CSCE 613: Structure, Abstractions

[1] Robert C. Daley and Jack B. Dennis, "Virtual Memory, Processes, and Sharing in **MULTICS**". Communications of the ACM, Vol(II)-5, May 1968.

[2] E. W. Dijkstra, "The Structure of the **THE**-Multiprogramming System". Communications of the ACM, Vol(2)-5, May 1968.

[3] Dennis M. Ritchie and Ken Thompson, "The UNIX Time-Sharing System". Communications of the ACM, Vol(17)-7, July 1974.

[4] Jochen Liedtke, "On μ-Kernel Construction" Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP), Copper Mountain Resort, CO, December 1995.

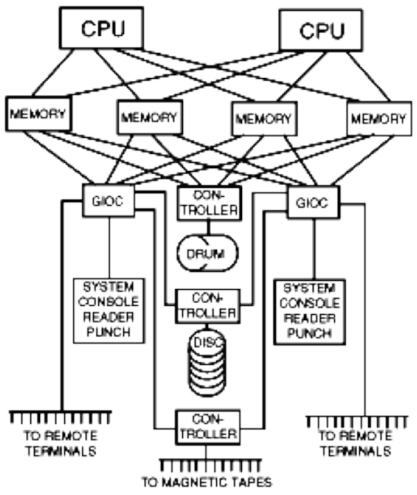
[5] M. Frans Kaashoek, Dawson R. Engler, Gregory R. Ganger, Hector M. Briceno, Russel Hunt, David Mazieres, Thomas Pinckney, Robert Grimm, John Jannotti, and Kenneth Mackenzie, "Application Performance and Flexibility on **Exokernel** Systems". Proceedings of the 16th Symposium on Operating Systems Principles (SOSP), October 1997.

[1] Multics: Background

From F.J. Corbato and V.A. Vyssotsky, "Introduction and Overview of the Multics System", 1965 Fall Joint Computer Conference.

- Multiplexed Information and Computing Service:
  - Computing system as 24/7 **utility**.
- Requirement for generality/adaptability -> programming in high-level language (PL/I).
- Requirements for Large, service-oriented computer installations:
  1. Economies of scale, with more opportunities to amortize man-power, consolidation of resource, and management overhead.
  2. Meet increasing user demand through replication of processors and memory units.
  3. Demand for high availability.
  4. Need for intelligent use of multiprogramming.
  5. Timesharing for efficient utilization and sharing of information.

Multics System Configuration (example)



- Composability / "Partitionability"
- Components are independent/ asynchronous -> easy upgrade and increase in system capacity.
- For maintenance, system can be partitioned.

Multics: Addressing

- **Segmentation** with **dual-page-size paging**. (64 words vs. 1024 words).
- Segments are visible to user, Pages are invisible.

Multics: Reasons for **Segments**

1. "The user is able to program in a two-dimensional virtual memory system. Thus, any single segment can grow (or shrink) during execution.
2. The user can, by merely specifying a starting point in a segment, operate a program implicitly without prior planning of the segments needed or of the storage requirements. For example, if an error diagnostic segment is unexpectedly called for, it is brought in automatically by the supervisor; it is never brought in unless needed.
3. The largest amount of code which must be bound (loaded) together as a solid block is a single segment.
4. Program segments appear to be the only reasonable way to permit pure procedures and data bases to be shared among several users simultaneously. Pure procedure programs, by definition, do not modify themselves."
5. Per-segment privileges, e.g. "execute-only" bit (!?)

Multics: **Paging**

Advantages of Paging:

1. The use of paged memory allows flexible techniques for dynamic storage management without the overhead of moving programs back and forth in the primary memory. This reduced overhead is important in responsive time-shared systems where there is heavy traffic between primary and secondary memories.

2. The mechanism of paging, when properly implemented, allows the operation of incompletely loaded programs; the supervisor need only retain in main memory the more active pages, thus making more effective use of high-speed storage. Whenever a reference to a missing page occurs, the supervisor must interrupt the program, fetch the missing page, and re-initiate the program without loss of information.

