CS 5264/4224; ECE 5414/4414
(Advanced) Linux Kernel Programming
Lecture 13

Synchronization

March 18, 2025

Huaicheng Li

https://people.cs.vt.edu/huaicheng/lkp-sp25/

Acknowledgement: Credits to Dr. Changwoo Min for the original LKP lecture slides.

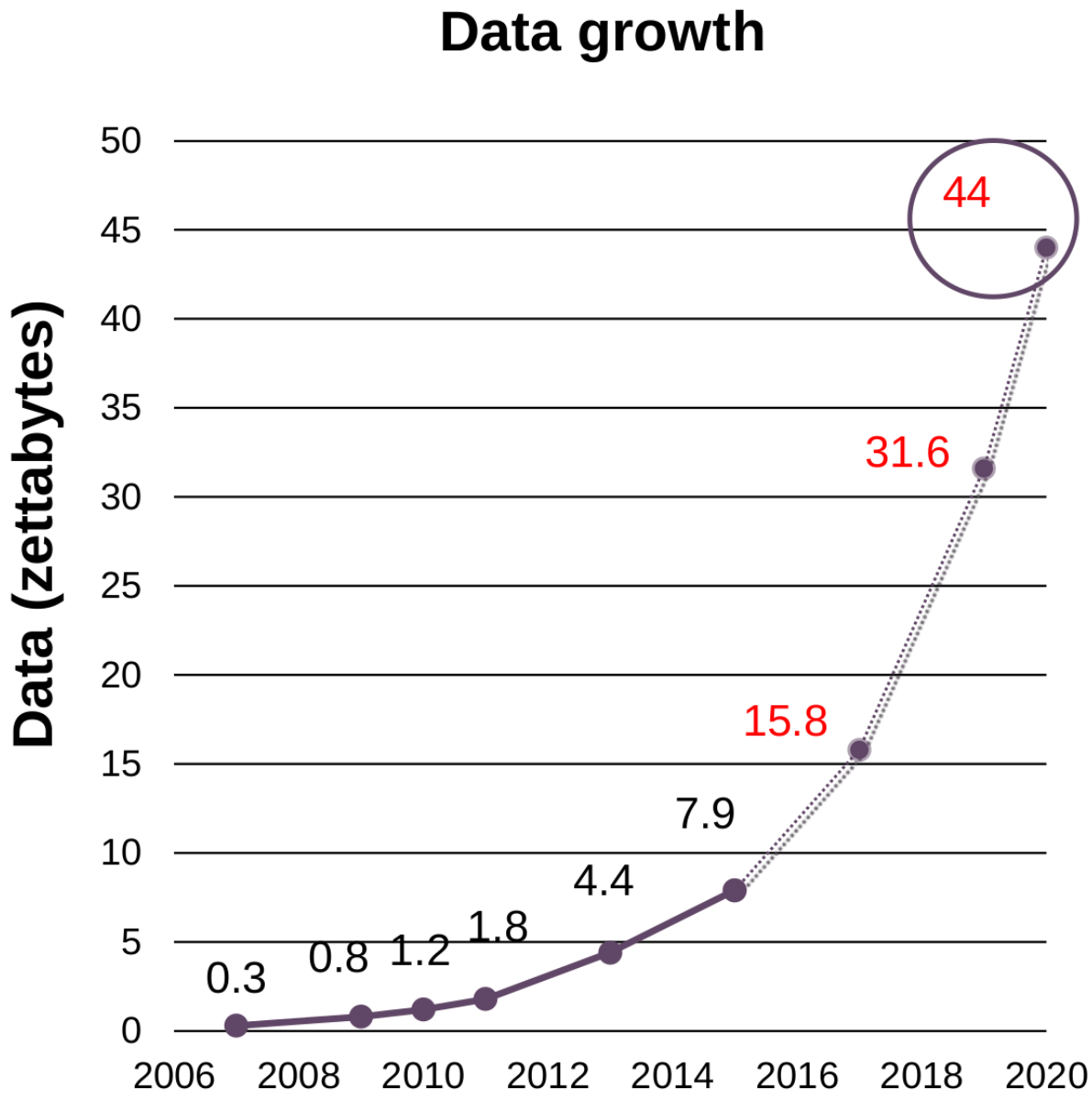
Thus Far ...

