

Hardware Support for Operating System

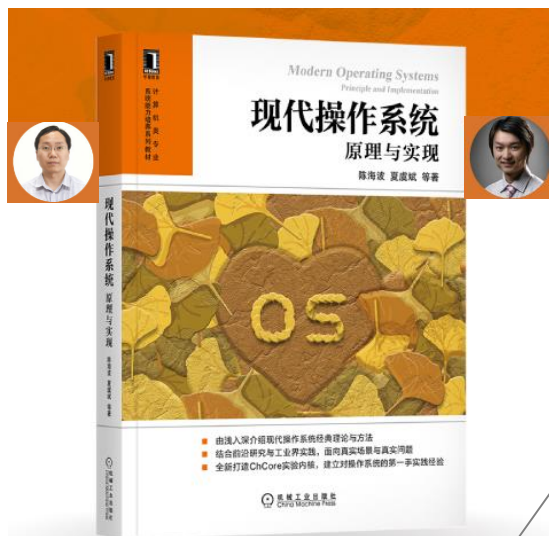
Dong Du

IPADS, Shanghai Jiao Tong University



Hardware and OS

• The preface of 《Modern Operating Systems》



为什么又要写一本操作系统的书

操作系统是现代计算平台的基础与核心支撑系统，负责管理硬件资源、控制程序运行、改善人机交互以及为应用软件提供运行环境等。长期以来，我国信息产业处于“缺芯少魂”的状态，作为信息产业之“魂”的操作系统是释放硬件能力、构筑应用生态的基础，也是关键的“卡脖子”技术之一。当前，以华为海思麒麟与鲲鹏处理器、银河飞腾处理器等为代表的 ARM 平台在智能终端和服务器等应用场景的崛起，以及以开源为特色的 RISC-V 指令集架构的出现与其生态的蓬勃发展，逐步改变了 x86 处理器在我国一统天下的局面。因此，需要结合 ARM 等指令集架构与体系结构来构筑新的操作系统或深入优化现有操作系统，从而充分发挥硬件资源的能力并给用户提供更流畅的体验。

操作系统自 20 世纪 50 年代诞生以来，经历了从专用操作系统（每个主机与应用场景均

“操作系统...负责管理硬件资源...”

Hardware and OS

- Operating system functionality fundamentally **depends upon** the architectural features of the computer
- Architectural support can greatly simplify or complicate OS tasks
 - Early PC operating systems (DOS, MacOS) lacked virtual memory in part because the architecture did not support it

Review: Hardware Support

- **Security** **Lect-10: Arch for Security**
 - Intel SGX, AMD SEV, ARM Trustzone, MPK, MPX, CET, TDX, PMP, ...
- **Virtualization** **Lect-06: Virtualization**
 - EPT/NPT, VT-d, Root/Non-Root modes, IOMMU, ...
- **Others**
 - HTM, ...

Hardware Support for OS

- Protection (kernel/user mode)
- Protected instructions
- Memory protection
- System calls
- Interrupts and exceptions
- Timer (clock)
- I/O control and operation
- Synchronization (atomic instructions)

How did we get here?

Let's start at the very beginning

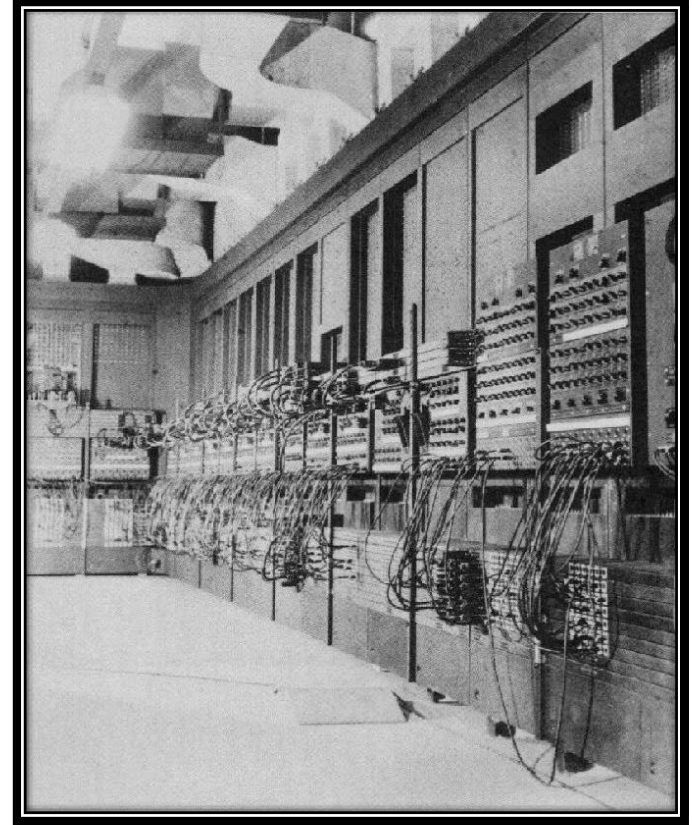


History of Operating Systems

- **Phase 1: Hardware expensive, humans cheap**
 - *User at console: single-user systems*
 - *Batching systems*
 - *Multi-programming systems*

Hand Programmed Machines (1945-1955)

- Single user systems
- OS = loader + libraries
- Problem: low utilization of expensive components



Batch processing (1955-1965)

- OS = loader + sequencer + output processor

Input

Card
Reader

Computer

User Data

User Program

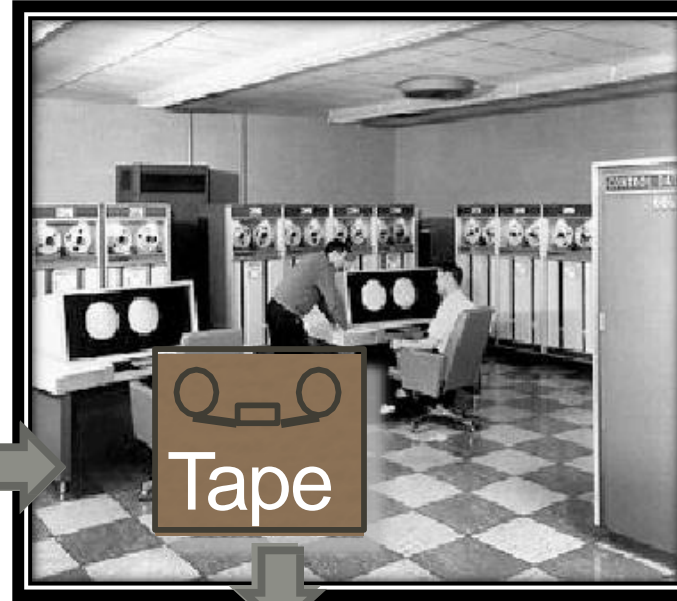
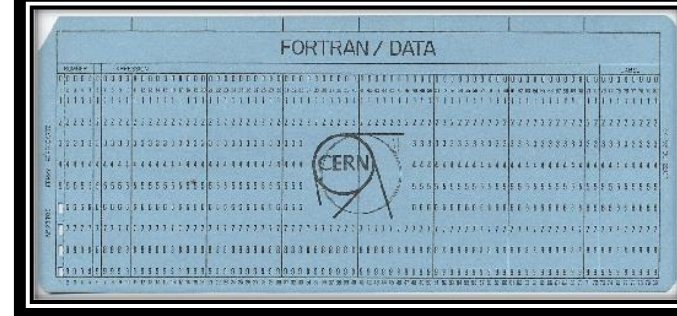
“System Software”

Operating System

Tape

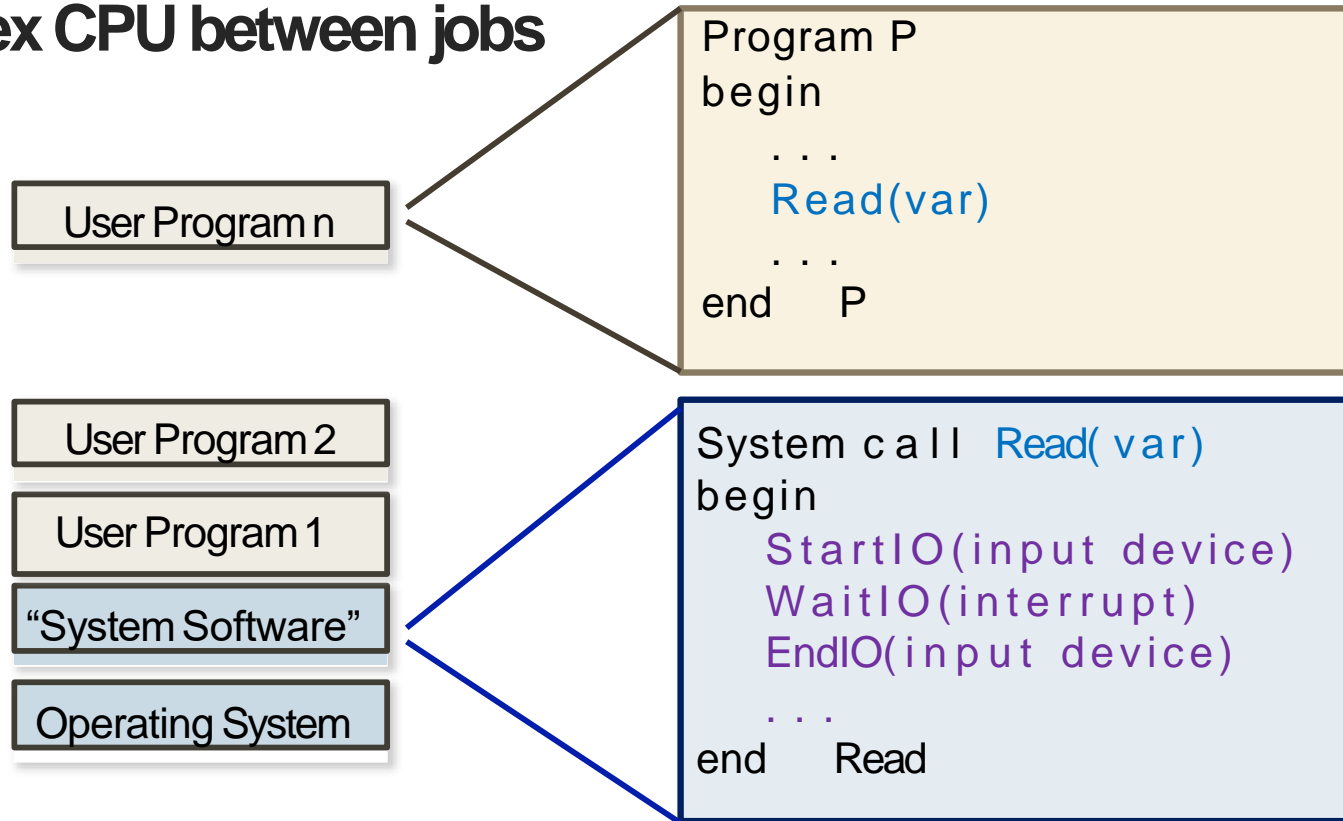
Tape

Printer
Output



Multiprogramming (1965-1980)