Multics: Design Features of the Software

- Any segment has to know another segment **only by symbolic name**. Intersegment binding occurs dynamically as needed during program execution. The mechanism operates at high efficiency after the first binding occurs.
- A segment is able to reference symbolically a location within another segment; dynamically and automatically; after binding occurs the first time, program execution is at full speed.
- It is straightforward for procedures to be **pure procedures**, capable of being shared by several users.
- Similarly, it is straightforward to write recursive procedures.
- The general conventions are such that the call, save, and return macros used to link one independently compiled procedure to another do not depend on whether or not the two procedures are in the same segment.
- Each user is provided with a private software "stack" for temporary storage within each subroutine.

Multics: Other Features

- Ability to have one process spawn other processes to run on several processors.
- Ability for data bases to be shared among simultaneously operating programs.
- Batch processing facilities included as a subset.
  - Users can attach terminals to processes.
- Scheduling of resources (and charging for their use) done by supervisor (OS)

Design Considerations in the File System

- File System as "memory system which gives the users and the supervisors alike the illusion of maintaining a private set of segments or files of information for an indefinite period of time."
- All files of information are referred to **by symbolic name** and not by address.
- "This retention is **independent of the complex of secondary storage devices** of different capacity and access."
- Issues of privacy and security, addressed through **access privileges**.
- **Synchronization of data access** (shared reads vs. exclusive writes).
- Automatic backup to recover from mishaps.

Multics: Other Considerations

- Uniform view of most **I/O devices**:
  - "A program can read from either a terminal or a disk file, or output can be sent either to a file or to a punch, a typewriter, or a printer."
- On-line documentation.
- Transition from traditional batch system: Multics runs concurrently, but independently, from GECOS batch processing system.

[2] THE - Overview

- Stated goal: "to process smoothly a continuous flow of user programs as a service …"
- Objectives:
  - Low response time for short jobs.
  - "Economic use" of peripheral devices
  - Automatic control of storage & high utilization of CPU
- Multi-user / Multi-access is not the intention.
  - -> No sharing.

THE - System Structure

- **Storage Allocation**
  - "Pages" vs. "Segments" (comparable to "frames" vs. "pages")
  - Paging eliminates fragmentation on drum.
- **Processor Allocation**
  - "Sequential Processes", one for each user job, and one for each device, plus drum and console processes.
  - "Harmonious Cooperation" ensured by explicit synchronization (semaphores).
  - Global scheduling possible in principle, i.e. independent of number of underlying processors.
- **System Hierarchy**
  - (see next slide)

THE – System Hierarchy

Level-0: Processor allocation / abstraction

Level-1: Virtual memory management / memory abstraction.

Level-2: "Message Interpreter" : operator console communicates to processes.

Level-3: Buffer management for input/output streams : device abstraction.

Level-4: User programs

Level-5: Operator.

Levels also used for proof of deadlock freedom.

THE: Semaphores

- Synchronization Primitives:
  - Semaphores
- Two uses of semaphores:
  - Mutual Exclusion
  - "Private Semaphores"

[3] UNIX: Design Considerations

- System designed to "make it easy to write, test, and run programs".
- Support for interactive use.
- Elegance of design enforced by severe resource constraints.
- Ability of System to "maintain itself".
  - Source programs always available and easily modified, so system rewrite easy to perform on-line.
- All programs should be usable with any file or device as input or output -> device considerations pushed into the kernel -> device drivers!