- Linux kernel tooling, debugging, code exploration ...
- syscall
- module
- data structure
- scheduling
- interrupts
- time

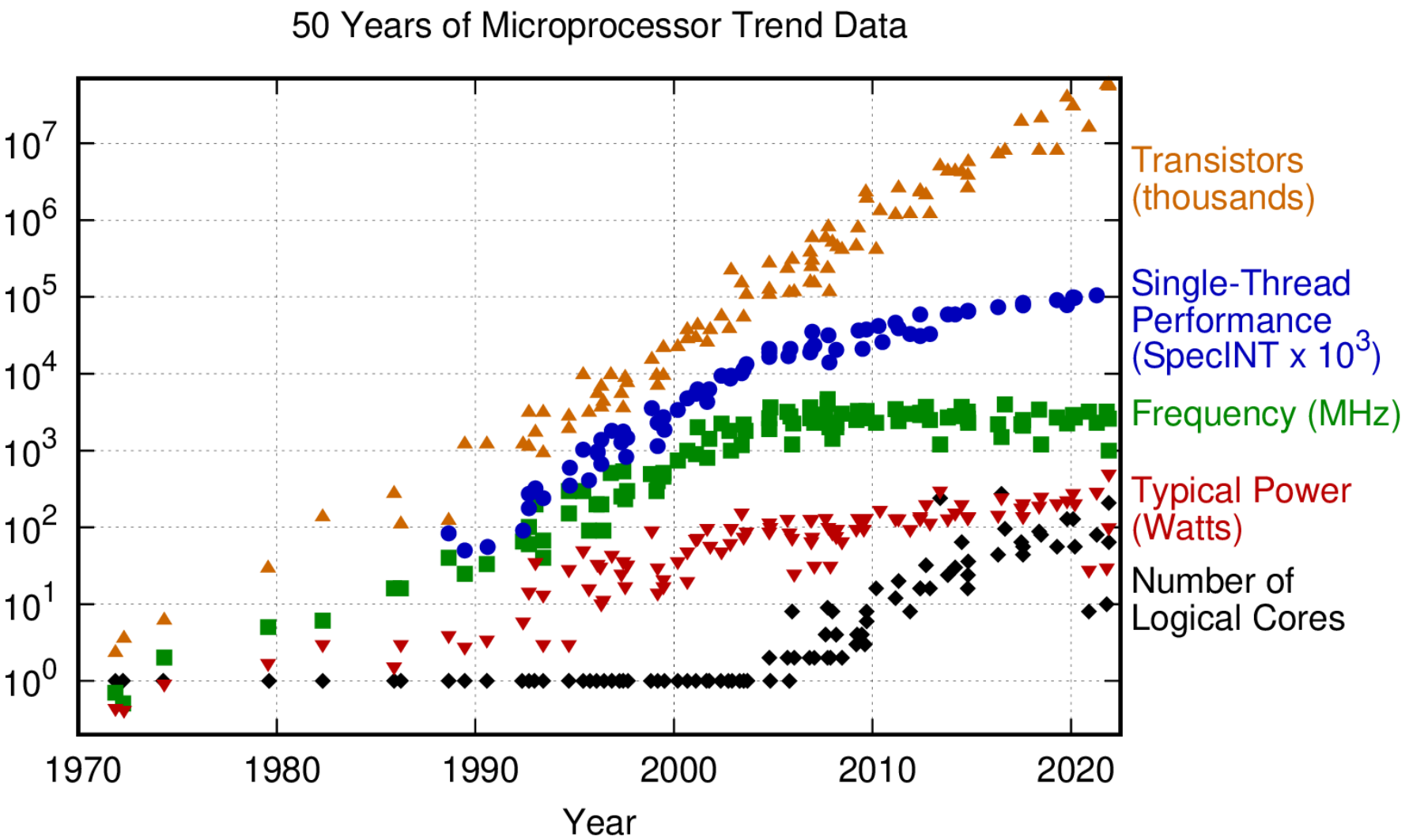
Today's Agenda: Kernel Synchronization

- Multi-core processing
- Introduction to synchronization
- Next lecture
  - synchronization mechanisms in Linux kernel
  - RCU