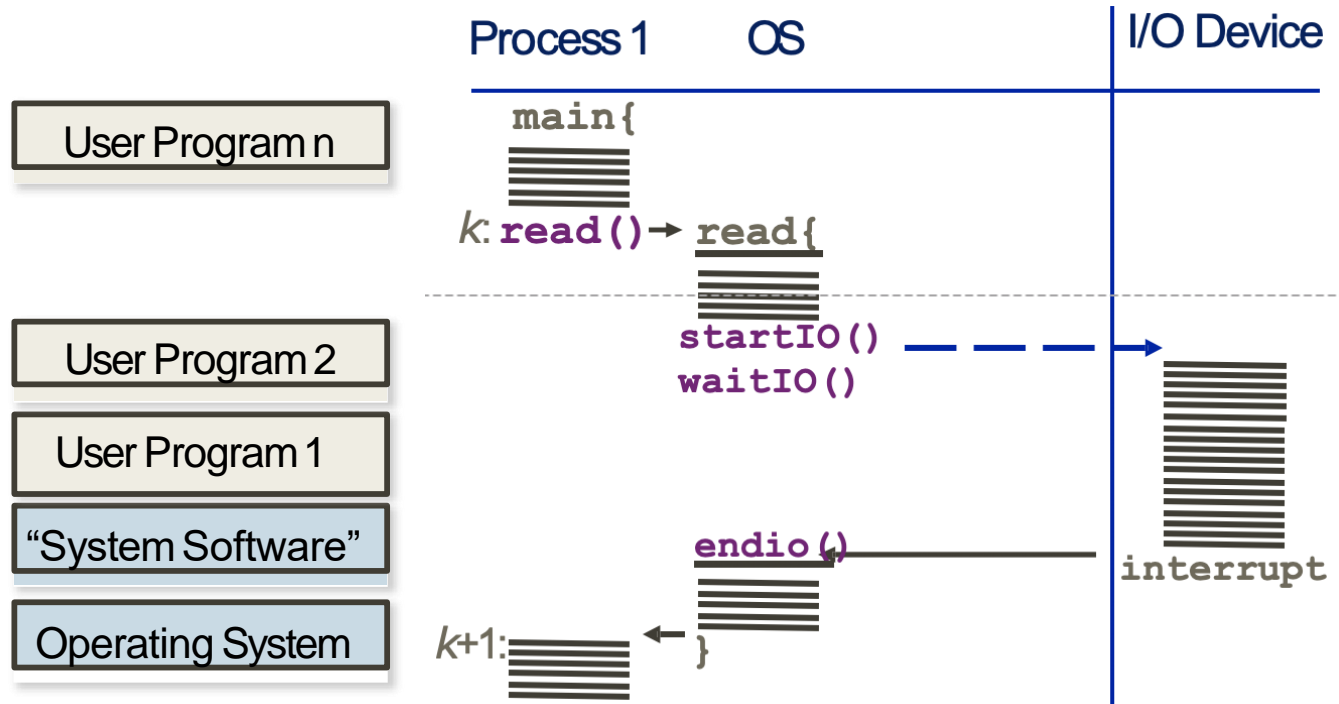
- Keep several jobs in memory
- Multiplex CPU between jobs



Multiprogramming (1965-1980)

✓ interrupt & I/O

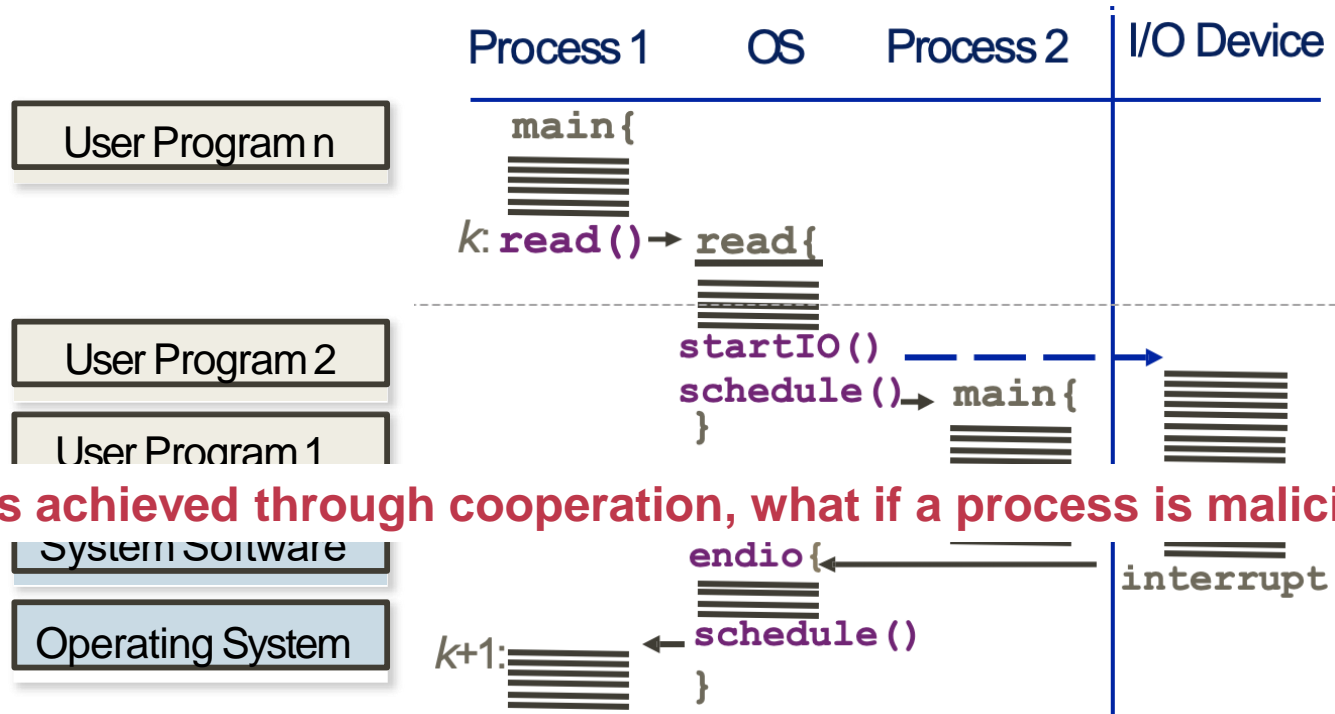
- Keep several jobs in memory
- Multiplex CPU between jobs



Multiprogramming (1965-1980)

- Keep several jobs in memory
- Multiplex CPU between jobs

- ✓ interrupt & I/O
- ✓ Mem protection
- ✓ OS protection



Multi-tasking is achieved through cooperation, what if a process is malicious?

History of Operating Systems

- **Phase 1: Hardware expensive, humans cheap**
 - *User at console: single-user systems*
 - *Batching systems*
 - *Multi-programming systems*
- **Phase 2: Hardware cheap humans expensive**
 - *Timesharing for multiple-tasks*

Timesharing (1970-)

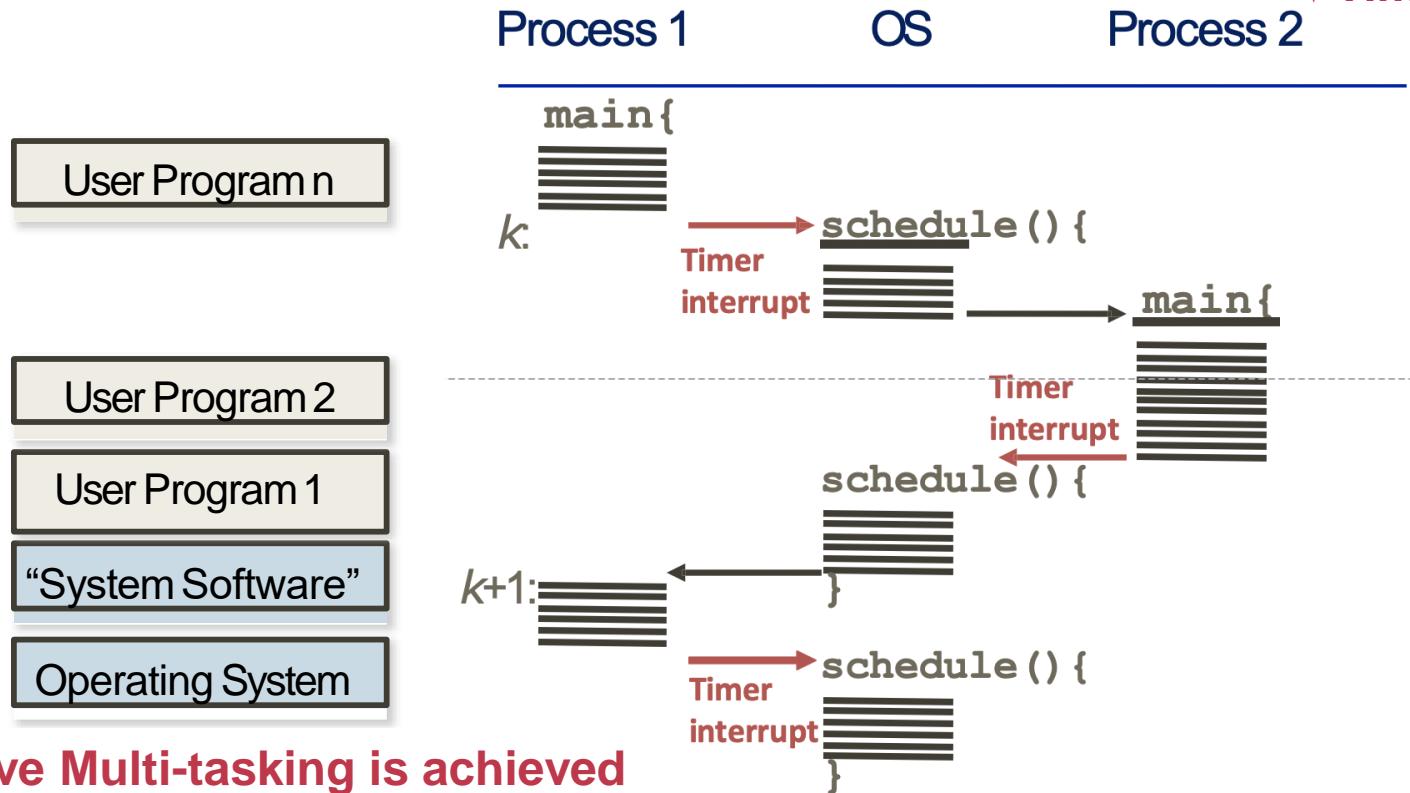
- Timer interrupt used to multiplex CPU between jobs

✓ interrupt & I/O

✓ Mem protection

✓ OS protection

✓ **Timer**



Preemptive Multi-tasking is achieved

History of Operating Systems

- **Phase 1: Hardware expensive, humans cheap**
 - *User at console: single-user systems*
 - *Batching systems*
 - *Multi-programming systems*
- **Phase 2: Hardware cheap humans expensive**
 - *Timesharing for multiple-tasks*
- **Phase 3: H/W very cheap humans very expensive**
 - *Personal computing: One system per user*
 - *Distributed computing: many systems per user*
 - *Ubiquitous computing: lots of systems per users*

Hardware Support for OS

How did we get here?

- Protection (kernel/user mode)
 - Protected instructions
 - Memory protection
 - System calls
 - Interrupts and exceptions
 - Timer (clock)
 - I/O control and operation
 - Synchronization (atomic instructions)
- Protection!*

OS Protection: Protected Instructions

A subset of instructions of every CPU is restricted to use only by the OS

- ◆ Known as protected (privileged) instructions

Only the operating system can

- ◆ Directly access I/O devices (disks/SSDs, NICs, etc.)
 - » Security, fairness (why?)
- ◆ Manipulate memory management state
 - » Page table pointers (CR3, sptbr), page protection, TLB management, etc.
- ◆ Manipulate protected control registers
 - » Kernel mode, interrupt level
- ◆ Halt instruction

OS Protection

How do we know if we can execute a protected instruction?

