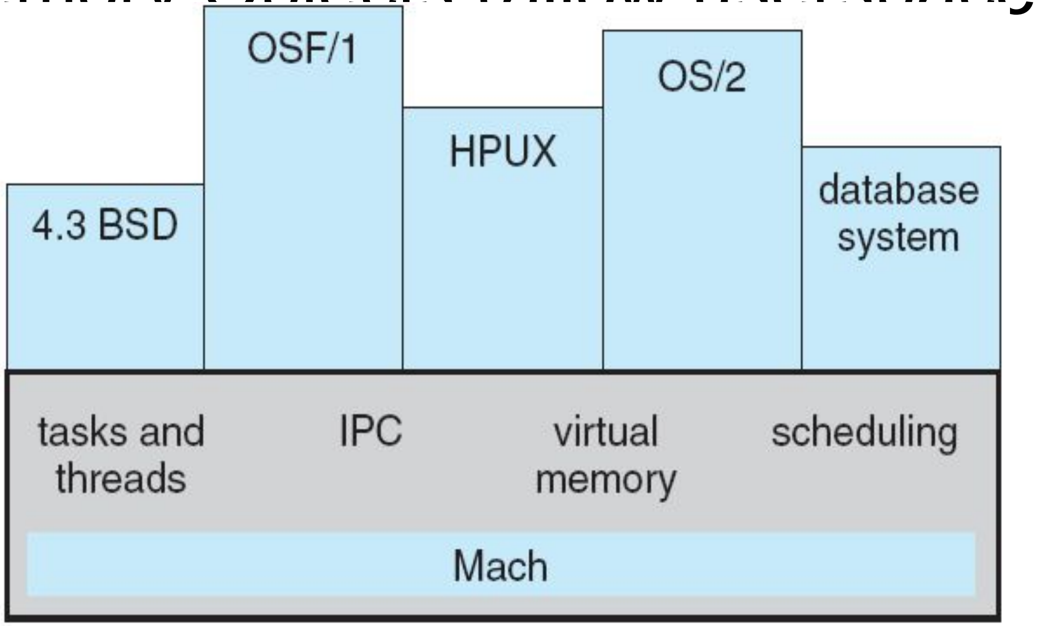
Mach

First generation microkernel.

- Runs OS personality on top of microkernel.

Core Abstractions

- Tasks and Threads (kernel provides scheduling)
- Messages (instead of system calls)
- Memory Objects (allow userspace paging)



Mach Abstractions

Task: unit of execution consisting of an address space, ports, and threads.

Thread: basic unit of execution, shares address space, ports with other threads in task.

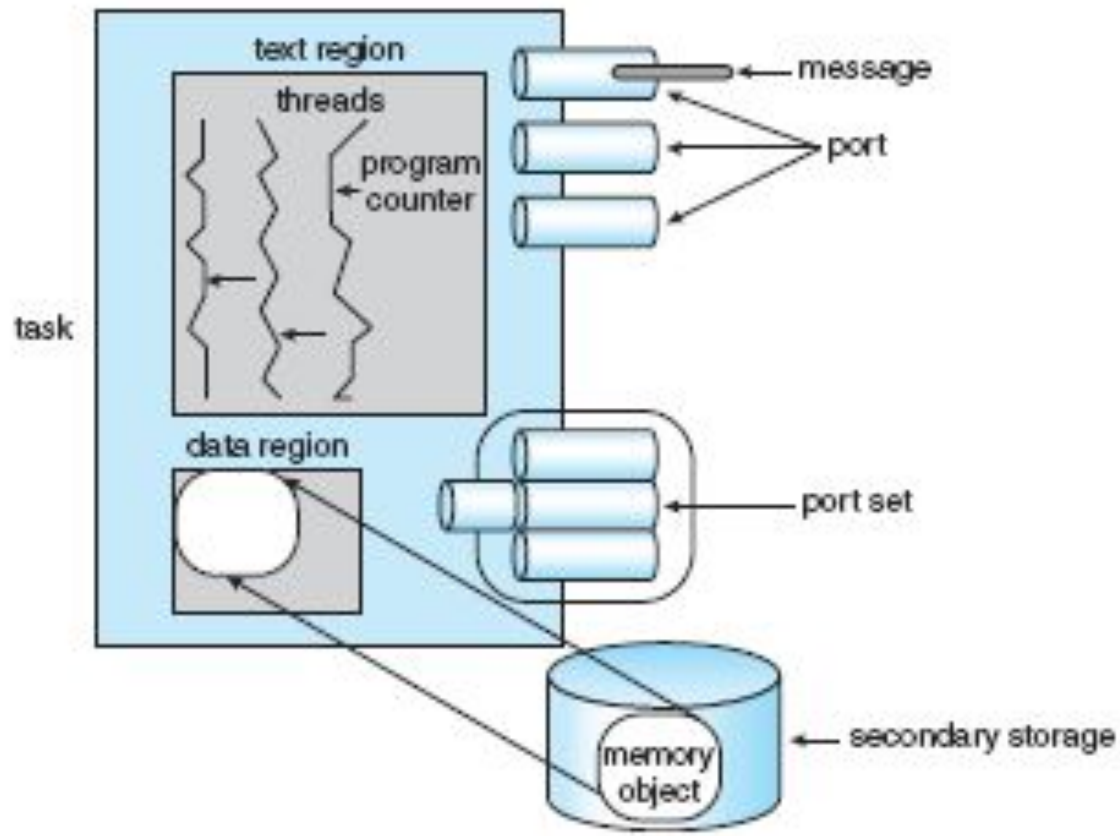
Port: communication channel used to send messages between tasks. Tasks must have correct port rights to send message to a task.