Exponential Data Growth



Single-core Scaling Stopped



Source: https://github.com/karlrupp/microprocessor-trend-data

- Increasing clock frequency is not possible anymore
  - Power consumption: higher frequency → higher power consumption
  - Wire delay: range of a wire in one clock cycle
- Limitations in Instruction Level Parallelism (ILP)
  - 1980s: more transistors → superscalar → pipeline
    - » 10 CPI (cycles per instruction) → 1 CPI
  - 1990s: multi-way issue, out-of-order (OOO) issue, branch prediction
    - » 1 CPI → 0.5 CPI
- Moore's Law: #transistors doubles approximately every two years
- ~2007: make a single-core processor faster
  - deeper pipeline, branch prediction, ooo, etc.
- 2007~: increase the number of cores in a chip
  - multi-core processor

Ryzen Threadripper PRO 7000 Series | Ryzen Threadripper PRO 5000 Series | Ryzen Threadripper 7000 Series

Name	Graphics Model	# of CPU Cores	# of Threads	Max. Boost Clock	Base Clock	Thermal Solution (PIB)	Default TDP
AMD Ryzen™ Threadripper™ PRO 7995WX	Discrete Graphics Card Required	96	192	Up to 5.1 GHz	2.5 GHz	Not Included	350W
AMD Ryzen™ Threadripper™ PRO 7985WX	Discrete Graphics Card Required	64	128	Up to 5.1 GHz	3.2 GHz	Not Included	350W
AMD Ryzen™ Threadripper™ PRO 7975WX	Discrete Graphics Card Required	32	64	Up to 5.3 GHz	4 GHz	Not Included	350W
AMD Ryzen™ Threadripper™ PRO 7965WX	Discrete Graphics Card Required	24	48	Up to 5.3 GHz	4.2 GHz	Not Included	350W
AMD Ryzen™ Threadripper™ PRO 7955WX	Discrete Graphics Card Required	16	32	Up to 5.3 GHz	4.5 GHz	Not Included	350W
AMD Ryzen™ Threadripper™ PRO 7945WX	Discrete Graphics Card Required	12	24	Up to 5.3 GHz	4.7 GHz	Not Included	350W

Source: here

Name	# of CPU Cores	# of Threads	Max. Boost Clock	Base Clock	L3 Cache	Default TDP
AMD EPYC™ 9965	192	384	Up to 3.7 GHz	2.25 GHz	384 MB	500W
AMD EPYC™ 9845	160	320	Up to 3.7 GHz	2.1 GHz	320 MB	390W
AMD EPYC™ 9825	144	288	Up to 3.7 GHz	2.2 GHz	384 MB	390W
AMD EPYC™ 9755	128	256	Up to 4.1 GHz	2.7 GHz	512 MB	500W
AMD EPYC™ 9745	128	256	Up to 3.7 GHz	2.4 GHz	256 MB	400W
AMD EPYC™ 9655P	96	192	Up to 4.5 GHz	2.6 GHz	384 MB	400W
AMD EPYC™ 9655	96	192	Up to 4.5 GHz	2.6 GHz	384 MB	400W
AMD EPYC™ 9645	96	192	Up to 3.7 GHz	2.3 GHz	256 MB	320W
AMD EPYC™ 9575F	64	128	Up to 5 GHz	3.3 GHz	256 MB	400W
AMD EPYC™ 9565	72	144	Up to 4.3 GHz	3.15 GHz	384 MB	400W
AMD EPYC™ 9555P	64	128	Up to 4.4 GHz	3.2 GHz	256 MB	360W
AMD EPYC™ 9555	64	128	Up to 4.4 GHz	3.2 GHz	256 MB	360W
AMD EPYC™ 9535	64	128	Up to 4.3 GHz	2.4 GHz	256 MB	300W
AMD EPYC™ 9475F	48	96	Up to 4.8 GHz	3.65 GHz	256 MB	400W
AMD EPYC™ 9455P	48	96	Up to 4.4 GHz	3.15 GHz	256 MB	300W
AMD EPYC™ 9455	48	96	Up to 4.4 GHz	3.15 GHz	256 MB	300W

Source: here

Model	Cores/ Threads	Base/Boost (GHz)	TDP	L3 Cache (MB)
Xeon 6980P (GNR)	128 / 256	2.0 / 3.9	500W	504
Xeon 6979P (GNR)	120 / 240	2.1 / 3.9	500W	504

Source: here

The Ampere Roadmap: Powerful Roadmap with Rapid Innovation

Continued Commitment to Leadership Performance Per Rack for AI Compute in Air Cooled Environments

