Kernel Synchronization I

Dongyoon Lee

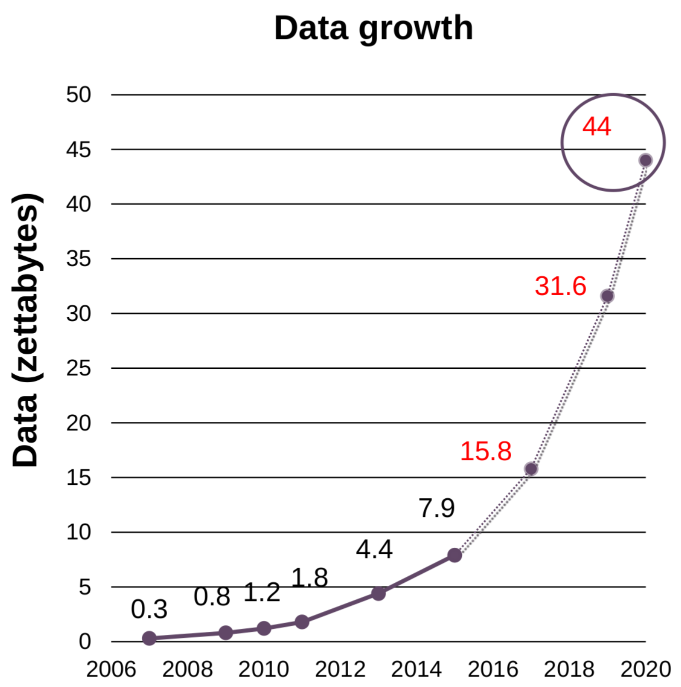
Summary of last lectures

- Tools: building, exploring, and debugging Linux kernel
- Core kernel infrastructure
    - syscall, module, kernel data structures
- Process management & scheduling
- Interrupt & interrupt handler

Today: kernel synchronization

- Kernel synchronization I
    - Background on multicore processing
    - Introduction to synchronization
- Kernel synchronization II
    - Synchronization mechanisms in Linux Kernel
- Kernel synchronization III
    - Read-Copy-Update (RCU)