Message: basic unit of communication consisting of a typed set of data objects.

Memory Object: source of memory tasks can map into their address space; includes files and pipes.

Mach Threads and Messages

- Threads have multiple ports with different port rights
- Send messages to ports instead of system calls.
- Task must have port rights to send message to port.



Mach Performance

- System calls take 5-6X as long as UNIX.
- Message Passing
  - Uses pointers, copy-on-write, and memory mapping to avoid unnecessary copies.
  - Port rights checks are expensive.
- Paging
  - Pageout kernel thread determines system paging policy (which pages are paged out to disk.)
  - Pager servers handle actual writing.

L4 Microkernel

- Second generation microkernel.
- Smaller
  - L4 is 12KB. Compare to Mach 3 (330KB)
  - Memory management policy moved entirely to userspace.

L4 Microkernel

- Second generation microkernel.
- Smaller
  - L4 is 12KB. Compare to Mach 3 (330KB)
  - Memory management policy moved entirely to userspace.
- Faster
  - IPC is about 10X faster than Mach.
  - IPC security checks moved to user space processes if needed.

Microkernels in Use

Mach

Underlying microkernel for UNIX systems.

Examples: Mac OS X, MkLinux, NeXTStep

QNX

POSIX-compliant real-time OS for embedded sys.

Fits on a single floppy.

Underlying microkernel for Cisco IOS XR.

Symbian

Microkernel OS for cell phones.

Key Points

1. Microkernel provides minimal features
    1. Address spaces
    2. IPC
    3. Scheduling
2. Microkernel advantages
    1. Flexibility
    2. Safety
    3. Modularity
3. Early microkernels were slow, but flexible memory/disk policies can allow for superior application performance.