AmpereOne® Family

"Siryn" A1

Up to 192 Cores 5nm
8 Ch DDR5

AmpereOne®

Shipping Now

"Polaris" A2

Up to 192 Cores 5nm
***12 Ch** DDR5*

AmpereOne® "M"

Shipping Q4 '24

"Magnetrix" A2+

Up to 256 Cores 3nm
12 Ch DDR5

AmpereOne® "MX"

In Fabrication

"Aurora" A3

Up to 512 Cores
Integrated AI Silicon
Training and Inference
Air Cooled

AmpereOne® Aurora

Next Design Product

Ampere® Altra® Family

"Mystique" N1+

Up to 80 Cores 7nm
8 Ch DDR4

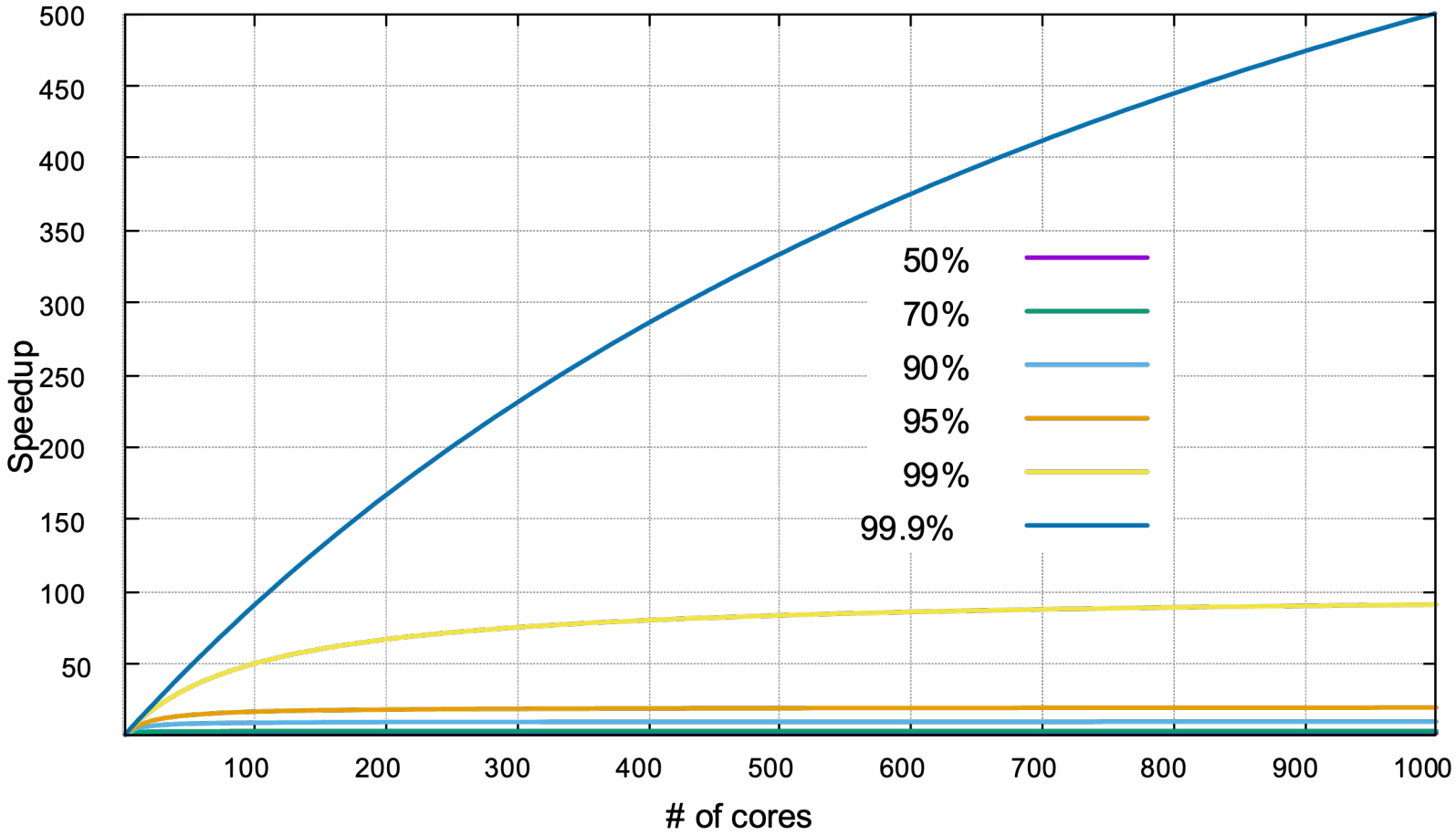
"Quicksilver" N1

Up to 128 Cores 7nm
8 Ch DDR4

Continued Ship Support at Least Through 2030

Source: here

- Amdahl's Law: theoretical speedup of the execution of a task
  - Speedup = 1 / (1 – p + p/n)
  - p: parallel portion of a task
  - n: # of CPU cores



Where are the Sequential Parts?

- Applications: sequential algorithm
- Libraries: memory allocators (buddy structure)
- OS:
  - memory management: VMA (virtual memory area)
  - file system: file descriptor table, journaling
  - network stack: receive queue

The applications may not scale even if its design and implementation are scalable ....

Kernel Synchronization

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

Concurrent Data Access in the Kernel

- Real concurrent access on multiple CPU cores
  - Same as user-space threaded 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?

An Example

```
01: int total = get_total_from_account();      /* total funds in account */
02: int withdrawal = get_withdrawal_amount(); /* amount asked to withdrawal */
03:
04: /* check whether the user has enough funds in her account */
05: if (total < withdrawal) {
06:     error("You do not have that much money!");
07:     return -1;
08: }
09:
10: /* OK, the user has enough money:
11:  * deduct the withdrawal amount from her total */
12: total -= withdrawal;
13: update_total_funds(total);
14:
15: /* give the user their money */
16: spit_out_money(withdrawal);
```

What happens if two transactions are happening nearly at the same time?

e.g., Total = 105
- withdrawl_1 = 100
- withdrawl_2 = 50

One of the transactions should fail b/c (100+50) > 105!

One Possible Incorrect Scenario

- Two threads check that 100 < 105 and 50 < 105 (Line 5)
- Thread 1 updates (Line 13)
  - Total = 105 – 100 = 5
- Thread 2 updates (Line 13)
  - Total = 105 – 50 = 55
- Total withdrawl = 150 but there is 55 left on the account
- Must lock the account during certain operations, make each transaction atomic

Updating a Variable

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

- What happens if two threads concurrently execute foo()?
- What happens if two threads concurrently update i?
- Is incrementing i an atomic operation?

- A single C statement: i++
- Multiple machine instructions
  - (1). get the current value of i and copy it into a register