- ♦ Architecture must support (at least) two modes of operation: **kernel** mode and **user** mode
 - » VAX, x86 support four modes; earlier archs (Multics) even more
 - » Why? **Privilege modes** [\[edit \]](#)

The VAX has four hardware implemented privilege modes:

No.	Mode	VMS Usage	Notes
0	Kernel	OS Kernel	Highest Privilege Level
1	Executive	File System	
2	Supervisor	Shell (DCL)	
3	User	Normal Programs	Lowest Privilege Level

Protect the OS from itself (software engineering)

OS Protection

How do we know if we can execute a protected instruction?

- ◆ Architecture must support (at least) two modes of operation: **kernel** mode and **user** mode
 - » VAX, x86 support four modes; earlier archs (Multics) even more
 - » Why? Protect the OS from itself (software engineering)
- ◆ Mode is indicated by a status bit in a protected control register
- ◆ User programs execute in user mode
- ◆ OS executes in kernel mode (OS == “kernel”)

Protected instructions only execute in kernel mode

- ◆ CPU checks mode bit when protected instruction executes
- ◆ Setting mode bit must be a protected instruction

Memory Protection

OS must be able to protect programs from each other

OS must protect itself from user programs

May or may not protect user programs from OS E.g., SGX


Memory management hardware provides memory protection mechanisms

- ◆ Base and limit registers
- ◆ Page table pointers, page protection, TLB
- ◆ Virtual memory
- ◆ Segmentation

Manipulating memory management hardware uses protected (privileged) operations

Hardware Support for OS

How did we get here?

- Protection (kernel/user mode)
 - Protected instructions
 - Memory protection
 - System calls
 - Interrupts and exceptions
 - Timer (clock)
 - I/O control and operation
 - Synchronization (atomic instructions)
- 
- Protection*
- Events!*

Events

An event is an “unnatural” change in control flow

- ◆ Events immediately stop current execution
- ◆ Changes mode, context (machine state), or both

The kernel defines a handler for each event type

- ◆ Event handlers always execute in kernel mode
- ◆ The specific types of events are defined by the machine
 - ◆ Load/store fault, instruction fault, syscall, ...

Once the system is booted, all entries to the kernel occur as the result of an event

The operating system is one big event handler

Categorizing Events

Two kinds of events, **interrupts** and **exceptions**

Exceptions are caused by executing instructions

- ◆ CPU requires software intervention to handle a fault or trap

Interrupts are caused by an external event

- ◆ Device finishes I/O, timer expires, etc.

Two reasons for events, **unexpected** and **deliberate**

Unexpected events are, well, unexpected

- ◆ **What is an example?**

Deliberate events are scheduled by OS or application

- ◆ **Why would this be useful?**

▶ Categorizing Events (2)

- ◆ Terms may be used slightly differently by various OSes, CPU architectures...

	Unexpected	Deliberate
Exceptions (sync)	fault	syscall trap
Interrupts (async)	interrupt	software interrupt

- ◆ Software interrupt – a.k.a. async system trap (AST), async or deferred procedure call (APC or DPC)

Faults

Hardware detects and reports “exceptional” conditions

- ◆ **Page fault**, unaligned access, divide by zero

Upon exception, hardware “faults” (verb)

- ◆ Must save state (PC, regs, mode, etc.) so that the faulting process can be restarted

Modern OSes use page faults for many functions

- ◆ Debugging, distributed shared memory, GC, copy-on-write

Are fault exceptions necessary?

Fault exceptions are a performance optimization

- ◆ Could detect faults by inserting extra instructions into code (at a significant performance penalty)

Handling Faults

“Fixing” the exceptional condition and returning to the faulting context

- ◆ Page faults cause the OS to place the missing page into memory
- ◆ Fault handler resets PC of faulting context to re-execute instruction that caused the page fault

Notifying the process

- ◆ Fault handler changes the saved context to transfer control to a user-mode handler on return from fault
- ◆ Handler must be registered with OS
- ◆ Unix **signals** or NT **user-mode Async Procedure Calls (APCs)**
 - » SIGALRM, SIGHUP, SIGTERM, SIGSEGV, etc.

Handling Faults (2)

The kernel may handle unrecoverable faults by **killing the user process**

- ◆ Program fault with no registered handler
- ◆ Halt process, write process state to file, destroy process
- ◆ In Unix, the default action for many signals (e.g., SIGSEGV)

What about faults in the kernel?



Your PC ran into a problem and needs to restart. We're just collecting some error info, and then we'll restart for you. (0% complete)

If you'd like to know more, you can search online later for this error: PDP_DETECTED_FATAL_ERROR

Handling Faults (2)

The kernel may handle unrecoverable faults by killing the user process

- ◆ Program fault with no registered handler
- ◆ Halt process, write process state to file, destroy process
- ◆ In Unix, the default action for many signals (e.g., SIGSEGV)

What about faults in the kernel?

- ◆ Dereference NULL, divide by zero, undefined instruction
- ◆ These faults considered fatal, operating system crashes
- ◆ **Unix panic**, **Windows “Blue screen of death”**

System Calls

For a user program to do something “privileged” (e.g., I/O) it must call an OS procedure

- ◆ Some old term:
 - ◆ crossing the protection boundary, or
 - ◆ protected procedure call

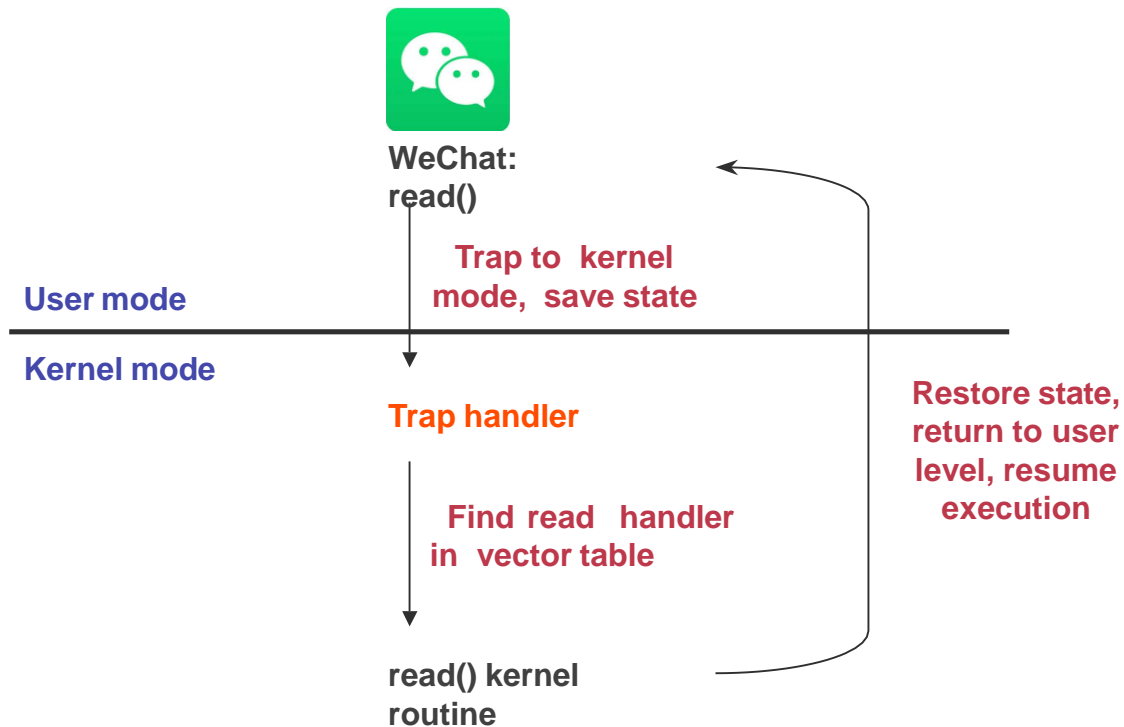
Arch provides a *system call* instruction that:

- ◆ Causes an exception, which vectors to a kernel handler
- ◆ Passes a parameter determining the system routine to call
- ◆ Saves caller state (PC, regs, mode) so it can be restored
 - » Why save mode?
- ◆ Returning from system call restores this state

Arch must permit OS to:

- ◆ Verify input parameters (e.g., valid addresses for buffers)
- ◆ Restore saved state, reset mode, resume execution

System Call



System Call Questions

What would happen if kernel did not save state?

What if the kernel executes a system call?

How to reference kernel objects as arguments or results to/from system calls?

- ◆ A naming issue
- ◆ Use integer object handles or descriptors
 - » E.g., Unix file descriptors
- ◆ Also called capabilities (more later)
- ◆ Why not use kernel addresses to name kernel objects?

Interrupts

Interrupts


- ◆ Precise interrupts – CPU transfers control only on instruction boundaries
- ◆ Imprecise interrupts – CPU transfers control in the middle of instruction execution
- ◆ OS designers like precise interrupts, CPU designers like imprecise interrupts
 - » Why?

Interrupts signal asynchronous events

- ◆ Timer, I/O, etc.

Hardware Support for OS

How did we get here?

- Protection (kernel/user mode)
 - Protected instructions
 - Memory protection
 - System calls
 - Interrupts and exceptions
 - Timer (clock) *Timesharing with preemptive!*
 - I/O control and operation *Managing I/O!*
 - Synchronization (atomic instructions)
- 
- Protection*
- Events*

Timer

The timer is critical for an operating system

It is the fallback mechanism by which the OS **reclaims control over the machine**

- ◆ Timer is set to generate an interrupt after a period of time
 - » Setting timer is a privileged instruction
- ◆ When timer expires, generates an interrupt

Prevents infinite loops

- ◆ OS can always regain control from erroneous or malicious programs that try to hog CPU

Also used for time-based functions (e.g., *sleep()*)

I/O Control

I/O issues

- ◆ Initiating an I/O
- ◆ Completing an I/O

Initiating an I/O

- ◆ Special instructions
- ◆ Memory-mapped I/O
 - » Device registers mapped into address space
 - » Writing to address sends data to I/O device

I/O Completion

Interrupts are the basis for asynchronous I/O

- ◆ OS initiates I/O
- ◆ Device operates independently of rest of machine
- ◆ Device sends an interrupt signal to CPU when done
- ◆ OS maintains a vector table containing a list of addresses of kernel routines to handle various events
- ◆ CPU looks up kernel address indexed by interrupt number, context switches to routine

Synchronization

Interrupts cause difficult problems

- ◆ An interrupt can occur at any time
- ◆ A handler can execute that interferes with code that was interrupted

OS must be able to synchronize concurrent execution

Need to guarantee that short instruction sequences execute atomically

- ◆ Disable interrupts – turn off interrupts before sequence, execute sequence, turn interrupts back on
- ◆ Special atomic instructions – read/modify/write a memory address, test and conditionally set a bit based upon previous value

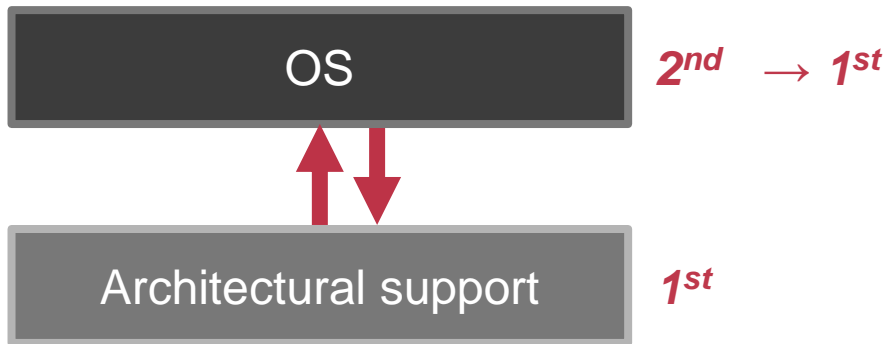
Part-2

Can we go further?