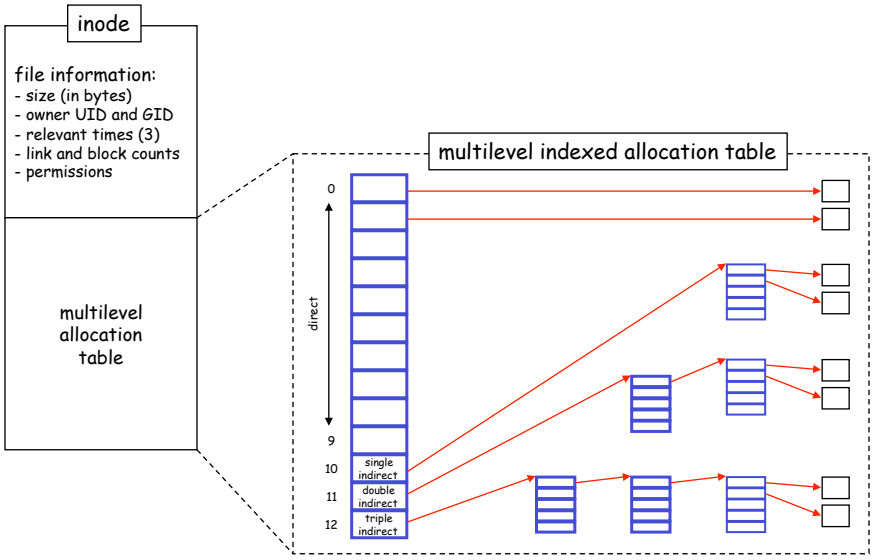
UNIX: Hardware-Software Environment

- Memory Hierarchy: 144KB core memory, one 1MB fixed-head disk, four 2.5MB moving-head disks.
- After early iterations in assembler, the 1973 version is implemented in C.

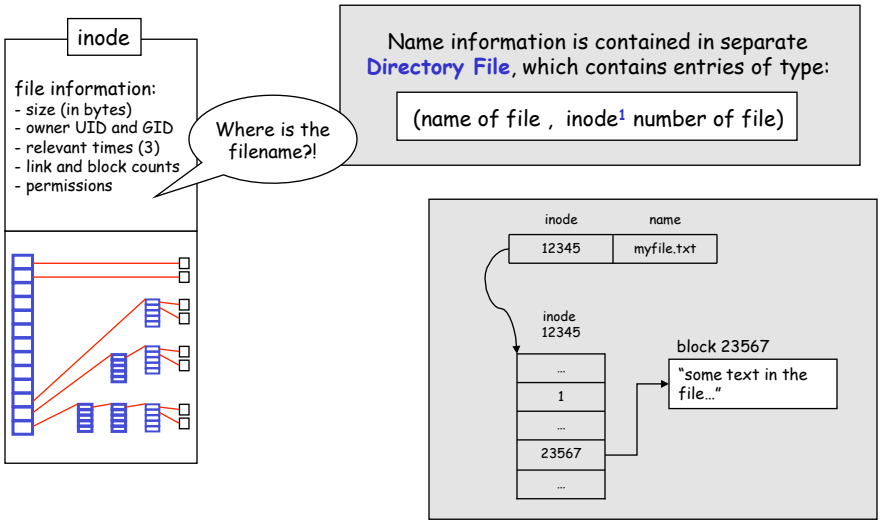
UNIX File System

- **Ordinary Files:**
  - Structure of files controlled by the programs that use them.
- **Directories:**
  - Map hierarchical name space to files in system.
- **Special Files**
  - Particular to UNIX
  - Each I/O device associated with a special file in `/dev` directory, and device accessed by reading/writing to file.
  - Benefits:
    - Uniform interface to devices
    - Can borrow file system name space to identify device
    - Can borrow file system protection mechanisms
- **Removable File Systems**
- **Protection**

Unix File System Implementation: inodes

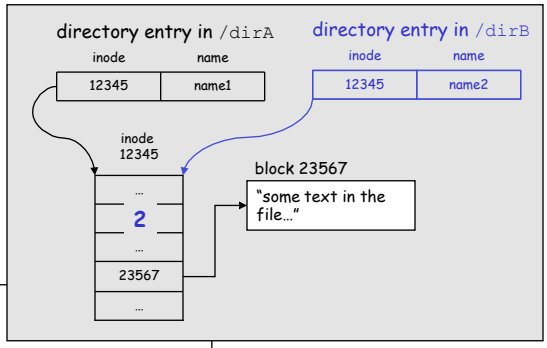


Directory Implementation



[1] More precisely: Number of block that contains inode.