  - (2). add one to the value stored in the register
  - (3). write back to memory the new value of i
- Now, check what happens if two threads concurrently update i

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)

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)

One Solution: Atomic Instruction

- Atomic operations won't interleave
- Hardware (CPU) guarantees that ...

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)	—

Atomic Instructions in x86

- 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

Volatile Variables ...

- Operations on volatile variables are not atomic
- They won't be optimized by compilers ...

```
/* 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
```

```
/* 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)
```

```
/* 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 to Use "volatile"?

- Memory location can be modified by other entity
  - Other threads from a memory location
  - Other processes for a shared memory location
  - I/O devices for an I/O address

Locking

- Atomic operations are not sufficient for protecting shared data in long and complex critical sections
  - e.g., page_tree of an inode (page cache)
- What is needed is a way to ensure only one thread manipulates the data structure at a time
  - A mechanism for preventing access to a resource while other threads of execution are ”blocked” → 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 */
};
```

- 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

```
== 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
```

Causes of Concurrency

- Symmetrical multiprocessing (true concurrency)
  - 2 or more processors/cores can execute kernel code at exactly the same time
- Kernel preemption (pseudo-concurrency)
  - B/c 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 ...
- Sleeping and synchronization with user-space
  - A task in the kernel can sleep and thus invoke the scheduler to run a new process
- Interrupts
  - An interrupt can occur asynchronously at almost any time, interrupt the currently executing code
- Softirqs, tasklets
  - The kernel can raise or schedule a softirq or tasklet at almost any time, ...

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 */
};
```

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 the 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

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

- Typical deadline

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

```
/* 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)
 *  ...
 */
```

```c
/* 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);
    return NULL;
}
```

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

- Moore's Law, wikipedia
- Amdahl's Law, wikipedia
- Intel SDM