New Hardware Extensions Designed for OS

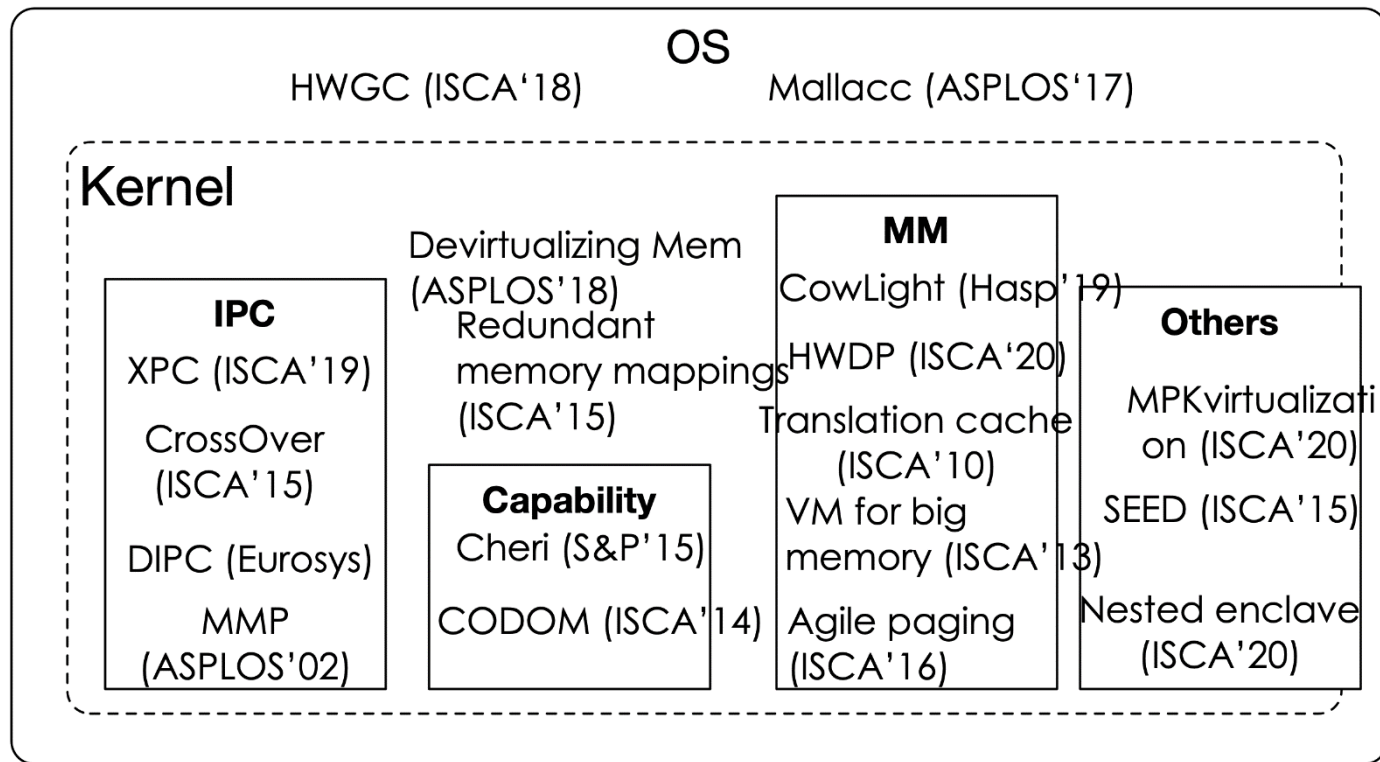
Architecture and OS

- The old fashion **has many limitations**
 - Designing OS for Hardware
- The new fashion
 - Hardware extensions to optimize OS design
 - Hardware-Software co-design



The Big Picture

- Of course it is incomplete



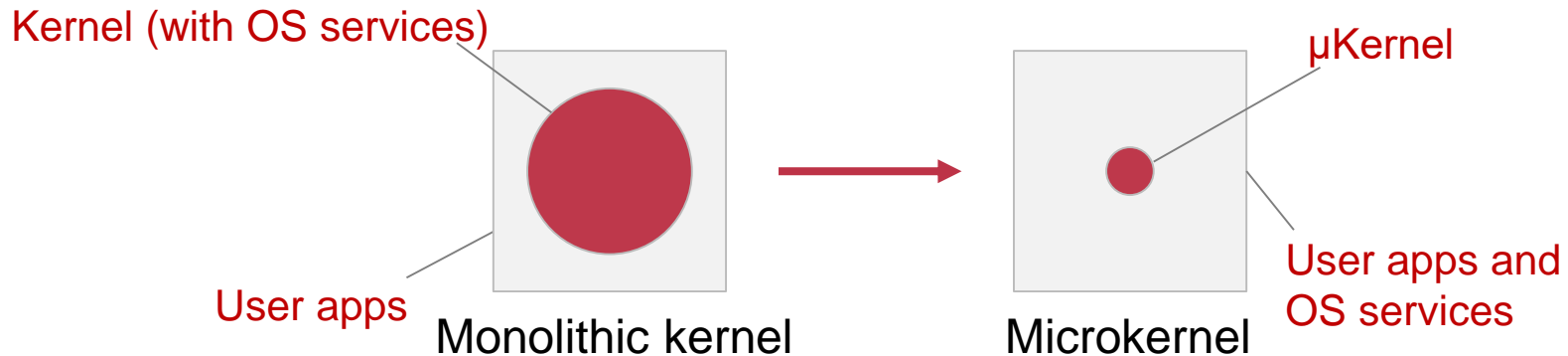
XPC

ARCHITECTURAL SUPPORT FOR INTER-PROCESS COMMUNICATION

XPC: Architectural Support for Secure and Efficient Cross Process Call. Dong Du, Zhichao Hua, Yubin Xia, Binyu Zang, Haibo Chen. (ISCA'2019)

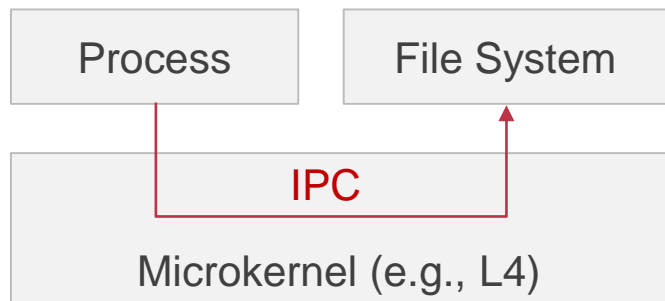
Review: Microkernel

- Cases: Mach, L4, MINIX, Google Zircon
- Less in the core
 - Inter-Process Communication (IPC)
- Put the major parts in user space
 - Drivers, file system, network, memory manager



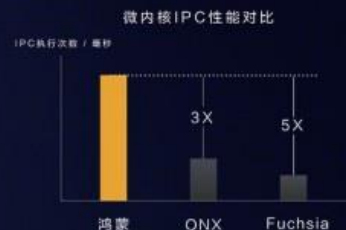
Why Optimizing IPC?

Inter-Process Communication



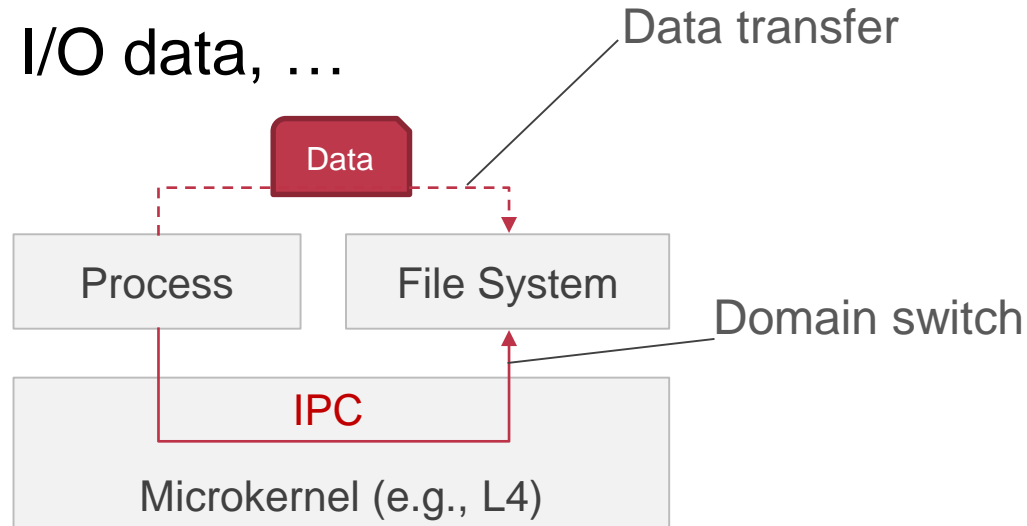
高性能IPC

进程间通信效率可提升5倍



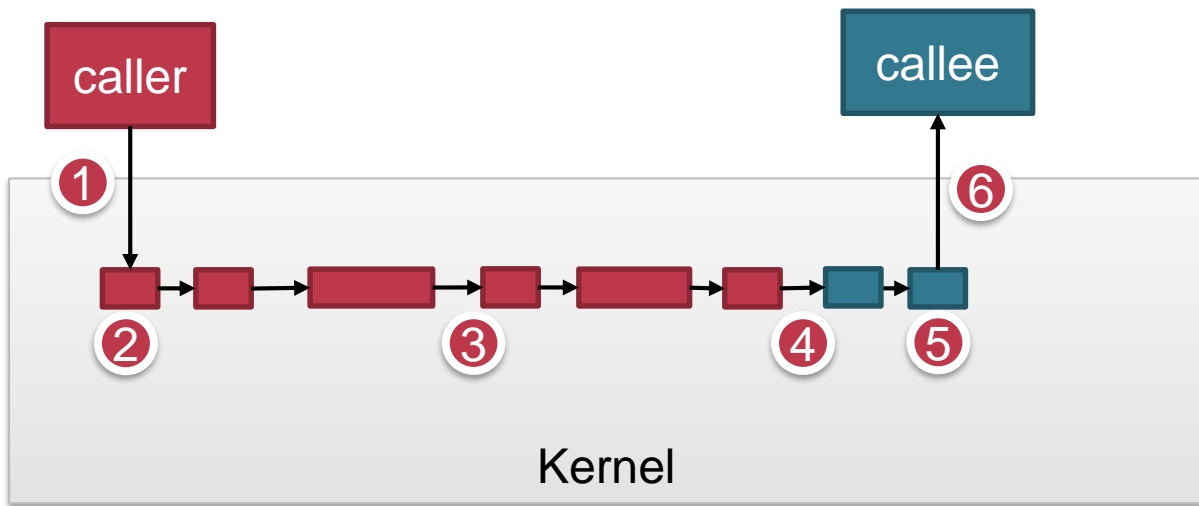
IPC Simplified

- Domain switch
 - Switching from *caller* to *callee*
- Data transfer
 - Arguments, I/O data, ...