Hard Links



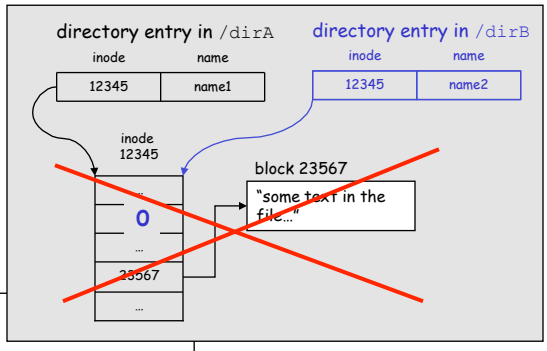
shell command
`ln /dirA/name1 /dirB/name2`

is typically implemented using the **`link`** system call:

```
#include <stdio.h>
#include<unistd.h>

if (link("/dirA/name1", "/dirB/name2") == -1)
  perror("failed to make new link in /dirB");
```

Hard Links: `unlink`




```
#include <stdio.h>
#include<unistd.h>

if (unlink("/dirA/name1") == -1)
  perror("failed to delete link in /dirA");

if (unlink("/dirB/name2") == -1)
  perror("failed to delete link in /dirB");
```

UNIX: Processes

- **Def [Image]:** An **image** is a computer execution environment.
  - core image, general register values, status of open files, current directory, etc.
- **Def [Process]:** A **process** is the execution of an image.
  - While the process is executing on behalf of a process, the image must reside in core.
- User-core part of the image:
  - program **text** segment, writable **data** segment, **stack** segment
- The **fork** system call: generates two independent copies of core image, with shared open files.
- IPC through **pipes** (make use of file I/O read/write calls)

UNIX: Traps

- Program termination upon hardware detection of program faults.
- Process termination by terminal-generated signals.
- Specialized signal handlers to either ignore or catch signals within the process.
- Example: catch non-implemented instructions and emulate in software.

[4] On µ-kernel Construction

- Basic idea: Minimize mandatory part of OS ("kernel") and "implement outside of kernel whatever is possible".
- Advantages:
  - "A clear interface enforces a modular system structure."
  - "Servers malfunction is as isolated as any other user program's malfunction."
  - "The system is more flexible and tailorable."
- Problem: performance.
- Folklore: "increased user-kernel mode and address-space switches are responsible"

µ-kernels: Functional Requirements

- Functional as opposed to Performance requirements.
- Principle of **Independence**:
  - "A programmer must be able to implement an arbitrary subsystem *S* in such a way that it cannot be disturbed or corrupted by other subsystems *S'*."
  - Then "*S* can give guarantees independent of *S'*."
- Principle of **Integrity**:
  - "There must be a way for S_1 to address S_2 and establish a communication channel that can neither be corrupted nor eavesdropped by *S'*."

µ-kernels: Address Spaces

- "At hardware level, **address space** is mapping that associates each virtual page to a physical page frame or marks it 'non-accessible'."
- Micro-kernel has to hide hardware concept of address space. (Otherwise, protection would be impossible to implement.)
- Micro-kernel abstraction of address space has to be simple and similar to hardware concept.
- Basic idea: "Support **recursive** construction of address spaces outside the kernel."