Data growth is already exponential

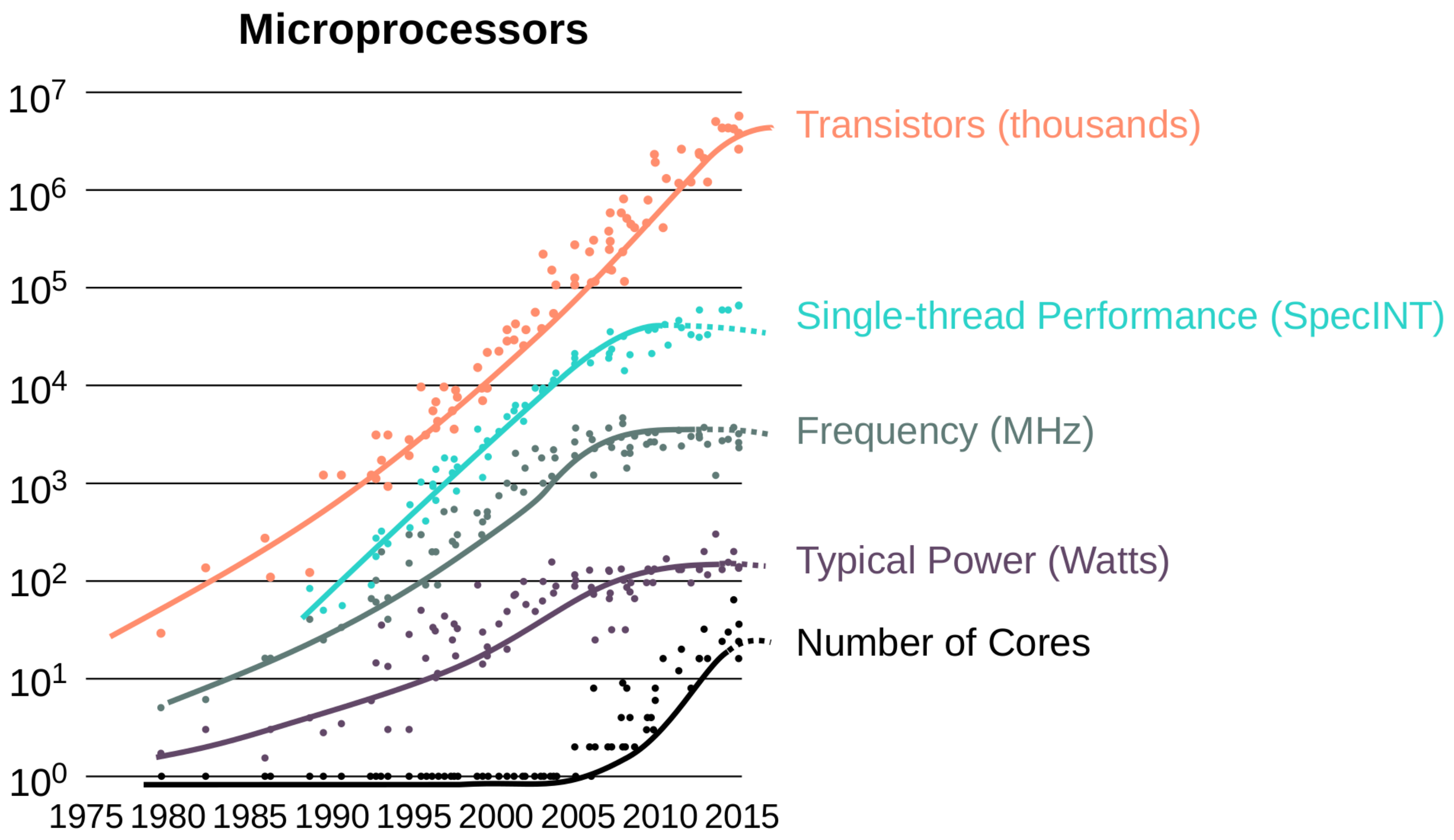


- 1 zettabytes = 10^9 terabytes

Data growth is already exponential

- **Data nearly doubles every two years (2013-20)**
- By 2020
    - 8 billion people
    - 20 billion connected devices
    - 100 billion infrastructure devices
- Need more processing power