Topics

- THE: History of OS Architecture
- Micro Kernel
- ExoKernel
- Extensible Kernel

ExoKernel

Exokernel

An OS Architecture for

Application-Level Resource Managment

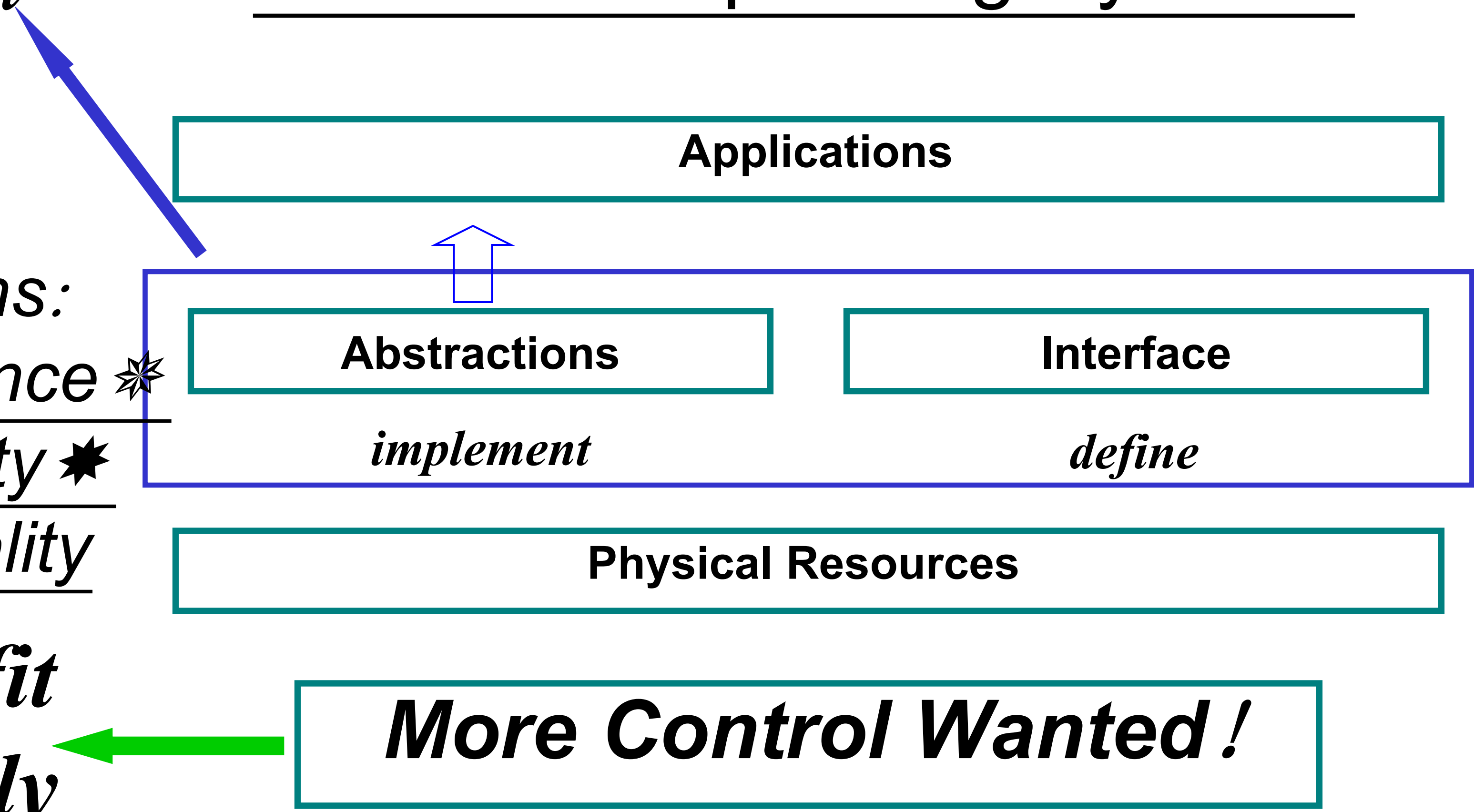
Outline

- What is the Observed Problem?
- What is the Proposed Solution?
- How is the Solution?

Problem

fixed

Traditional Operating Systems



Limit applications:

.Performance

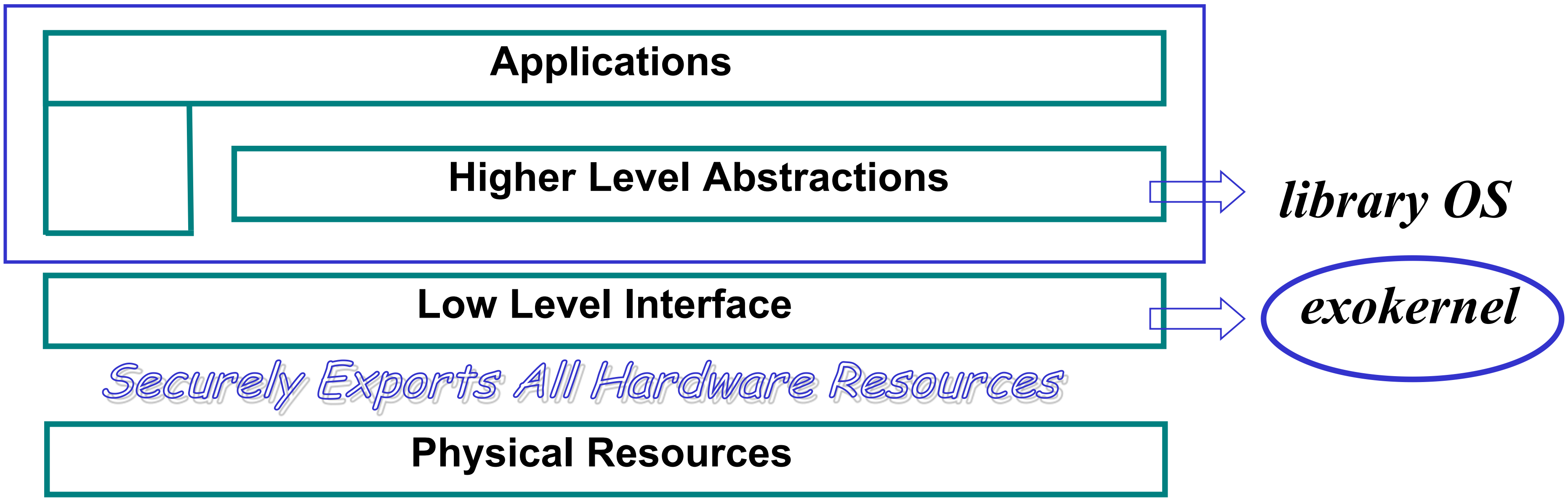
.Complexity

.Functionality

Benefit Greatly

Solution

Proposed Operating System Architecture



Application Level Resource Management

Solution – Exokernel

- Applications Know Better Than OS
- A Simple, Thin veneer:
  – Multiplex and export physical resources securely through a set of primitives
- Library OS:
  – Simpler and more specialized
  – Portability and compatibility
  – Simplified by modular design

Solution – Design

- ***<u>One</u>*** Goal
  - Give applications more freedom in managing
- ***<u>One</u>*** Way
  - Separate protection from management
- ***<u>Three</u>*** Tasks
  - Track ownership
  - Ensure protection
  - Revoke access
- ***<u>Three</u>*** Techniques
  - Secure binding
  - Visible revocation
  - Abort protocol

Solution – Design Principles

- Securely expose hardware
  - The central tenet of the architecture
  - All privileged instructions, hardware DMA capabilities, and machine resources
- Expose allocation
  - Allow to request specific physical resources
- Expose Names
  - Remove a level of indirection: Translation
- Expose Revocation
  - Allow to relinquish

Secure Bindings – Examples

- Multiplexing Physical Memory
  - Using *self-authenticating capability* and *address translation hardware*
  - *To ensure protection*: guards access by requiring to present the capability
  - *To break*: change capability and free resource
- Multiplexing the Network
  - A software support is provided by *packet filters*
  - Application code, *filters,* is downloaded into kernel

How's the Solution?

Machine	Processor	SPEC rating	MIPS
DEC2100 (12.5 MHz)	R2000	8.7 SPECint89	~ 11
DEC3100 (16.67 MHz)	R3000	11.8 SPECint89	~ 15
DEC5000/125 (25 MHz)	R3000	16.1 SPECint92	~ 25

Table 1: Experimental platforms.

How's the Solution?

***Aegis*: As an Exokernel**

System call	Description
`Yield`	Yield processor to named process
`Scall`	Synchronous protected control transfer
`Acall`	Asynchronous protected control transfer
`Alloc`	Allocation of resources (*e.g.*, physical page)
`Dealloc`	Deallocation of resources

Table 2: A subset of the Aegis system call interface.

Primitive operations	Description
`TLBwr`	Insert mapping into TLB
`FPUmod`	Enable/disable FPU
`CIDswitch`	Install context identifier
`TLBvadelete`	Delete virtual address from TLB

Table 3: A sample of Aegis's primitive operations.

How’s the Solution?

Aegis: Base Costs

Machine	OS	Procedure call	Syscall (getpid)
DEC2100	Ultrix	0.57	32.2
DEC2100	Aegis	0.56	3.2 / 4.7
DEC3100	Ultrix	0.42	33.7
DEC3100	Aegis	0.42	2.9 / 3.5
DEC5000	Ultrix	0.28	21.3
DEC5000	Aegis	0.28	1.6 / 2.3

Table 4: Time to perform null procedure and system calls. Two numbers are listed for Aegis’s system calls: the first for system calls that do not use a stack, the second for those that do. Times are in microseconds.

How's the Solution?