µ-kernels: Address Spaces (2)

- Basic idea: "Support recursive construction of address spaces outside the kernel."
- Given: Let Subsystem S_0 manage Address space σ_0, the physical memory.
- Kernel supports creation and maintenance of additional address spaces on top of σ_0 through three operations:
  - **Grant**: Owner of address space can grant any of its pages to another space, provided recipient agrees. Page is removed from granter's address space.
  - **Map**: The owner of an address space can map any of its pages into another address space, provided recipient agrees.
  - **Flush**: Flushed page remains accessible in flusher's address space, but is removed from all other address spaces.
- Only Grant/Map/Flush operations are retained inside the kernel.
- Memory managers and pagers are implemented on top of the kernel.

Address Spaces: Example

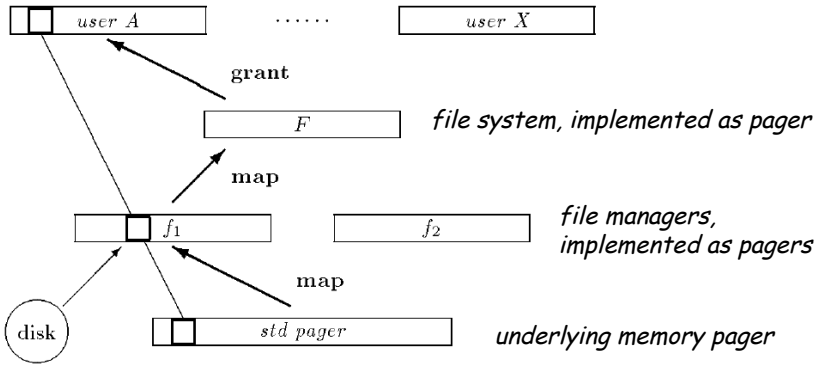


Figure 1: *A Granting Example.*

µ-kernels: I/O

- Use address spaces to abstract device ports.
- Trivial for memory-mapped I/O (e.g. PowerPC, which has pure memory mapped I/O)
- Possible for I/O ports as well.

µ-kernels: Threads and IPC

- **Threads** must be provided as abstraction by the kernel:
  - This allows to control mapping of thread to associated address space.
  - Also supports preemption, e.g.
- Cross-Address-Space communication (IPC) must be supported by kernel.
  - Possible implementation: messages between threads in kernel.
  - Other forms of communication (RPC, thread migration) can be built on top of message based IPC.
- Note: **grant** and **map** operations for address spaces rely on IPC, since they require communication between issuer and recipient.

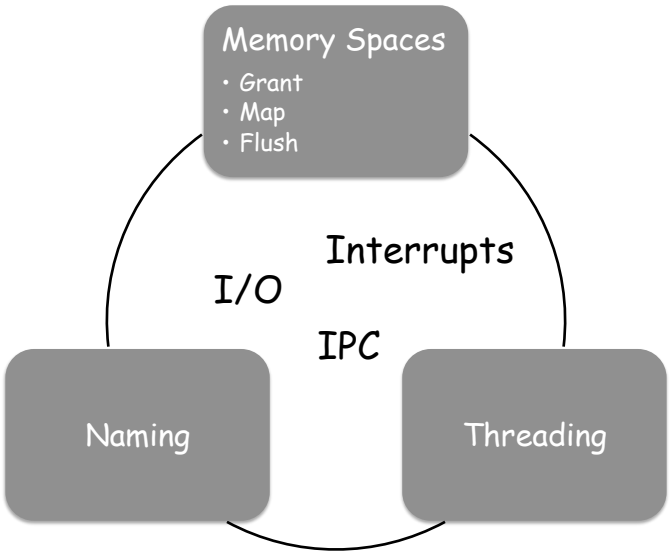
µ-kernels: Interrupts

- "Natural abstraction for hardware interrupt is the IPC message."
- Regard hardware as "a set of threads that have special thread ids and send empty messages (only consisting of sender id) to interrupt handling threads.
- Interrupt handling thread then de-multiplexes interrupt based on sender id.
- Mapping from interrupt to message is handled by kernel.
  - But kernel does not need to know interrupt semantics.
- Technical details like releasing of interrupt and invocation of `iret` instruction needs to be carefully handled.