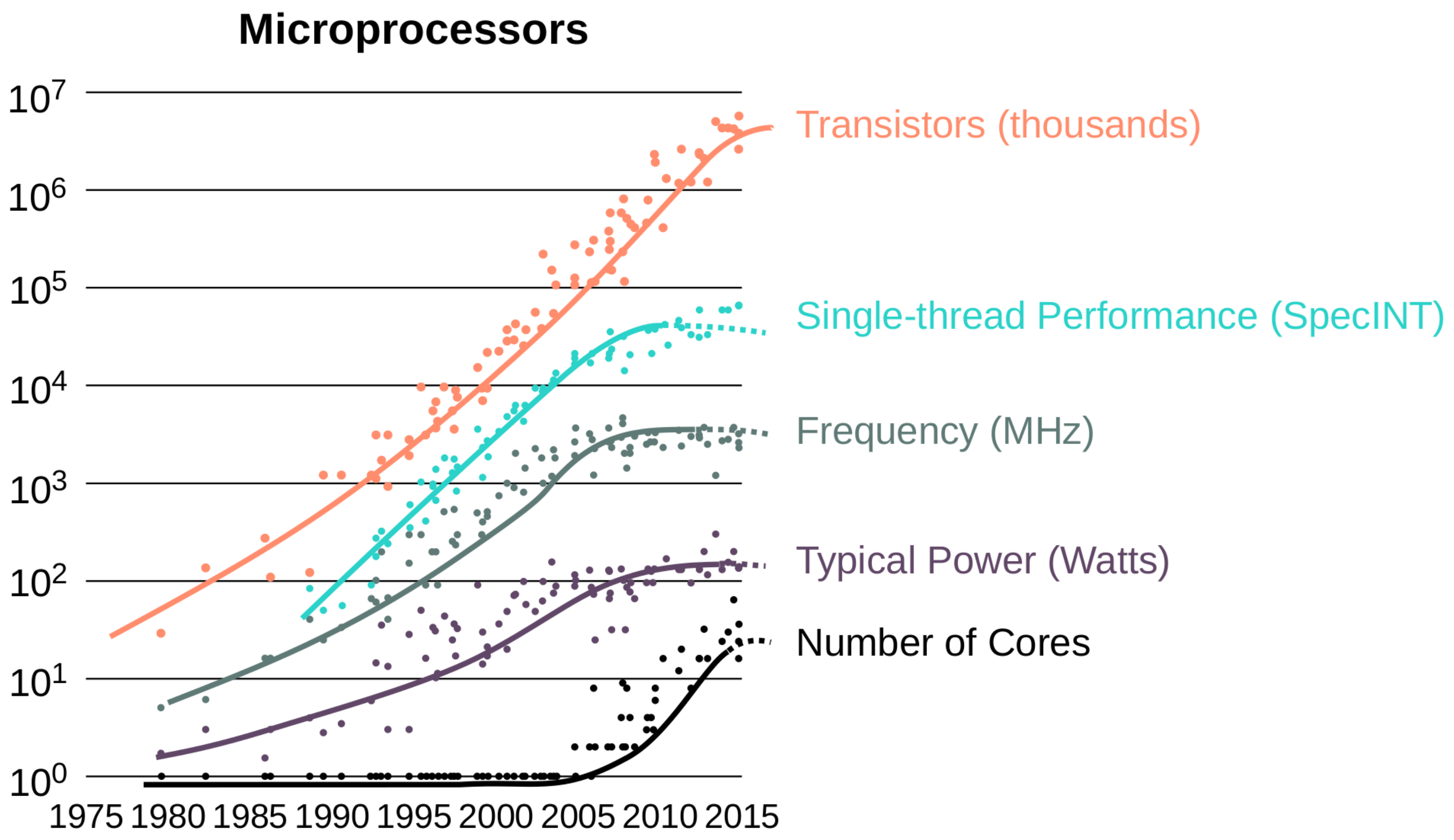
Single-core performance scaling stopped



Single-core performance scaling stopped

- **Increasing clock frequency is not possible anymore**
    - Power consumption: higher frequency → higher power consumption
    - Wire delay: range of a wire in one clock cycle
- **Limitation in Instruction Level Parallelism (ILP)**
    - 1980s: more transistors → superscalar → pipeline
        - 10 CPI (cycles per instruction) → 1 CPI
    - 1990s: multi-way issue, out-of-order issue, branch prediction
        - 1 CPI → 0.5 CPI

The new normal: multi-core processors

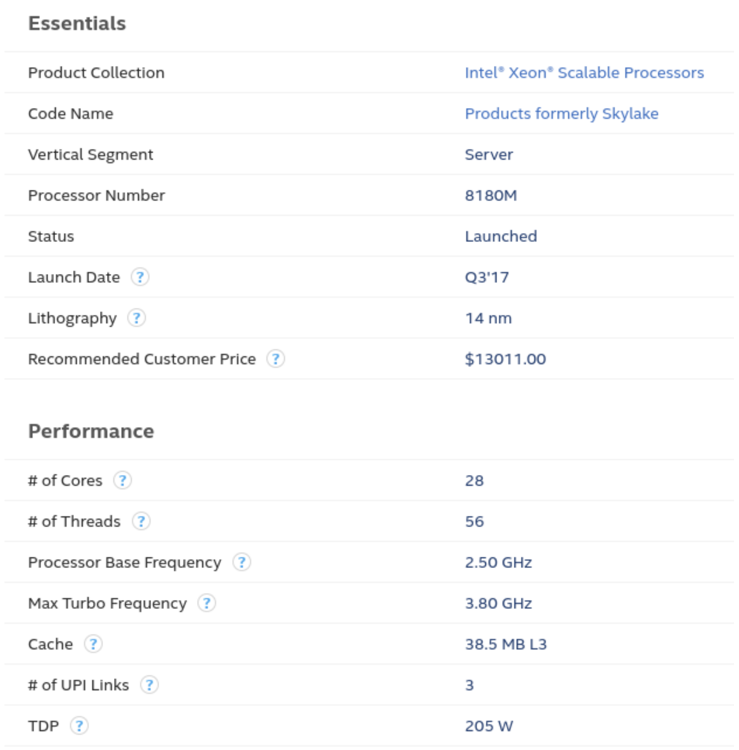


The new normal: multi-core processors

- **Moore's law**: the observation that the number of transistors in a dense integrated circuit doubles approximately every two years
- **Q: Where to use such a doubled transistors in processor design?**
- **~ 2007: make a single-core processor faster**
  - deeper processor pipeline, branch prediction, out-of-order execution, etc.
- **2007 ~: increase the number of cores in a chip**
  - multi-core processor