Breakdown of Domain Switch (e.g., seL4)

- ① Switch mode (U→K)
- ② Save context ~107 cycles
- ③ Check capability ~211 cycles
- ④ Switch page table ~146 cycles
- ⑤ Restore context ~199 cycles
- ⑥ Switch mode (K→U)



End of Software Optimization

Systems			Domain Switch		Message passing			
Type	Name	Address space	No trap	No sched	Security	Handover	Granularity	Copy time
Baseline	Mach-3.0	Multi	✗	✗	✓	✗	✓	2N
Software optimization	LRPC	Multi	✗	✓	✓	✗	✓	2N
	Mach (94)	Multi	✗	✓	✓	✗	✓	N
	Tornado	Multi	✗	✓	✓	✗	✓	0+ Δ
	L4	Multi	✗	✓	✓	✗	✓	N

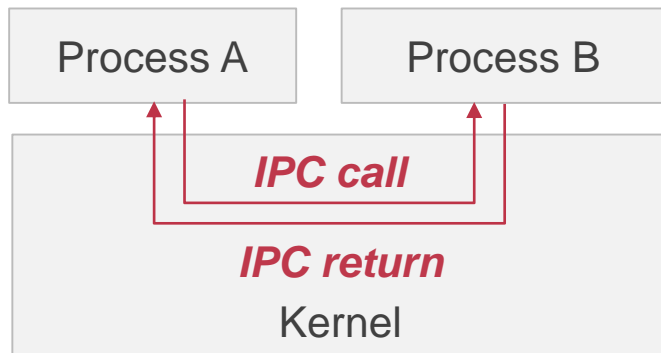
- Domain switch:
 - Trap into kernel: 2 mode switching, context save and restore (306 cycles)
 - Software capability checking (211 cycles)

Can We Implement All IPC Logics into HW?

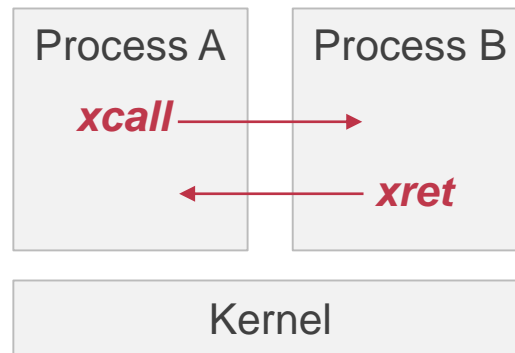
- **No, hardware should not know much about OS details**
 - L4: process direct switching
 - seL4: fastpath & slowpath
 - Google Zircon: async & throughput-target IPC
 - LRPC: migration thread model

→ **HW-SW co-design**

XPC: User-level Direct Domain Switch



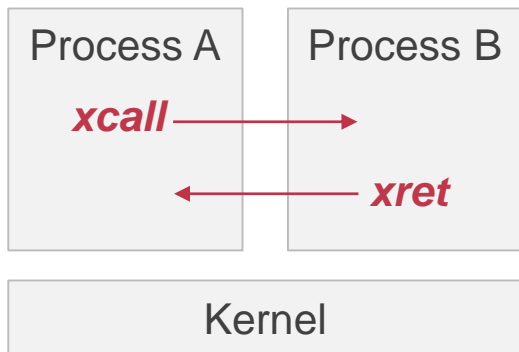
1. Switch mode (U→K)
2. Save context
3. Check capability
4. Switch page table
5. Restore context
6. Switch mode (K→U)



- Two new user instructions
- Bypass the kernel to avoid mode switching

- ~~1. Switch mode (U→K)~~
2. Save context
3. Check capability
4. Switch page table
5. Restore context
- ~~6. Switch mode (K→U)~~

Direct Domain Switch: Context



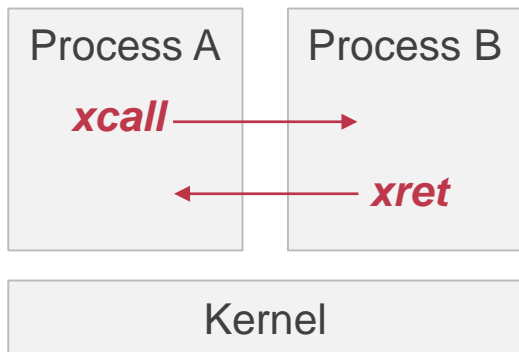
1. Switch mode (U → K)
2. Save context
3. Check capability
4. Switch page table
5. Restore context
6. Switch mode (K → U)

- **Traditional kernel**
 - Save/restore all registers
 - Lead to at least **62** *ld/st* instructions

```
46
47 trap_entry:
48
49     csrrw t0, sscratch, t0
50
51     STORE ra, (0*REGBYTES)(t0)
52     STORE sp, (1*REGBYTES)(t0)
53     STORE gp, (2*REGBYTES)(t0)
54     STORE tp, (3*REGBYTES)(t0)
55     STORE t1, (5*REGBYTES)(t0)
56     STORE t2, (6*REGBYTES)(t0)
57     STORE s0, (7*REGBYTES)(t0)
58     STORE s1, (8*REGBYTES)(t0)
59     STORE a0, (9*REGBYTES)(t0)
60     STORE a1, (10*REGBYTES)(t0)
61     STORE a2, (11*REGBYTES)(t0)
62     STORE a3, (12*REGBYTES)(t0)
63     STORE a4, (13*REGBYTES)(t0)
64     STORE a5, (14*REGBYTES)(t0)
65     STORE a6, (15*REGBYTES)(t0)
66     STORE a7, (16*REGBYTES)(t0)
67     STORE s2, (17*REGBYTES)(t0)
68     STORE s3, (18*REGBYTES)(t0)
69     STORE s4, (19*REGBYTES)(t0)
70     STORE s5, (20*REGBYTES)(t0)
71     STORE s6, (21*REGBYTES)(t0)
72     STORE s7, (22*REGBYTES)(t0)
73     STORE s8, (23*REGBYTES)(t0)
74     STORE s9, (24*REGBYTES)(t0)
75     STORE s10, (25*REGBYTES)(t0)
76     STORE s11, (26*REGBYTES)(t0)
77     STORE t3, (27*REGBYTES)(t0)
78     STORE t4, (28*REGBYTES)(t0)
79     STORE t5, (29*REGBYTES)(t0)
80     STORE t6, (30*REGBYTES)(t0)
81
```

Store **all** 31 user-
registers at entry
point (seL4 RISC-V)

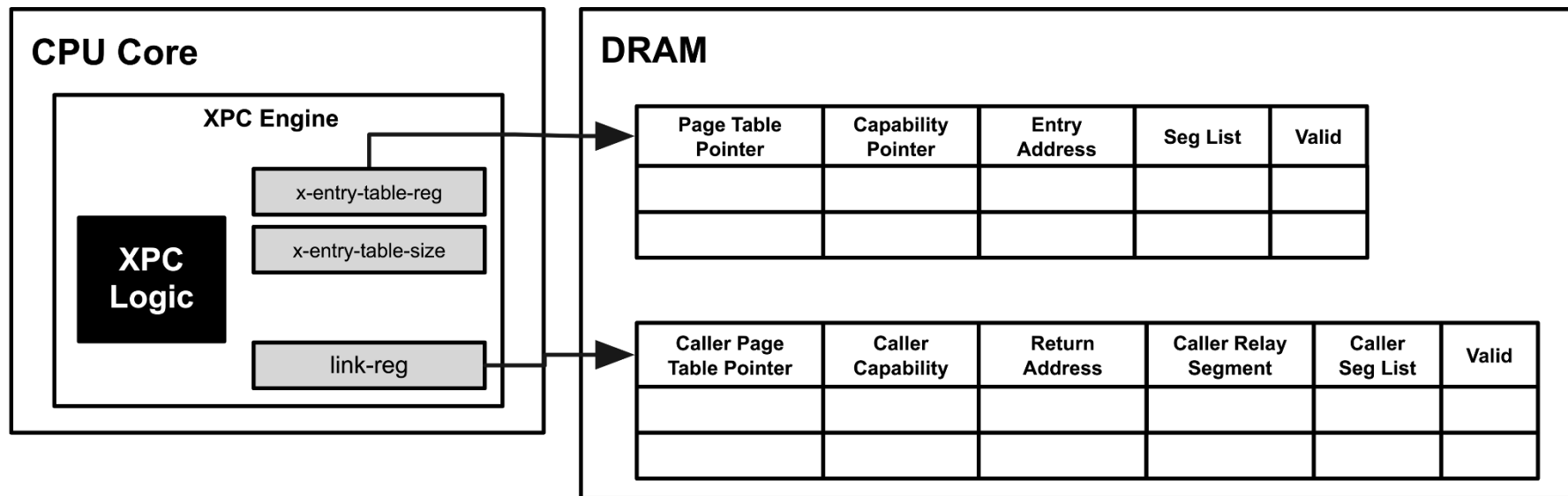
Direct Domain Switch: Context



1. ~~Switch mode (U → K)~~
2. Save context *selectively*
3. Check capability
4. Switch page table
5. Restore context *selectively*
6. ~~Switch mode (K → U)~~

- **Traditional kernel**
 - Save/restore all registers
 - Lead to at least **62** *ld/st* instructions
- **XPC: selectively save/restore**
 - Hardware: save *necessary* registers
 - Program counter
 - Page table base register
 - Software: *selectively* save registers
 - Decided by applications themselves

Direct Domain Switch: Context



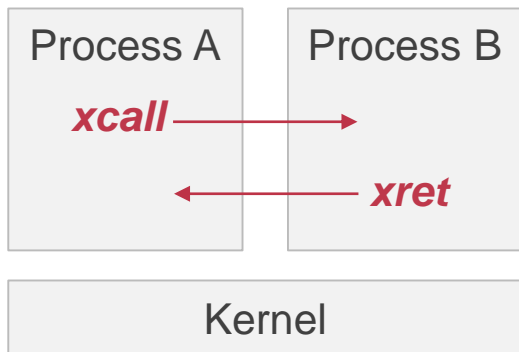
- **X-entry table**

- Record callee's context
- In-memory & scale
- Global

- **Link stack**

- Save caller's privileged context
- Per-thread

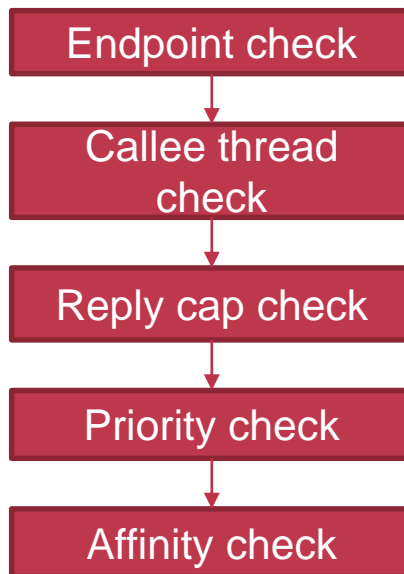
Direct Domain Switch: Capability



1. Switch mode (U→K)
2. Save context *selectively*
3. Check capability
4. Switch page table
5. Restore context *selectively*
6. Switch mode (K→U)

- **Traditional kernel**

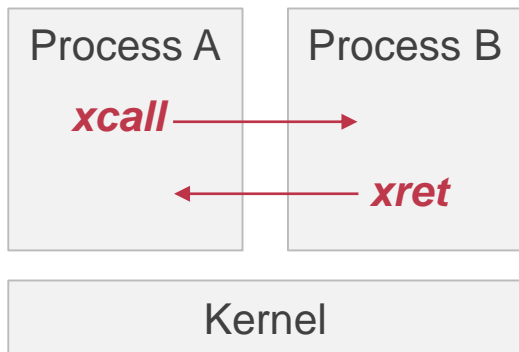
- Check capability: whether a caller can call a callee (and other checks). ~ 211 cycles



Long path in seL4!