µ-kernels: Support for Naming

- Kernel supplies identifiers for subsystems, threads, and address spaces.
- Identifiers must be unique in space and time.
- Also, kernel is able to issue identifiers in trustworthy fashion.

µ-kernels: Facilities



µ-kernels: Flexibility

- Examples of basic OS services that can be implemented on top of micro-kernel:
  - **Memory Manager.** Physical memory manager can be implemented as server. Other memory managers can be stacked on top of it.
  - **Pager.** Can be integrated with memory manager or user memory manager as server. Can be integrated or stacked with file system servers.
  - **Multimedia Resource Allocation.** Use memory allocators to request fixed allocations of resources.
  - **Device Drivers.** Implemented as threads and use memory managers.
  - **Second Level Cache and TLB.** …
  - **Remote Communication.** Implemented by communication server that translate local messages to external communication protocols. Access communication hardware through device drivers.
  - **Unix Server.** …

µ-kernels: Performance Implications

- Why are Operating Systems so slow?!
- System call overhead
  - Kernel-User Context Switch Overhead
  - Address Space Switch Overhead
- Memory

Why are OSs so Slow?

(Why Aren't Operating Systems Getting Faster As Fast As Hardware? *John Ousterhout*, 1989)

Hardware	Abbreviation	RISC/CISC	MIPS
MIPS M2000	M2000	RISC	20
DECstation 3100	DS3100	RISC	13
Sun-4/280	Sun4	RISC	9
VAX 8800	8800	CISC	6
Sun-3/75	Sun3	CISC	1.8
Microvax II	MVAX2	CISC	0.9

Table 1: Hardware Platforms

Configuration	Time (microseconds)	MIPS-Relative Speed
M2000 RISC/os 4.0	18	0.54
DS3100 Sprite	26	0.49
DS3100 Ultrix 3.1	25	0.60
8800 Ultrix 3.0	28	1.15
Sun4 SunOS 4.0	32	0.68
Sun4 Sprite	32	0.58
Sun3 Sprite	92	1.0
Sun3 SunOS 3.5	108	1.0
MVAX2 Ultrix 3.0	207	0.9

Table 2: Getpid kernel call time

Why are OSs so Slow? (2)

Hardware	Abbreviation	RISC/CISC	MIPS
MIPS M2000	M2000	RISC	20
DECstation 3100	DS3100	RISC	13
Sun-4/280	Sun4	RISC	9
VAX 8800	8800	CISC	6
Sun-3/75	Sun3	CISC	1.8
Microvax II	MVAX2	CISC	0.9

Table 1: Hardware Platforms

Configuration	Time (ms)	MIPS-Relative Speed
M2000 RISC/os 4.0	0.30	0.71
DS3100 Ultrix 3.1	0.34	0.96
DS3100 Sprite	0.51	0.65
8800 Ultrix 3.0	0.70	1.0
Sun4 SunOS 4.0	1.02	0.47
Sun4 Sprite	1.17	0.41
Sun3 SunOS 3.5	2.36	1.0
Sun3 Sprite	2.41	1.0
MVAX2 Ultrix 3.0	3.66	1.3

Table 3: Cswitch: echo one byte between processes using pipes.

Autopsy of a Kernel Call

Example: Mach `get_self_thread` call on 50MHz 486:

- 18 µsec on 486/50
- This is 900 cycles
  - Bare machine instruction to enter kernel mode: 71 cycles
  - Instruction to return to user mode: 36 cycles
  - The remaining 800 cycles are kernel overhead.
  - These could be about:
    - 500 instructions, or
    - 270 cache misses, or
    - 90 TLB misses

- Counterexample: L3 has kernel overhead of 15 cycles.
  - can grow to 57 cycles (3 TLB misses, 10 cache misses)

Kernel Calls: Reflection and Conclusion

- What is a kernel call fundamentally?

  indirect call + stack switch + set "kernel bit"

- Similarly the return operation consists of

  normal return operation + stack switch + reset "kernel bit"

- Stack switching can be hidden with appropriate support from CPU.