The new normal: multi-core processors

Example: Intel Xeon 8180M processor

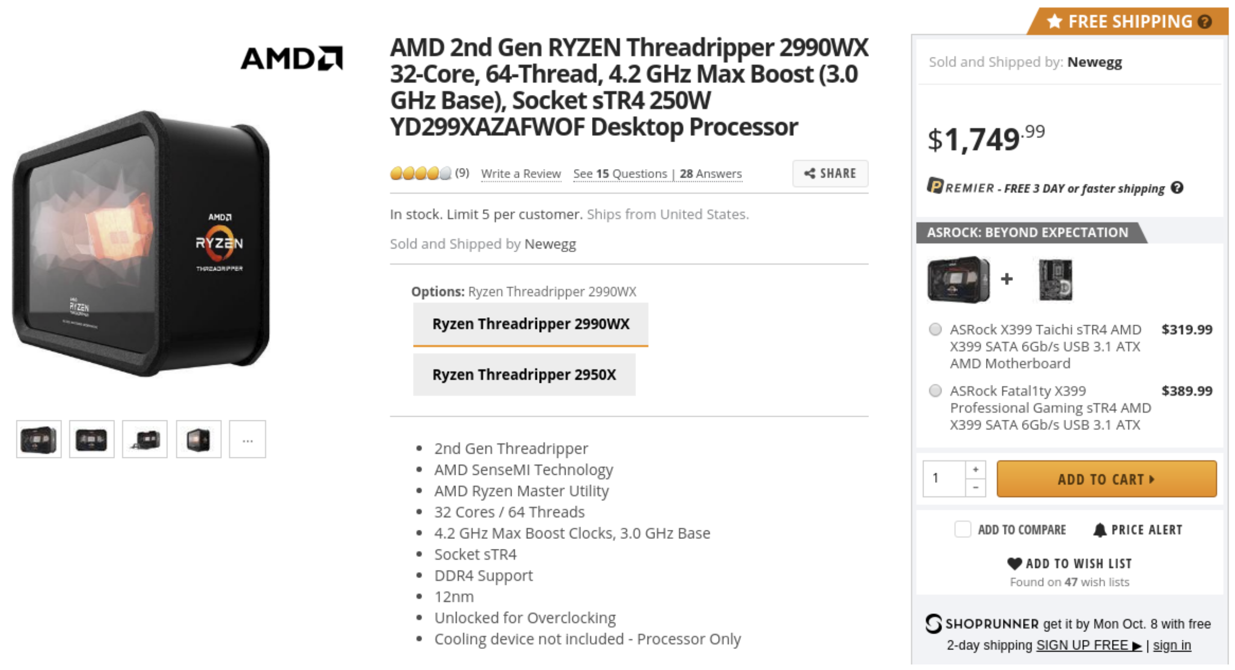


Essentials	
Product Collection	Intel® Xeon® Scalable Processors
Code Name	Products formerly Skylake
Vertical Segment	Server
Processor Number	8180M
Status	Launched
Launch Date ?	Q3'17
Lithography ?	14 nm
Recommended Customer Price ?	$13011.00

Performance	
# of Cores ?	28
# of Threads ?	56
Processor Base Frequency ?	2.50 GHz
Max Turbo Frequency ?	3.80 GHz
Cache ?	38.5 MB L3
# of UPI Links ?	3
TDP ?	205 W

- Support up to 8 sockets: 28*8 = 224 cores (or 448 H/W threads)

Example: AMD Ryzen Threadripper 2


AMD 2nd Gen RYZEN Threadripper 2990WX 32-Core, 64-Thread, 4.2 GHz Max Boost (3.0 GHz Base), Socket sTR4 250W YD299XAZAFWOF Desktop Processor

(9) Write a Review See 15 Questions | 28 Answers SHARE

In stock. Limit 5 per customer. Ships from United States.

Sold and Shipped by Newegg

Options: Ryzen Threadripper 2990WX

Ryzen Threadripper 2990WX

Ryzen Threadripper 2950X

- 2nd Gen Threadripper
- AMD SenseMI Technology
- AMD Ryzen Master Utility
- 32 Cores / 64 Threads
- 4.2 GHz Max Boost Clocks, 3.0 GHz Base
- Socket sTR4
- DDR4 Support
- 12nm
- Unlocked for Overclocking
- Cooling device not included - Processor Only

FREE SHIPPING

Sold and Shipped by: Newegg

$1,749.99

PREMIER - FREE 3 DAY or faster shipping

ASROCK: BEYOND EXPECTATION

ASRock X399 Taichi sTR4 AMD X399 SATA 6Gb/s USB 3.1 ATX AMD Motherboard	$319.99
ASRock Fatal1ty X399 Professional Gaming sTR4 AMD X399 SATA 6Gb/s USB 3.1 ATX	$389.99

1 ADD TO CART

ADD TO COMPARE PRICE ALERT

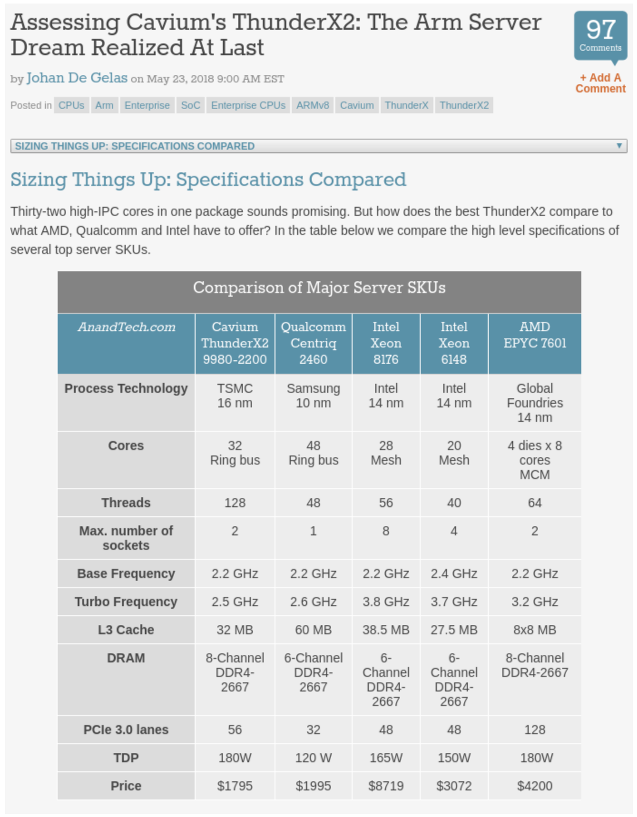
ADD TO WISH LIST

Found on 47 wish lists

SHOPRUNNER get it by Mon Oct. 8 with free 2-day shipping SIGN UP FREE ▶ | sign in

- Support up to 2 sockets: 32*2 = 64 cores (or 128 H/W threads)

Example: Cavium's ThunderX2 (ARM server)



Assessing Cavium's ThunderX2: The Arm Server Dream Realized At Last

97 Comments

by Johan De Gelas on May 23, 2018 9:00 AM EST

+ Add A Comment

Posted in CPUs Arm Enterprise SoC Enterprise CPUs ARMv8 Cavium ThunderX ThunderX2

SIZING THINGS UP: SPECIFICATIONS COMPARED

Sizing Things Up: Specifications Compared

Thirty-two high-IPC cores in one package sounds promising. But how does the best ThunderX2 compare to what AMD, Qualcomm and Intel have to offer? In the table below we compare the high level specifications of several top server SKUs.