Observation: Most of the checking can be done offline

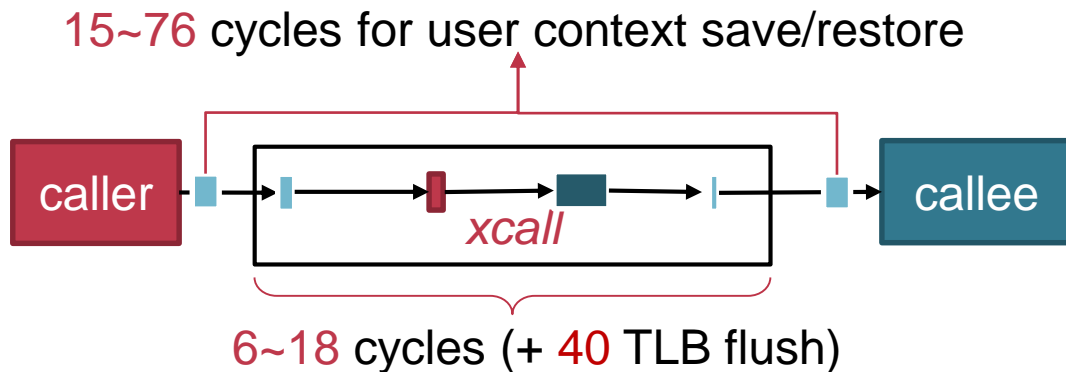
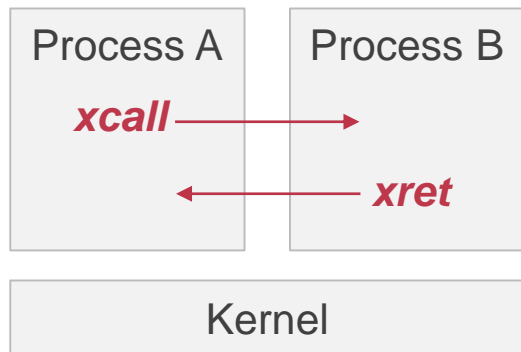
Direct Domain Switch: Capability



1. Switch mode (U→K)
2. Save context **selectively**
3. Check capability **result bit**
4. Switch page table
5. Restore context **selectively**
6. Switch mode (K→U)

- **Traditional kernel**
 - Check capability: whether a caller can call a callee (and other checks). ~ **211 cycles**
- **XPC: hardware-aware capability**
 - Introduce a **capability table** in hardware
 - Each bit represents capability of caller→callee
 - Software: set capability **offline**
 - Set bits in capability table if check passes
 - Hardware: check capability bits **online**
 - Check the bits in the capability table
 - Within 1 cycle. Fast!

Direct Domain Switch: Put All Together



best case: 21 cycles an IPC call

- | | |
|--------------------------------------|--|
| 0. Switch mode (U → K) | |
| 2. Save PC <i>Link-stack</i> | ← 1. Save registers selectively by caller |
| 3. Check result bit <i>Xcall-cap</i> | Check capability and set result bits (offline) |
| 4. Switch page table & flush TLB | |
| 5. Restore PC <i>Xentry-table</i> | → 6. Restore registers selectively by callee |
| 0. Switch mode (K → U) | |

in Hardware

in Software

CoWLight

ARCHITECTURAL SUPPORT FOR COPY-ON-WRITE

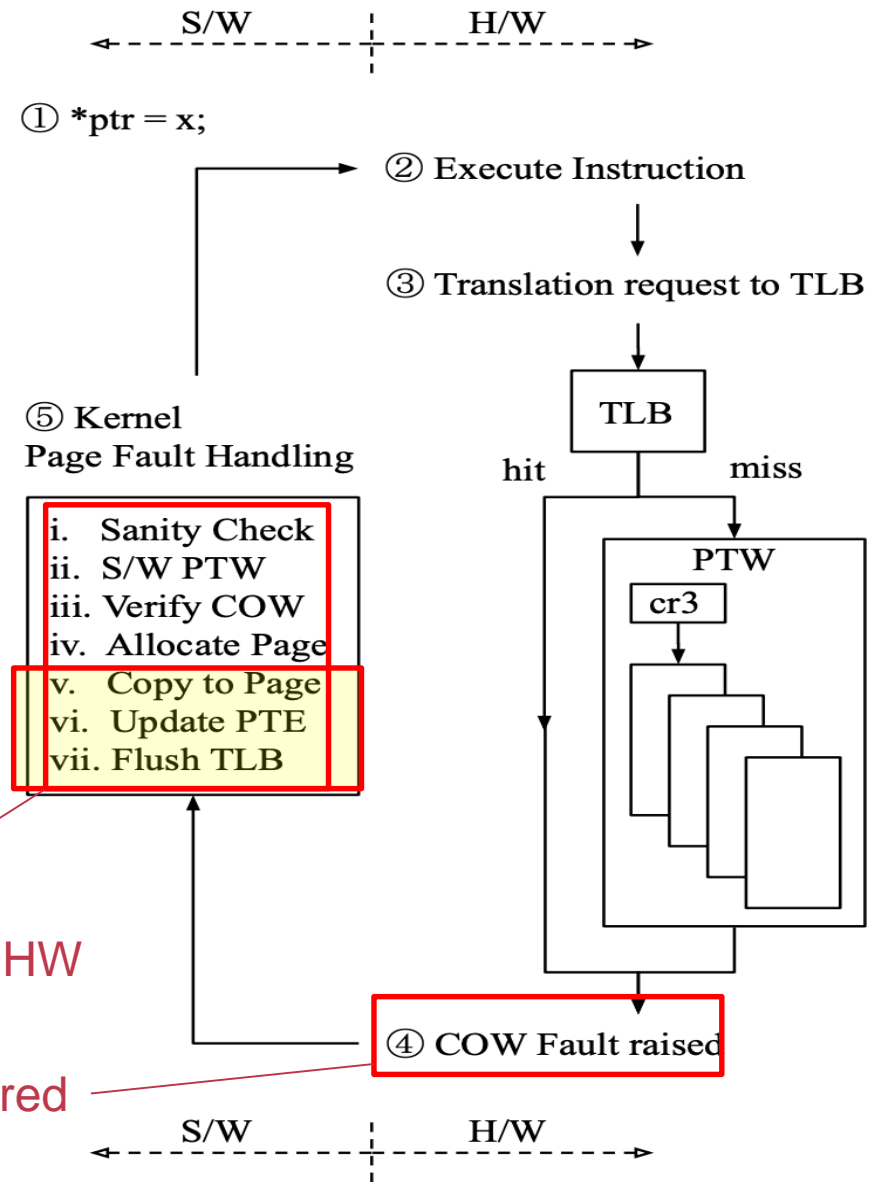
CoWLight: Hardware Assisted Copy-on-Write Fault Handling for Secure Deduplication.
Kumar T, Santhosh, et al. (HASP'2019)