Aegis: Exceptions

Machine	OS	unalign	overflow	coproc	prot
DEC2100	Ultrix	n/a	208.0	n/a	238.0
DEC2100	Aegis	2.8	2.8	2.8	3.0
DEC3100	Ultrix	n/a	151.0	n/a	177.0
DEC3100	Aegis	2.1	2.1	2.1	2.3
DEC5000	Ultrix	n/a	130.0	n/a	154.0
DEC5000	Aegis	1.5	1.5	1.5	1.5

Table 5: Time to dispatch an exception in Aegis and Ultrix; times are in microseconds.

How's the Solution?

***Aegis*: providing protected control transfer as substrate for efficient IPC implementation**

OS	Machine	MHz	Transfer cost
Aegis	DEC2100	12.5MHz	2.9
Aegis	DEC3100	16.67MHz	2.2
Aegis	DEC5000	25MHz	1.4
L3	486	50MHz	9.3 (normalized)

Table 6: Time to perform a (unidirectional) protected control transfer; times are in microseconds.

L3: the fastest published result.

How's the Solution?

Aegis: using Dynamic Packet Filter

Filter	Cold Cache	Warm Cache
MPF	71.0	35.0
PATHFINDER	39.0	19.0
DPF	7.5	1.5

Table 7: Time on a DEC5000/200 to classify TCP/IP headers destined for one of ten TCP/IP filters; times are in microseconds.

MPF: a widely used packet filter engine.

PATHFINDER: fastest packet filter engine.

How’s the Solution?

Conclusion for *Aegis*

An exokernel *can* be implemented *efficiently*!

How’s the Solution?

***ExOS*: Manage OS abstractions at application level**

Focus on:

- IPC Abstractions
- Application-level Virtual Memory
- Remote Communication

How's the Solution?

ExOS: IPC Abstractions

Machine	OS	pipe	pipe'	shm	lrpc
DEC2100	Ultrix	326.0	n/a	187.0	n/a
DEC2100	ExOS	30.9	24.8	12.4	13.9
DEC3100	Ultrix	243.0	n/a	139.0	n/a
DEC3100	ExOS	22.6	18.6	9.3	10.4
DEC5000	Ultrix	199.0	n/a	118.0	n/a
DEC5000	ExOS	14.2	10.7	5.7	6.3

Table 8: Time for IPC using pipes, shared memory, and LRPC on ExOS and Ultrix; times are in microseconds. Pipe and shared memory are unidirectional, while LRPC is bidirectional.

How’s the Solution?

ExOS: Virtual Memory measured by matrix multiplication

Machine	OS	matrix
DEC2100	Ultrix	7.1
DEC2100	ExOS	7.0
DEC3100	Ultrix	5.2
DEC3100	ExOS	5.2
DEC5000	Ultrix	3.8
DEC5000	ExOS	3.7

Table 9: Time to perform a 150x150 matrix multiplication; time in seconds.

How's the Solution?

ExOS: Virtual Memory On Seven Experiments of Particular Interest

Machine	OS	dirty	prot1	prot100	unprot100	trap	appel1	appel2
DEC2100	Ultrix	n/a	51.6	175.0	175.0	240.0	383.0	335.0
DEC2100	ExOS	17.5	32.5	213.0	275.0	13.9	74.4	45.9
DEC3100	Ultrix	n/a	39.0	133.0	133.0	185.0	302.0	267.0
DEC3100	ExOS	13.1	24.4	156.0	206.0	10.1	55.0	34.0
DEC5000	Ultrix	n/a	32.0	102.0	102.0	161.0	262.0	232.0
DEC5000	ExOS	9.8	16.9	109.0	143.0	4.8	34.0	22.0

Table 10: Time to perform virtual memory operations on ExOS and Ultrix; times are in microseconds. The times for **appel1** and **appel2** are per page.

How's the Solution?

ExOS: Remote Communication

Machine	OS	Roundtrip latency
DEC5000/125	ExOS/ASH	259
DEC5000/125	ExOS	320
DEC5000/125	Ultrix	3400
DEC5000/200	Ultrix/FRPC	340

Table 11: Roundtrip latency of a 60-byte packet over Ethernet using ExOS with ASHs, ExOS without ASHs, Ultrix, and FRPC; times are in microseconds.

FRPC: fastest RPC on comparable hardware.

How’s the Solution?

***ExOS*: No Conclusion in Paper☺**

Based on the results of these experiments, can *conclude* that:

The exokernel architecture is a *viable* structure for *high-performance*, *extensible* operating systems.