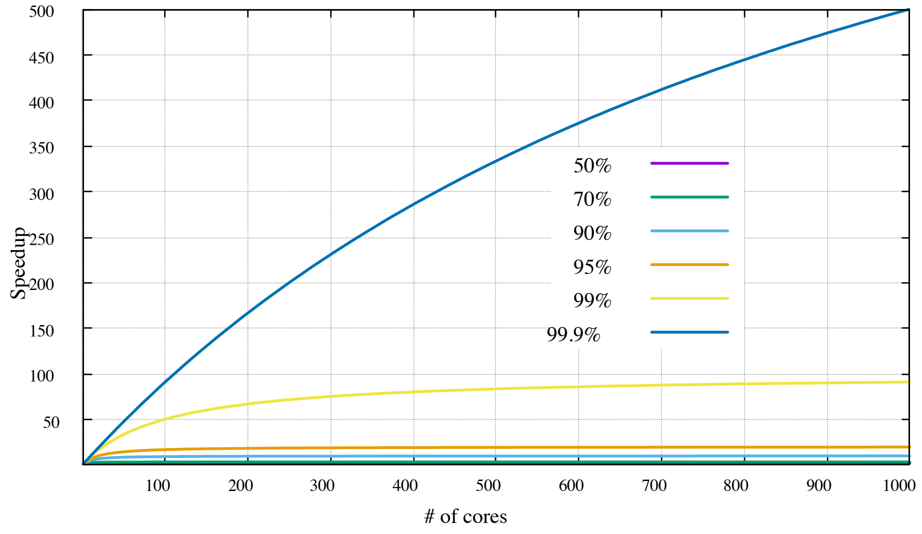
Comparison of Major Server SKUs

AnandTech.com	Cavium ThunderX2 9980-2200	Qualcomm Centriq 2460	Intel Xeon 8176	Intel Xeon 6148	AMD EPYC 7601
Process Technology	TSMC 16 nm	Samsung 10 nm	Intel 14 nm	Intel 14 nm	Global Foundries 14 nm
Cores	32 Ring bus	48 Ring bus	28 Mesh	20 Mesh	4 dies x 8 cores MCM
Threads	128	48	56	40	64
Max. number of sockets	2	1	8	4	2
Base Frequency	2.2 GHz	2.2 GHz	2.2 GHz	2.4 GHz	2.2 GHz
Turbo Frequency	2.5 GHz	2.6 GHz	3.8 GHz	3.7 GHz	3.2 GHz
L3 Cache	32 MB	60 MB	38.5 MB	27.5 MB	8x8 MB
DRAM	8-Channel DDR4-2667	6-Channel DDR4-2667	6-Channel DDR4-2667	6-Channel DDR4-2667	8-Channel DDR4-2667
PCIe 3.0 lanes	56	32	48	48	128
TDP	180W	120 W	165W	150W	180W
Price	$1795	$1995	$8719	$3072	$4200

- Support up to 2 sockets: 32*2 = 64 cores (or 256 H/W threads)

Small sequential part does matter

- **Amdhal's Law: theoretical speedup of the execution of a task**
  - Speedup = 1 / (1 - p + p/n)
  - `p` : parallel portion of a task, `n` : the number of CPU core



Where are such sequential parts?

- Applications: sequential algorithm
- Libraries: memory allocator (buddy structure)
- Operating system kernel
  - Memory managment: VMA (virtual memory area)
  - File system: file descriptor table, journaling
  - Network stack: receive queue
  - **Your application may not scale even if its design and implementation is scalable**

Introduction to kernel synchronization

- The kernel is programmed using the shared memory model
- **Critical section (also called critical region)**
    - Code paths that access and manipulate shared data
    - Must execute atomically without interruption
    - Should not be executed in parallel on SMP → *sequential part*
- **Race condition**
    - Two threads concurrently executing the same critical region → *Bug!*

The case of concurrent data access in kernel

- Real concurrent access on multiple CPUs
  - Same as user-space thread programming
- Preemption on a single core
  - Same as user-space thread programming
- **Interrupt**
  - Only in kernel-space programming
  - *Is a data structure accessed in an interrupt context, top-half or bottom-half?*

Why do we need protection?