CoW Fault Handling

- Hardware trigger faults
- OS is responsible to handle all the things

Simple & suitable for HW

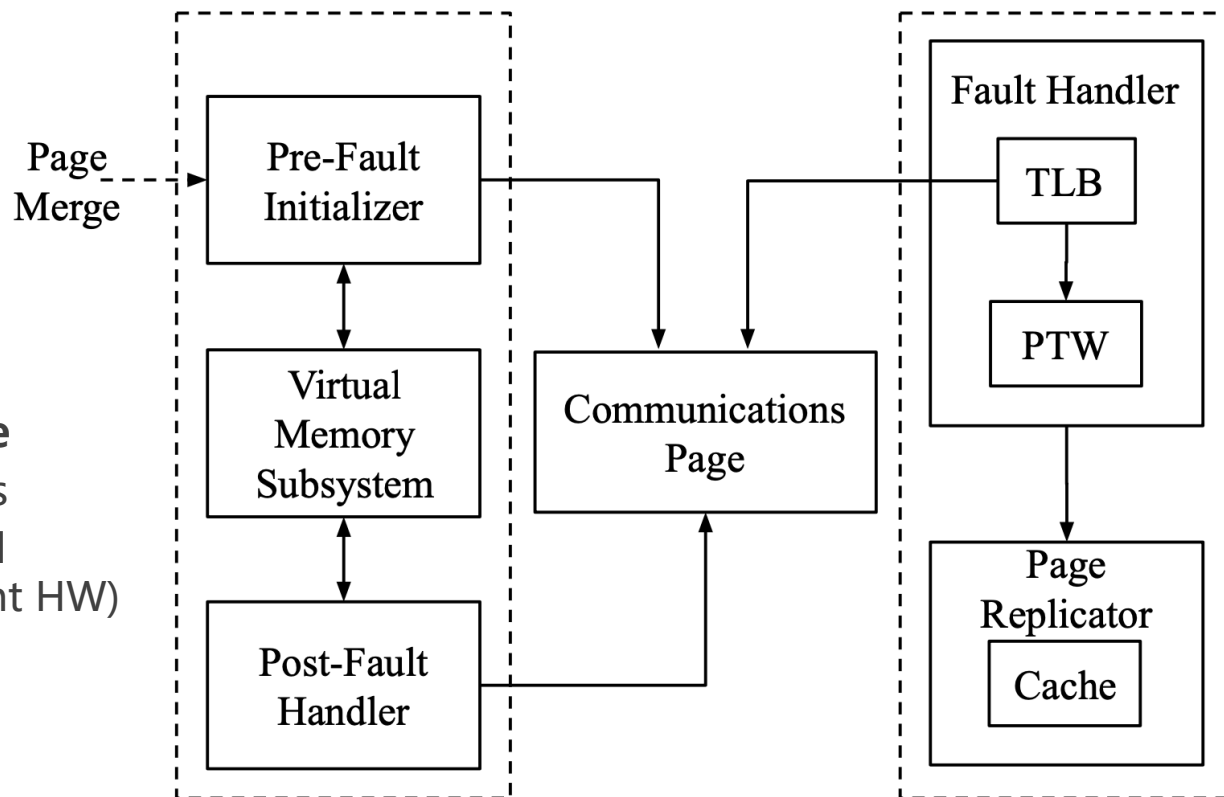
CoW triggered



CowLight

CoWLight OS Modifications

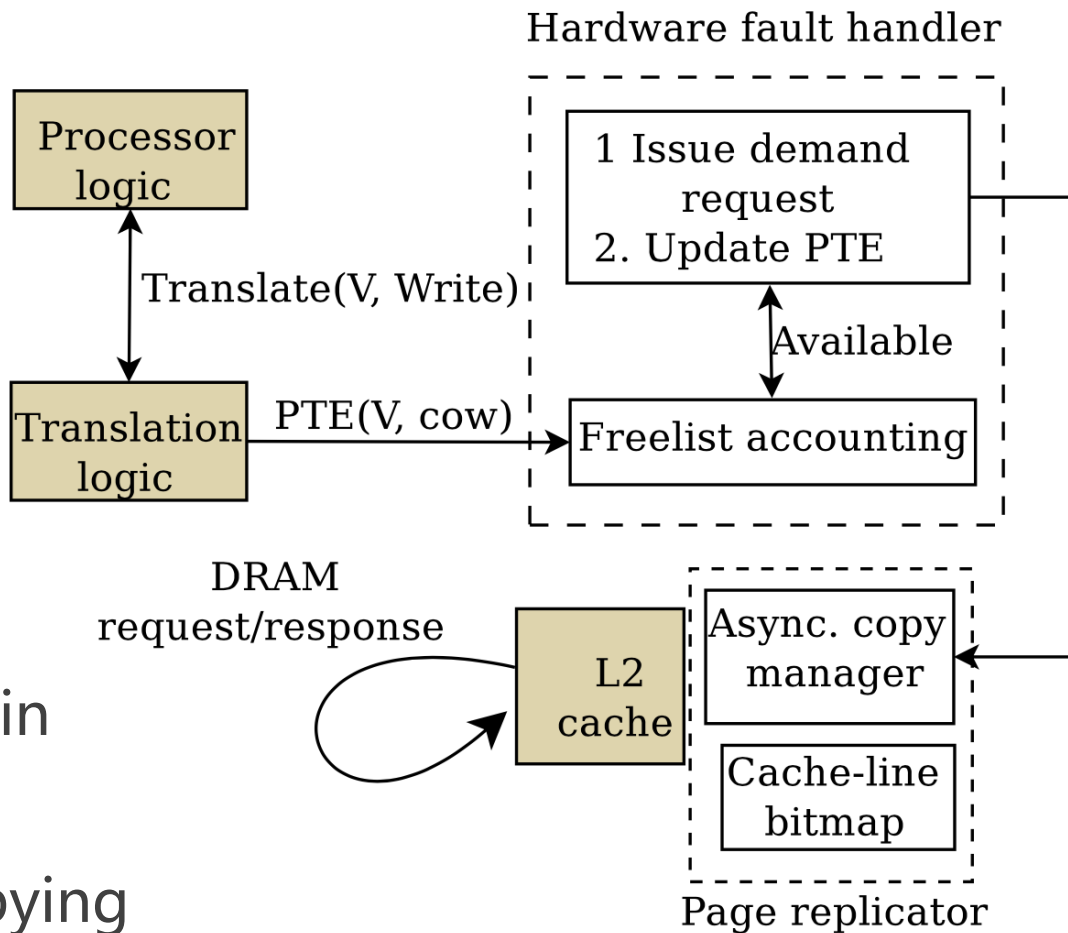
CoWLight H/W Modifications



- **Communication page**

- Free physical pages
- Consumed physical pages (by CoWLight HW)

CowLight Hardware Extensions



- CoW bit in PTE
- Update PTE with pages in freelist
- Asynchronous page copying

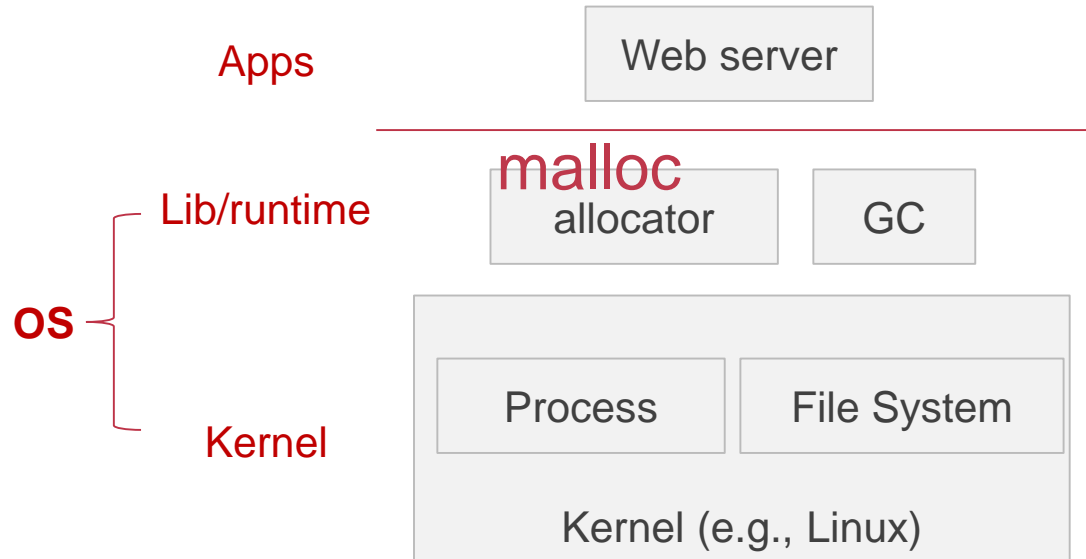
Mallacc



ARCHITECTURAL SUPPORT FOR MEMORY ALLOCATION

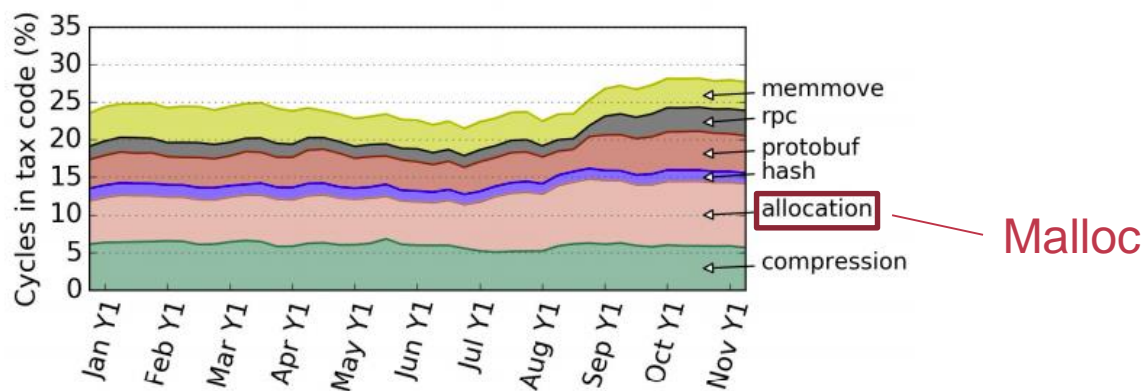
Mallacc: Accelerating memory allocation. Kanev, Svilen, et al. (ASPLOS'2017)

Optimization Target



Why Optimizing Malloc?

We are used to deep accelerators motivated by 90/10 rules



[Kanev et. al, ISCA 15]

What if there is no 90% hotspot?

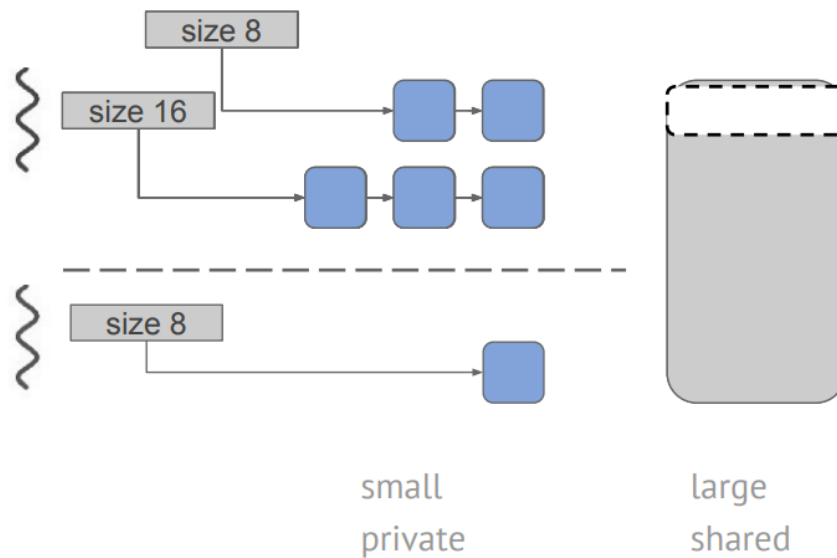
Largest hotspots at 1-7% **datacenter tax (datacenter OS)**

Allocator

- Bookkeep available memory
 - get pages from the kernel (e.g., mmap)
 - give them out to application requests (e.g., malloc)
- Modern allocators (tcmalloc, jemalloc, Hoard, ...)
 - round requests in **size classes** , e.g., Malloc(240)
 - keep **hierarchical pools** of memory
 - keep closest pools **thread-private**

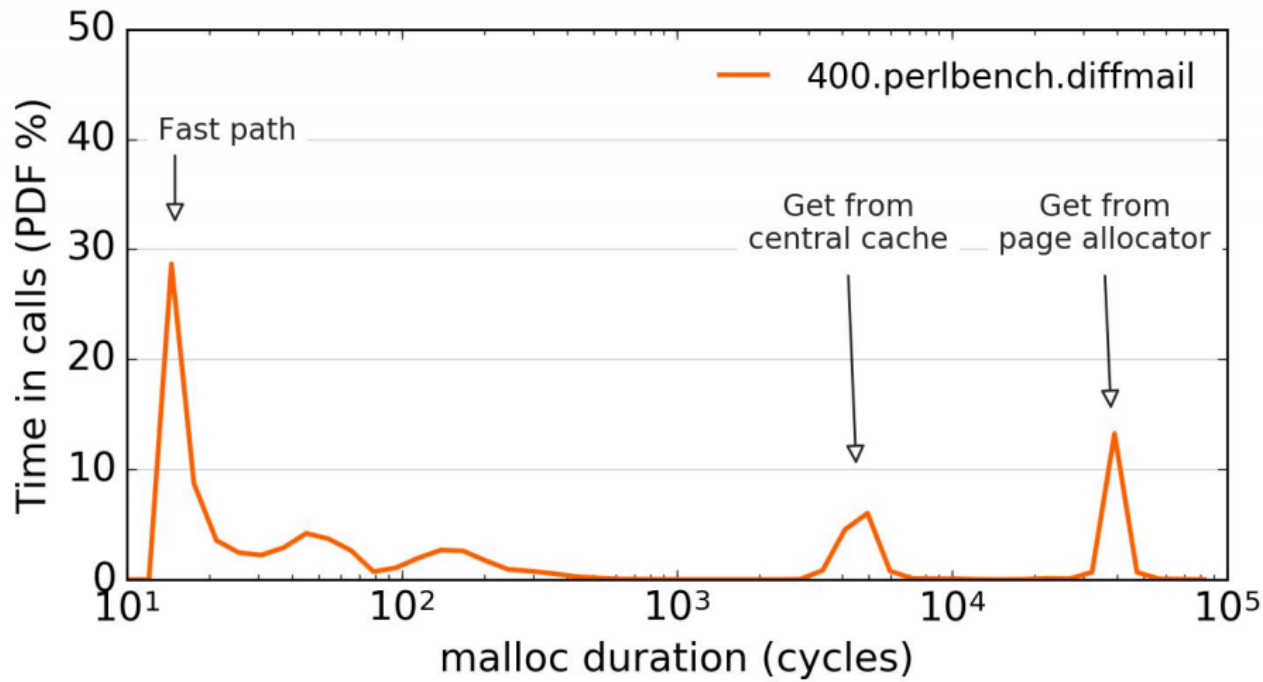
Allocator

➤ Case



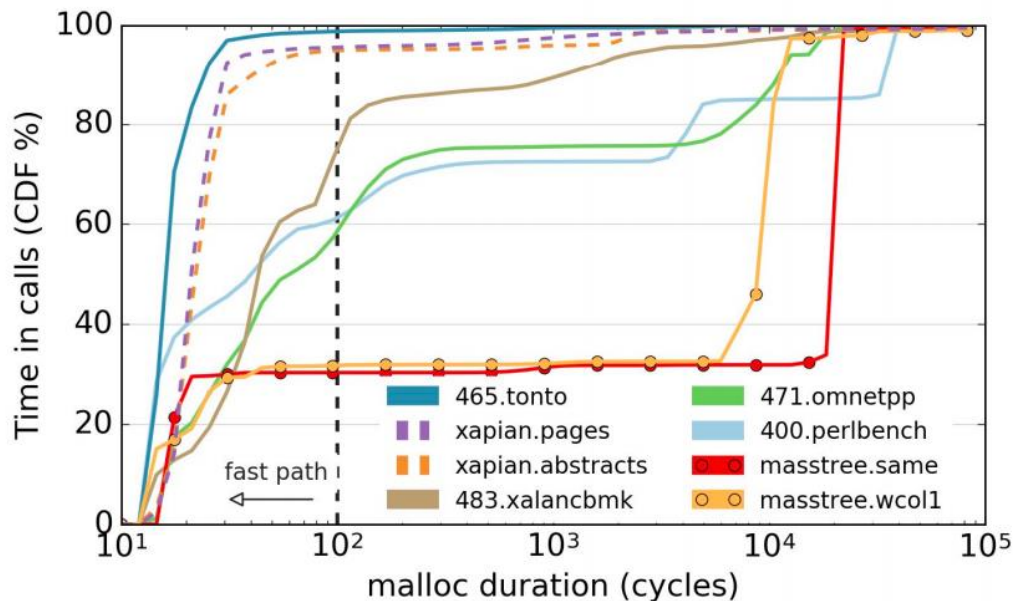
Allocator Performance

- Most optimization effort spent at avoiding costly shared pools
- fast paths are “fast enough”