Extensible Kernel

Outline

Extensibility, Safety and Performance
in the SPIN Operating System

- Introduction (SPIN, Goals, Ideas)
- Architecture (Protection model, Extension model, Core extensible services, Building services with SPIN)
- Performance (Microbenchmarks, Networked VS)
- Conclusions (Critiques, Related works)
- Discussion

SPIN

- An Extensible Operating System
- Motivation
  - Support high performance applications
  - High performance and functionality demands
  - Poorly matched by traditional operating systems

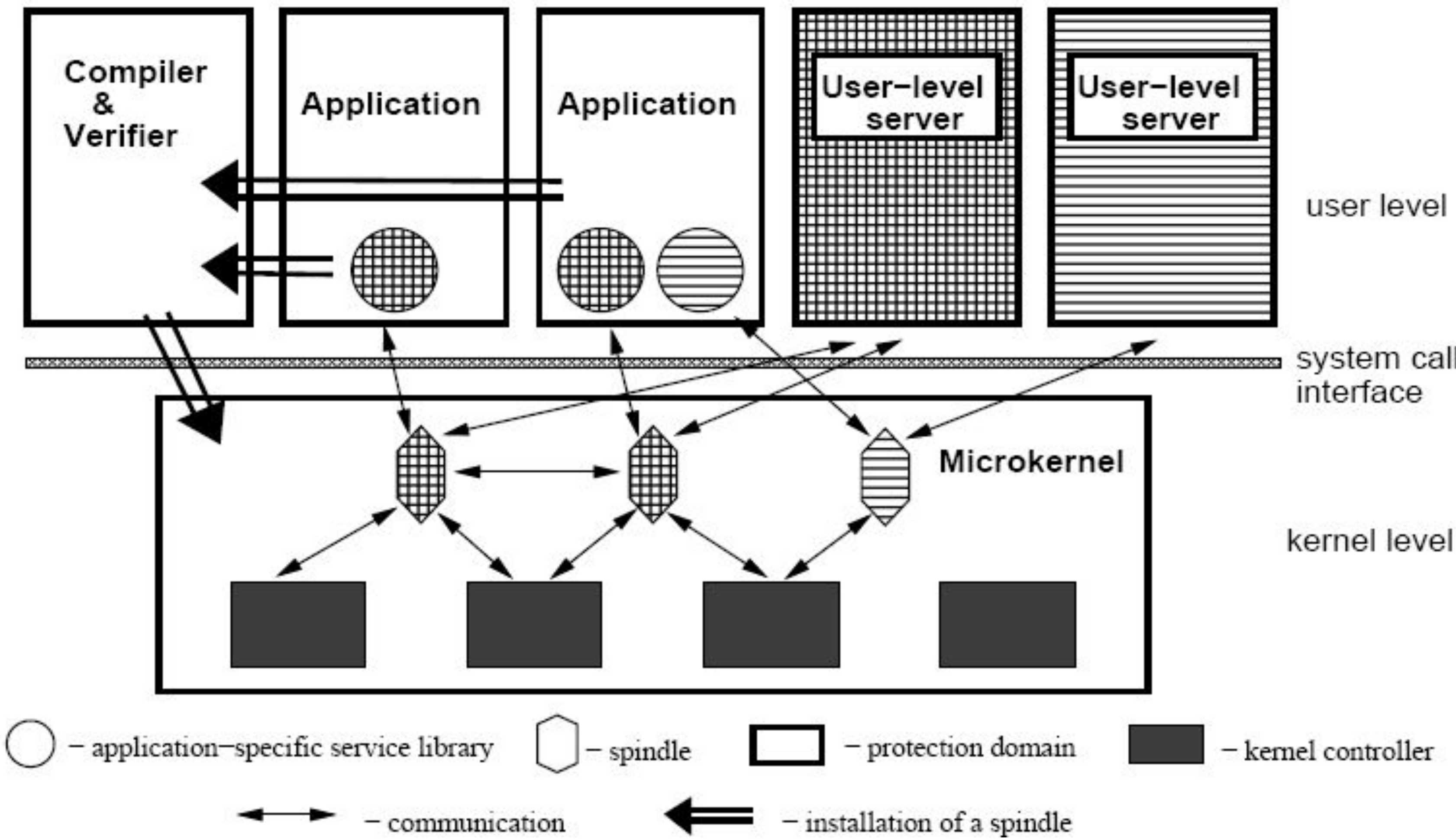
SPIN Goals

- Extensibility
  - Extensible infrastructure
  - Fine-grained access to system resources and functions
- Safety
  - Access is controlled at the same granularity
- Efficiency
  - Overhead of both protection and access is low