- Withdrawing money from an ATM

```
int total = get_total_from_account();     /* total funds in account */
int withdrawal = get_withdrawal_amount(); /* amount asked to withdrawal */

/* check whether the user has enough funds in her account */
if (total < withdrawal) {
    error("You do not have that much money!")
    return -1;
}

/* OK, the user has enough money:
 * deduct the withdrawal amount from her total */
total -= withdrawal;
update_total_funds(total);

/* give the user their money */
spit_out_money(withdrawal);
```

Concurrent withdrawal from an ATM

- *What happen if two transactions are happening nearly at the same time?*
    - Shared credit card account with your spouse
- Suppose that
    - total == 105
    - withdrawal1 == 100
    - withdrawal2 == 50
- Either of one transaction should fail because `(100 + 50) > 105`

One possible incorrect scenario

1. Two threads check that `100 < 105` and `50 < 105` (Line 5)
2. Thread 1 updates (Line 13)
    - `total = 105 - 100 = 5`
3. Thread 2 updates (Line 13)
    - `total = 105 - 50 = 55`

- **Total withdrawal = 150 but there is 55 left on the account!**
- **Must lock the account during certain operations, make each transaction atomic**

Updating a single variable

```
int i;

void foo(void)
{
    i++;
}
```

- **Q: What happens if two threads concurrently execute `foo()`?**
- **Q: What happens if two threads concurrently update `i`?**
- **Q: Is incrementing `i` atomic operation?**

Updating a single variable

- A single C statement

```
/* C code */
01: i++;
```

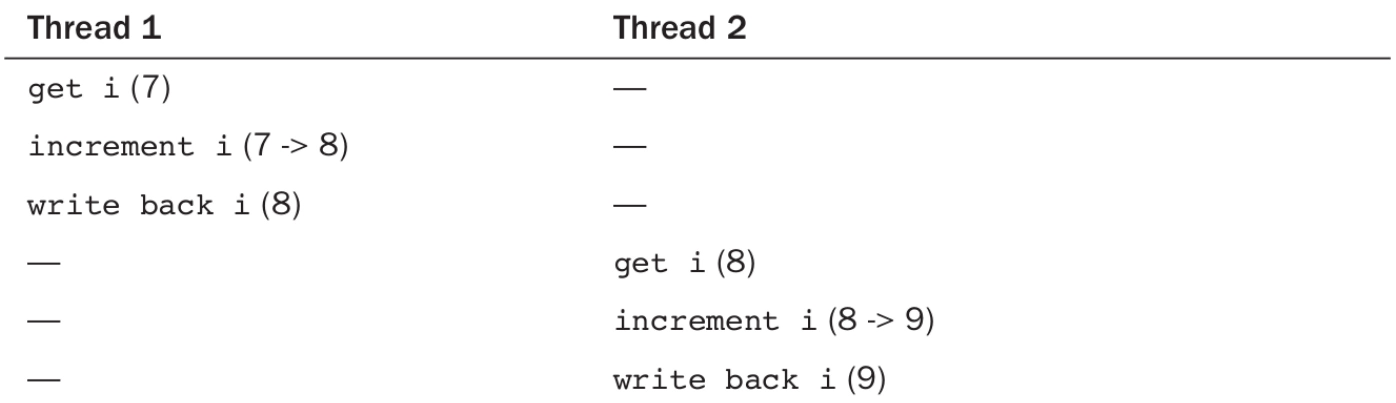
- It can be translated into multiple machine instructions

```
/* Machine instructions */
01: get the current value of i and copy it into a register
02: add one to the value stored in the register
03: write back to memory the new value of i
```

- Now, check what happens if two threads concurrently update `i`

Updating a single variable

- Two threads are running. Initial value of `i` is 7

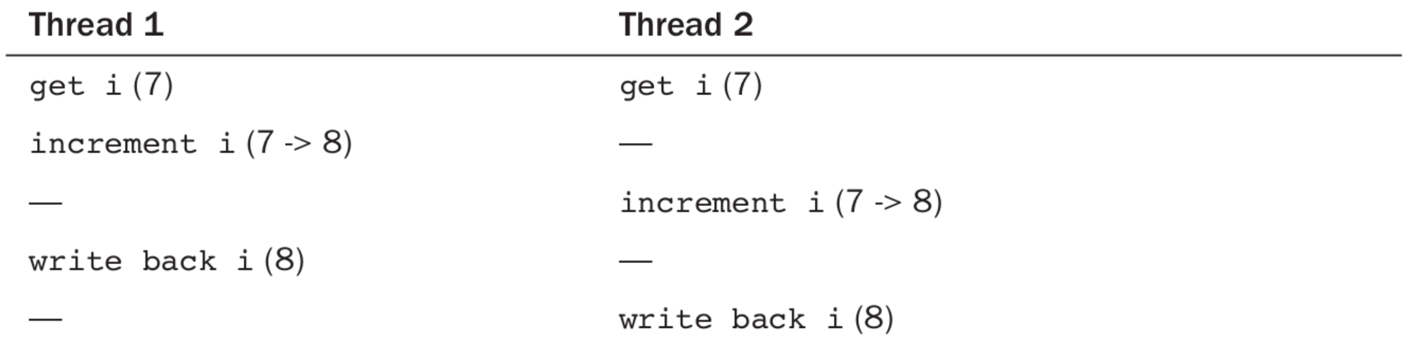


Thread 1	Thread 2
`get i` (7)	—
`increment i` (7 -> 8)	—
`write back i` (8)	—
—	`get i` (8)
—	`increment i` (8 -> 9)
—	`write back i` (9)