- Liedke's conclusion: "Kernel-user mode switches are not a serious conceptual problem but an implementation one."

Switching Address Spaces

- What happens during an address space switch?
  - Switching the cache (?)
  - Switching the page table: 1 to 10 cycles
  - TLB (!)
- Tagged TLBs (e.g. MIPS)
  - TLB entry contains address space id (ASID)
  - Switching therefore transparent
- Untagged TLBs (Pentium, PowerPC, Alpha)
  - Address switch requires **TLB flush**
  - Cost is in **TLB loads** after the flush (e.g. 900 cycles on Pentium)
- Solutions:
  - On PowerPC, use segments instead of address spaces.
  - On Pentium, the same, with some dynamic management
  - Large address spaces: Use address spaces, makes no difference. (TLB needs to be reloaded anyway to accommodate large working sets.)

Memory Overhead

- Measure: **Memory Cycle Overhead Per Instruction** (MCPI)

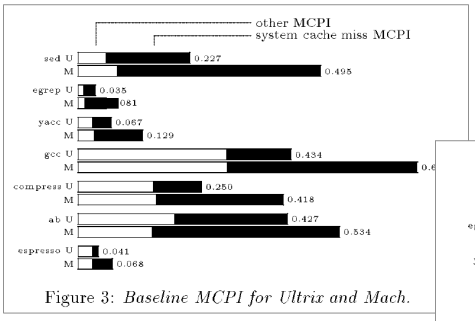


Figure 3: *Baseline MCPI for Ultrix and Mach.*

Contribute to MCPI:
- black: system i-cache or d-cache misses
- white: system write buffer stalls,
  system uncached reads,
  user i-cache and d-cache misses,
  user write buffer stalls

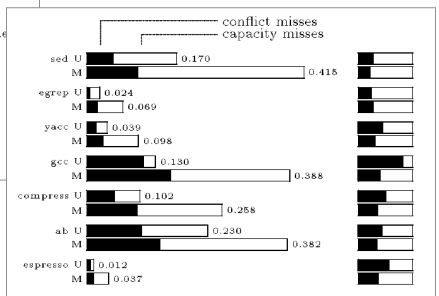
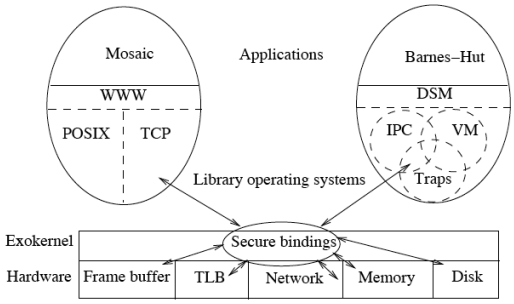


Figure 4: *MCPI Caused by Cache Misses.*

[5] ExoKernel: Motivation

- Traditionally, operating systems as interface between applications and physical resources.
- This provides abstraction and a virtual machine for applications to execute.
- **However**, this restricts flexibility of application builders:
  - application-level control over file caching?
  - application-specific virtual memory policies?
  - tuned implementations of file-systems for databases?
  - Application-level signal handling?
- **The Exokernel approach: Export** hardware resources securely (!) instead of emulating them.
- Three techniques:
  - **Secure bindings**: apps bind securely to resources and handle events.
  - **Visible resource revocation**: apps participate in resource revocation.
  - **Abort Protocol**: break bindings of uncooperative apps by force.

Exokernel: Library Operating Systems



- Abstractions should be implemented at application level.
- Library OSs use exokernel interface, and implement higher-level abstractions to best meet performance and functionality goals of applications.

Exokernel: Design

- **Challenge:** Give library operating systems maximum freedom in managing physical resources while protecting them from each other.
- To achieve this goal, an exokernel separates **protection** from **management** through a low-level interface.