System Overview

- Written in **Modula-3**
  - Extensions to be easily integrated
- Using language services to provide *safe extensibility* within the kernel
- Only code that requires *low-latency access* to system services *must* be written in the system's safe extension language

SPIN Architecture



SPIN Protection Model

- **Goal**: control the set of operations that can be applied to resources

- Capability-based protection system
- Protection domains

SPIN Core Services

- *Trusted*
- Statically linked into the kernel
- Interfaces are <u>extensible</u>

Memory Services

Processor Services

Building Services with SPIN

- Kernel and core services can be used to implement more conventional operating system abstractions
  - System calls
  - Address space
  - Networking

Performance

Microbenchmarks

Networked video application

Platform

- Alpha 133MHz DEC AXP 3000/400 workstations
  - 64 MB memory, HP C2247-300 1 GB disk drive
- Networking
  - 10Mb/s Ethernet
  - ATM (FORE TCA-100 155Mb/s)
- DEC SRC Modula-3 compiler v3.3
- **Comparison systems**
  - **SPIN**
  - **DEC OSF/1 v2.1**
  - **Mach 3.0+DEC OSF/1**

Microbenchmarks

- Page faults
- Thread management (*)
- System call overhead
- Cross-address space procedure call
- Address space management
- Networking (*)

Microbenchmarks
Thread management

Kernel thread management overhead

Kernel Thread Operation	Mach 3.0 kernel	DEC OSF/1 kernel	*SPIN* extension
Create	41	332	5
Ping-Pong	71	21	29
Terminate	18	260	7

SPIN's extensible threads does not incur a performance penalty

Overhead to use an implementation of the C-Threads interface for Mach 3.0, DEC OSF/1 2.1 and SPIN from a user-level application

User Thread Operation	Mach 3.0	DEC OSF/1	*SPIN* layered	*SPIN* native
Fork	50	1131	103	20
Fork,Run	233	1164	157	64
Ping-Pong	115	233	85	85
ForkJoin	338	1026	223	110

SPIN's thread management is much faster than the others

Microbenchmarks
Networking

Round trip network RPC time

	DEC OSF/1 kernel	*SPIN* extension	*SPIN* bounded
Ethernet	840	579	510
ATM	631	332 (raw 162)	241 (raw 100)

Substantially lower latency when sending directly from the kernel

The user-to-user networking bandwidth and sender CPU utilization for Ethernet and ATM

	DEC OSF/1 kernel		*SPIN* extension	
	Bwidth	CPU %	Bwidth	CPU %
Ethernet	8.9 Mb/s	35	8.9 Mb/s	20
ATM	25 Mb/s	82	41 Mb/s	55

Less CPU time using SPIN

Networked Video Application

- Server (3 extensions)
  - Read video frames from disk
  - Send the video out over the network
  - Transform a single send into multicast to a list of clients
- Client (1 extension)
  - Decompose the image and display it directly to the screen buffer

Server utilization as a function of the number of client video streams

# streams	DEC OSF/1 kernel CPU %	*SPIN* CPU %
1	28	5
5	64	19
10	75	37
15	78	55
20	NA	72

Conclusions

- Possible to achieve *good performance* in an *extensible* operating system without compromising *safety* ☺
- Efficient mechanisms for extending services + core extensible services
- Rely on the language, compiler and runtime mechanisms

Critique

- Weaknesses
  - Modula-3
  - The problem of garbage collection
- More about SPIN
  - http://www-spin.cs.washington.edu/
  - Developed at UW for approximately <u>two years</u>
  - SPIN Web server
  - SPIN → SPIN*E*

Related Works

- Other Extensible Operating Systems

  Exokernels (MIT) Mach (CMU)

  NOW (Berkeley) Spring (Sun)

  Scout (Arizona) VINO (Harvard)

  Synthetix (OGI)

- **The Singularity Project – Microsoft**

Discussion

(Topics from the submitted questions)

1. Language
   – Modula-3 or other type safe languages, C,..?
2. SPIN <> L4, Sandboxing
3. SPIN <> Open-source OS (Linux)
4. Safety (language, core services)
5. A large number of extensions installed into the OS?
6. Security