- As expected, 7 incremented twice is 9

Updating a single variable

- Two threads are running. Initial value of `i` is 7



Thread 1	Thread 2
`get i` (7)	`get i` (7)
`increment i` (7 -> 8)	—
—	`increment i` (7 -> 8)
`write back i` (8)	—
—	`write back i` (8)

- If both threads of execution read the initial value of `i` before it is incremented, both threads increment and save the same value.

Solution: using an *atomic instruction*

Thread 1	Thread 2
`increment & store i` (7 -> 8)	—
—	`increment & store i` (8 -> 9)

Or conversely

Thread 1	Thread 2
—	`increment & store` (7 -> 8)
`increment & store` (8 -> 9)	—

- It would never be possible for the two atomic operations to interleave.
- The processor would physically ensure that it was impossible.

x86 example of an atomic instruction

- `XADD DEST SRC`
- Operation
    - TEMP = SRC + DEST
    - SRC = DEST
    - DEST = TEMP
- `LOCK XADD DEST SRC`
- This instruction can be used with a LOCK prefix to allow the instruction to be executed **atomically**.

Wait! Then what is a `volatile` for?

- Operations on `volatile` variables are **not** atomic
- They shall not be **optimized out** or **reordered** by compiler optimization

```
/* C code */
int i;
void foo(void)
{
    /* ... */
    i++;
    /* ... */
}

/* Compiler-generated machine instructions */
/* Non-volatile variables can be optimized out without
 * actually accessing its memory location */
(01: get the current value of i and copy it into a register) <- optimized out
 02: add one to the value stored in the register
(03: write back to memory the new value of i)                <- optimized out
```

Wait! Then what is a `volatile` for?

- They shall not be **optimized out** or **reordered** by compiler optimization

```
/* C code */
int j, i;
void foo(void)
{
    i++;
    j++;
}

/* Compiler-generated machine instructions */
/* Non-volatile variables can be reordered
 * by compiler optimization */
(01/j: get the current value of j and copy it into a register)
(01/i: get the current value of i and copy it into a register)
 02/j: add one to the value stored in the register for j
 02/i: add one to the value stored in the register for i
(03/j: write back to memory the new value of j)
(03/i: write back to memory the new value of i)
```

Wait! Then what is a `volatile` for?

- Operations on `volatile` variables are guaranteed not optimized out or reordered → **disabling compiler optimization**

```
/* C code */
volatile int j, i;
void foo(void)
{
    i++;
    j++;
}

/* Compiler-generated machine instructions */
/* Volatile variables can be optimized out or reordered
 * by compiler optimization */
01/i: get the current value of i and copy it into a register
02/i: add one to the value stored in the register for i
03/i: write back to memory the new value of i
01/j: get the current value of j and copy it into a register
02/j: add one to the value stored in the register for j
03/j: write back to memory the new value of j
```

When we should use `volatile`?

- Memory location can be modified by other entity
    - Other threads for a memory location
    - Other processes for a shared memory location
    - IO devices for an IO address

Locking

- Atomic operations are not sufficient for protecting shared data in long and complex critical regions
  - E.g., `page_tree` of an `inode` (page cache)
- What is needed is a way of making sure that only one thread manipulates the data structure at a time
  - A mechanism for preventing access to a resource while another thread of execution is in the marked region. → **lock**

Linux radix tree example

```
/* linux/include/linux/fs.h */

struct inode {         /** metadata of a file */
    umode_t                i_mode;        /* permission: rwxrw-r-- */
    struct super_block   *i_sb;           /* a file system instance */
    struct address_space *i_mapping;      /* page cache */
};

struct address_space { /** page cache of an inode */
    struct inode             *host;       /* owner: inode, block_device */
    struct radix_tree_root  page_tree;   /* radix tree of all pages
                                          * - i.e., page cache of an inode
                                          * - key:   file offset
                                          * - value: cached page */
    spinlock_t              tree_lock;   /* lock protecting it */
};
```

Linux radix tree example

```
    Thread 1                                        Thread 2
=======================================  ========================================
 Try to lock the tree_lock
 Succeeded: acquired the tree_lock          Try to lock the tree_lock
 Access page_tree                           Failed: waiting...
 ...                                        Waiting...
 Unlock the tree_lock                       Waiting...
 ...                                        Succeeded: acquired the tree_lock
                                            Access page_tree...
                                            ...
                                            Unlock the tree_lock
```

- Locks are entirely a programming construct that the programmer must take advantage of → No protection generally ends up in data corruption
- Linux provides various locking mechanisms
  - Non-blocking (or spinning) locks, blocking (or sleeping) locks

Causes of concurrency

- **Symmetrical multiprocessing (true concurrency)**
  - Two or more processors can execute kernel code at exactly the same time.
- **Kernel preemption (pseudo-concurrency)**
  - Because the kernel is preemptive, one task in the kernel can preempt another.
  - Two things do not actually happen at the same time but interleave with each other such that they might as well.