- Three Tasks:
  1. tracking ownership of resources
  2. ensuring protection by guarding all resource usage or binding points
  3. revoking access to resources

Exokernel: Design Principles

- **Securely expose hardware.**
  - The kernel should provide secure low-level primitives that allow all hardware resources to be accessed as directly as possible.
- **Expose allocation.**
  - Allow library operating systems to request specific physical resources.
- **Expose Names.**
  - Export physical names.
  - Physical names are efficient, since they remove a level of indirection otherwise required to translate between virtual and physical names.
- **Expose Revocation.**
  - Utilize a visible resource revocation protocol so that well-behaved library operating systems can perform effective application-level resource management.

Exokernel: What about Policy?!

- An exokernel hands over resource policy decisions to library operating systems.
- However:
  - An exokernel must include policy to **arbitrate between competing library operating systems**: it must determine the absolute importance of different applications, their share of resources, *etc.*
  - While an exokernel cedes **management** of resources over to library operating systems, it controls the **allocation** and **revocation** of these resources.
  - By deciding which allocation requests to grant and from which applications to revoke resources, an exokernel can enforce traditional partitioning strategies, such as quotas or reservation schemes.

Exokernel: Secure Bindings

- A **secure binding** is a protection mechanism that decouples authorization from the actual use of a resource.
- Secure bindings improve performance in two ways.
  1. The protection checks involved in enforcing a secure binding are expressed in terms of simple operations that the kernel (or hardware) can implement quickly.
  2. A secure binding performs authorization only at bind time, which allows management to be decoupled from protection.
- "Simply put, a secure binding allows the kernel to protect resources without understanding them."
- Basic techniques for secure bindings:
  1. HW support: ownership checks may be done in HW
  2. SW caching: e.g. software TLBs
  3. Downloading of code into kernel: allow an application thread of control to be immediately executed on kernel events.

Example – Multiplexing Physical Memory

- When a lib OS allocates a physical memory page, the exokernel creates a secure binding for that page by recording the **owner** and the **read and write capabilities** specified by the library operating system.
- When the processor contains a **TLB**, and the exokernel must check memory capabilities when a lib OS attempts to enter a new virtual-to-physical mapping.
- To improve lib OS performance by reducing the number times secure bindings must be established, an exokernel may cache virtual-to-physical mappings in a **large software TLB**.
- If the underlying hardware defines a **page-table interface**, then an exokernel must **guard the page table** instead of the TLB.

Exokernel: Multiplexing the Network

- How much protocol-specific knowledge to we need?
- How to identify intended recipient?

- Packet filters: Implementation of secure binding where code – packet filters – are downloaded into the kernel.

- How to ensure that packet filter does not "lie"? (i.e., claim that a packet belongs to its application.)

- Sharing the network interface for **outgoing messages** is easy: Messages are simply copied from application space into a transmit buffer.
- With appropriate hardware support, transmission buffers can be mapped into application space just as easily as physical memory pages.

Downloading Code

- Downloading code into the kernel has two main performance advantages:
  1. Elimination of kernel crossings.
  2. Code can be executed when the application is not scheduled.

- **Application-specific Safe Handlers (ASH)** can be downloaded into the kernel to participate in message processing.
  - An ASH is associated with a packet filter and runs on packet reception.
  - an ASH can initiate a message.
  - Using this feature, roundtrip latency can be greatly reduced, since replies can be transmitted on the spot instead of being deferred until the application is scheduled.

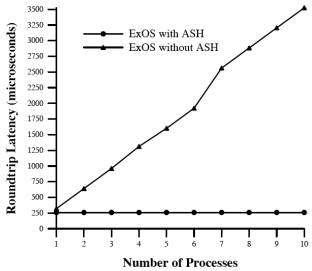


Figure 2: Average roundtrip latency with increasing number of active processes on receiver.