Allocator Performance

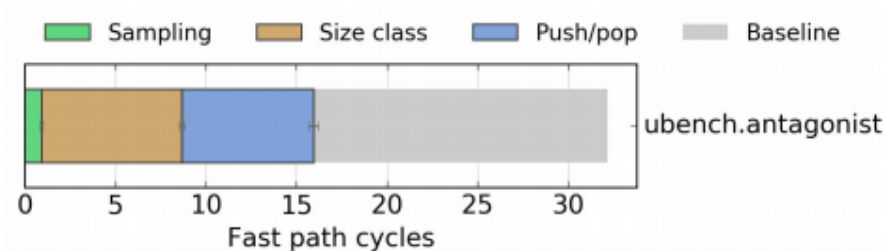
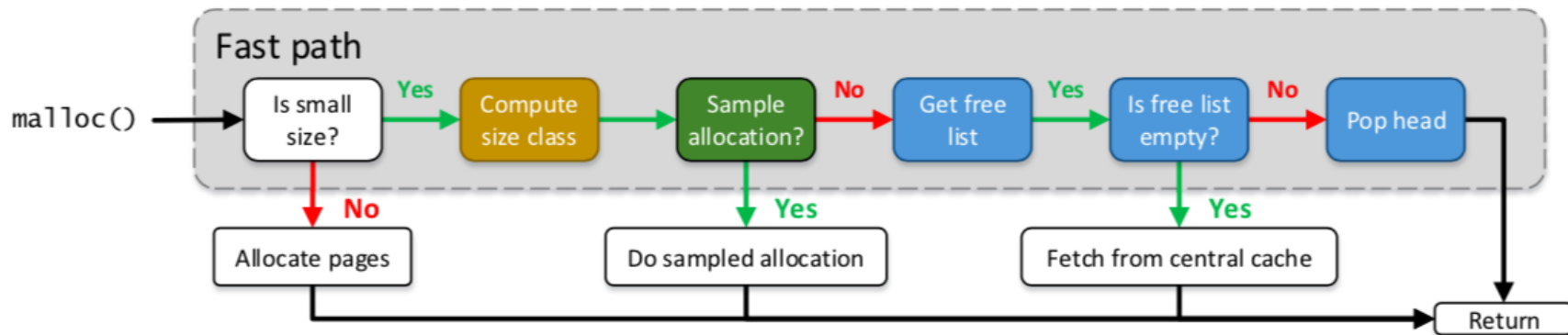
- Death by a thousand cuts
- ~~fast paths are “fast enough”~~



Fast paths can consume the bulk of allocation time (60+%)

Allocator Breakdown

- Optimize “size class” and “push/pop”



Size class loopkup

➤ Implementation

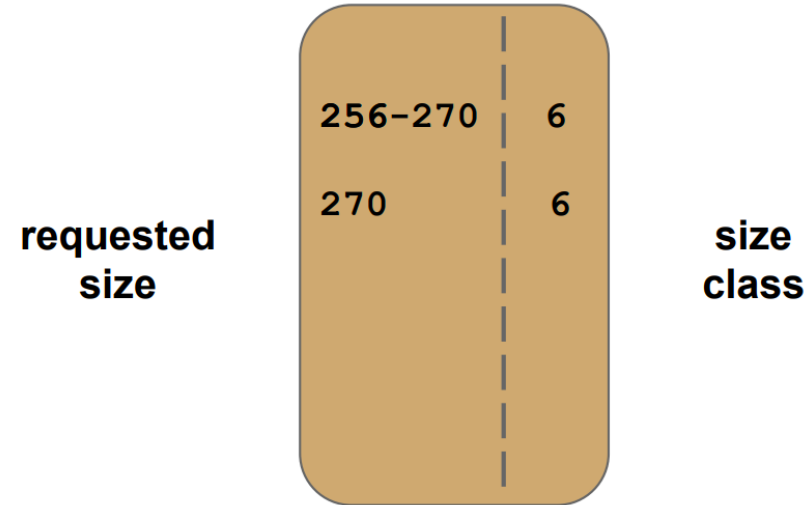
```
size_t SizeClass(size_t size) {  
    size_t class_index;  
    if (size <= 1024)  
        class_index = (size + 7) >> 3;  
    else  
        class_index = (size + 15487) >> 7;  
    return size_class_table[class_index];  
}
```

Deterministic lookup!

```
size_t class = SizeClass(requested size);  
size_t alloc_size = size_table[class];
```

Malloc Cache: memorize hot size classes

- Malloc Cache
 - request size -> size class + alloc size
 - Accelerate size class computation
- New instructions
 - **mcszlookup**(ReqSize)
 - Return: SizeClass, AllocSize
 - err : ZF =0 (otherwise ZF=1)
 - **mcszupdate**(ReqSize, AllocSize, SizeClass)



Size Class Lookup with Mallacc

- 1 cycle for fast path
- Cache miss: goto original routine

Start:

```
; rax = size class (dest)  
; rbx = allocated size (dest)  
; rcx = requested size (source)  
mcszlookup rax, rbx, rcx    ; Sets ZF  
je ComputeSizeClass          ; if ZF = 1, jump.
```

Resume:

```
; Continue with the rest of malloc.
```

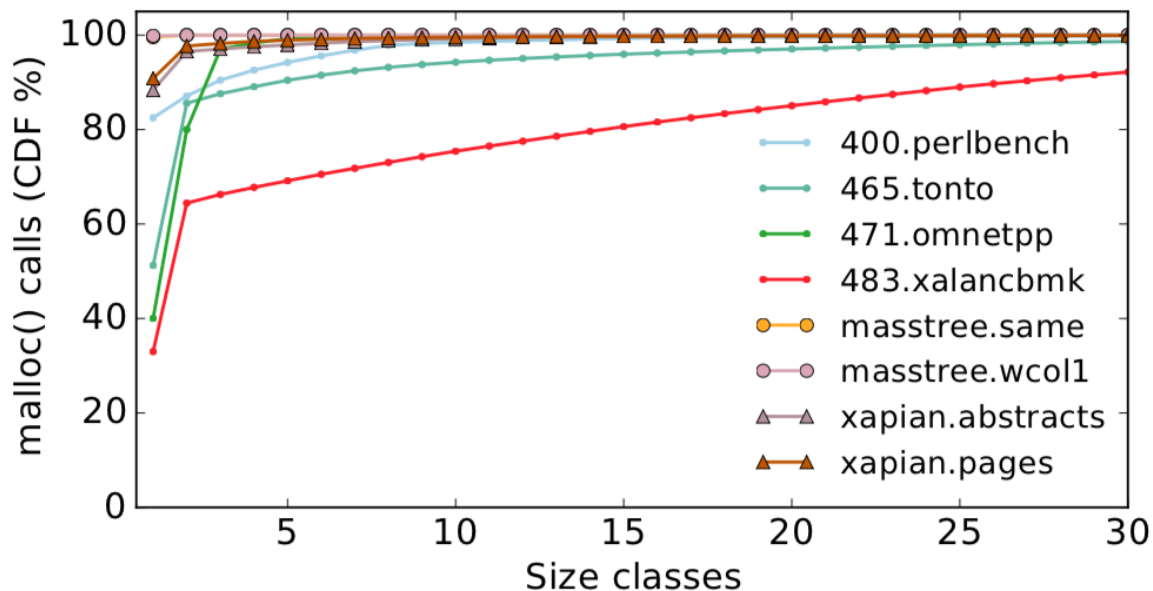
ComputeSizeClass:

```
; The usual software calculation for the size class (rax)  
; and allocated size (rbx). Then update the cache.  
mcszupdate rcx, rbx, rax  
jmp Resume
```

Why it works?

- Many benchmarks use a very **small number** of size classes.

High cache hit rate!



Push/pop on Free-lists

➤ Implementation

pop: 2*Load + 1*Store

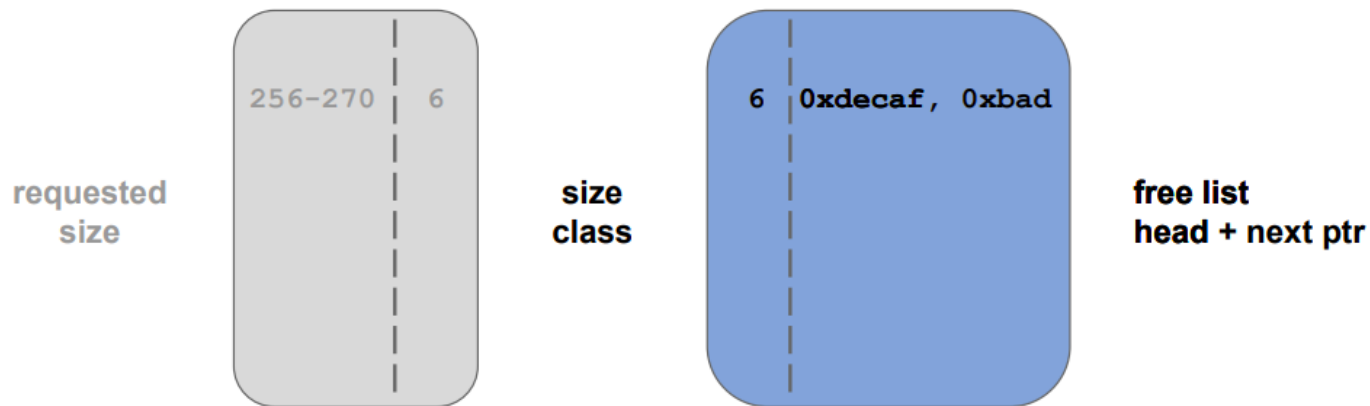
```
load temp, MEM[head]           ; Get the head.  
load next_head, MEM[temp]      ; Get head->next.  
store MEM[head], next_head     ; head = head->next.  
return temp
```

push: 1*Load + 2*Store

```
load temp, MEM[head]           ; Get the head.  
store MEM[head], new_head      ; Set new_head as head.  
store MEM[new_head], temp      ; new_head->next = temp.
```


Prefetch/Store free list heads

- Malloc Cache
 - size class → free list **head+next**
 - Accelerate push/pop on free lists



Summary

- **Architectural supports for OS have long history**
- **HW-SW co-design is key**
 - Do not put everything in HW!
- **HW-based optimizations are proposed when SW can not improve anymore**
 - Old techniques: IPC, CoW, malloc, GC