Causes of concurrency

- **Sleeping and synchronization with user-space**
  - A task in the kernel can sleep and thus invoke the scheduler, resulting in the running of a new process.
- **Interrupts**
  - An interrupt can occur asynchronously at almost any time, interrupting the currently executing code.
- **Softirqs and tasklets**
  - The kernel can raise or schedule a softirq or tasklet at almost any time, interrupting the currently executing code.

Concurrency safety

- **SMP-safe**
  - Code that is safe from concurrency on symmetrical multiprocessing machines
- **Preemption-safe**
  - Code that is safe from concurrency with kernel preemption
- **Interrupt-safe**
  - Code that is safe from concurrent access from an interrupt handler

What to protect?

- **Protect data not code**
  - `page_tree` is protected by `tree_lock`

```
/* linux/include/linux/fs.h */

struct inode {          /** metadata of a file */
    umode_t                 i_mode;        /* permission: rwxrw-r-- */
    struct super_block     *i_sb;          /* a file system instance */
    struct address_space *i_mapping;     /* page cache */
};

struct address_space { /** page cache of an inode */
    struct inode            *host;       /* owner: inode, block_device */
    struct radix_tree_root  page_tree;   /* radix tree of all pages
                                          * - i.e., page cache of an inode
                                          * - key:   file offset
                                          * - value: cached page */
    spinlock_t              tree_lock;   /* lock protecting it */
};
```

Questionnaire for locking

- Is the data global?
- Can a thread of execution other than the current one access it?
- Is the data shared between process context and interrupt context?
- Is it shared between two different interrupt handlers?
- If a process is preempted while accessing this data, can the newly scheduled process access the same data?
- If the current process sleep on anything, in what state does that leave any shared data?
- What happens if this function is called again on another processor?

Deadlocks

- Situations in which one or several threads are waiting on locks for one or several resources that will never be freed
  - None of the threads can continue
- **Self-deadlock**
  - **NOTE: Linux does not support recursive locks**

```
acquire lock
acquire lock, again
wait for lock to become available
...
```

Deadlocks

- **Deadly embrace (ABBA deadlock)**

Thread 1	Thread 2
`acquire lock A`	`acquire lock B`
`try to acquire lock B`	`try to acquire lock A`
`wait for lock B`	`wait for lock A`

Deadlock prevention: lock ordering

- Nested locks must *always* be obtained in the *same order*.
    - This prevents the deadly embrace deadlock.

```
/* linux/mm/filemap.c */
/*
 * Lock ordering:
 *
 *  ->i_mmap_rwsem      (truncate_pagecache)
 *    ->private_lock        (__free_pte->__set_page_dirty_buffers)
 *      ->swap_lock     (exclusive_swap_page, others)
 *        ->mapping->tree_lock
 *
 *  ->i_mutex
 *    ->i_mmap_rwsem        (truncate->unmap_mapping_range)
 *  ...
 */
```

Deadlock prevention: lock ordering

```
/* linux/fs/namei.c */
struct dentry *lock_rename(struct dentry *p1, struct dentry *p2)
{
    struct dentry *p;
    if (p1 == p2) {
        inode_lock_nested(p1->d_inode, I_MUTEX_PARENT);
        return NULL;
    }
    mutex_lock(&p1->d_sb->s_vfs_rename_mutex);
    p = d_ancestor(p2, p1);
    if (p) {
        inode_lock_nested(p2->d_inode, I_MUTEX_PARENT);
        inode_lock_nested(p1->d_inode, I_MUTEX_CHILD);
        return p;
    }
    p = d_ancestor(p1, p2);
    if (p) {
        inode_lock_nested(p1->d_inode, I_MUTEX_PARENT);
        inode_lock_nested(p2->d_inode, I_MUTEX_CHILD);
        return p;
    }
    inode_lock_nested(p1->d_inode, I_MUTEX_PARENT);
    inode_lock_nested(p2->d_inode, I_MUTEX_PARENT2);
```

Contention and scalability

- **Lock contention**: a lock currently in use but that another thread is trying to acquire
- **Scalability**: how well a system can be expanded with a large number of processors
- **Coarse- vs fine-grained locking**
  - Coarse-grained lock: bottleneck on high-core count machines
  - Fine-grained lock: overhead on low-core count machines
- **Start simple and grow in complexity only as needed. Simplicity is key.**

Further readings

- Memory-Driven Computing
- Wikipedia: Moore’s Law
- Wikipedia: Amdahl’s Law
- Intel 64 and IA-32 Architectures Software Developer’s Manual
- Intel Xeon Platinum 8180M Processor