Synchronization 1: Concurrency and Mutual Exclusion

Sam Kumar
CS 162: Operating Systems and System Programming
Lecture 8
https://inst.eecs.berkeley.edu/~cs162/su20

A&D 4.6, 5.1-3
Recall: User/Kernel Threading Models

Almost all current implementations

Simple One-to-One Threading Model

Many-to-One

Many-to-Many
Recall: Pintos Thread

- Single page (4 KiB)
  - Stack growing from the top (high addresses)
  - `struct thread` at the bottom (low addresses)

- `struct thread` defines the TCB structure and PCB structure in Pintos

```
    +-------------------------+    +-------------------------+
    | stack                  |    | Page Table              |
    |                         |    |                         |
    | magic                  |    |   +--------+            |
    | ... <fds>              |    |   | u/s      |            |
    | pagedir                |    |   +--------+            |
    | priority               |    |                         |
    | stack                 |    |                         |
    | name                  |    |                         |
    | status                |    |                         |
    | tid                   |    |                         |

    +-------------------------+    +-------------------------+
    | thread                 |    |                         |
```

Pintos: thread.c
These two threads:
- Are used internally by the kernel
- Don’t correspond to any particular user thread or process
Recall: User Process View of Memory

Process Virtual Address Space

- User code
- User data
- Stack
- Heap
- Kernel

Processor registers:
- sp
- ip

CPL: 3

Physical Memory

Page Table
- Page
- u/s

Virtual Address Space:
- 0xffffffff
- 0xc0000000
- 0x00000000
Recall: Memory Layout

Process Virtual Address Space

Processor registers
- ip
- sp

CPL: 0 - sys

PTBR:

0xffffffff
0xc0000000
0x08048000
0x00000000

kernel
ker data
ker code

argv
stack
heap
user data
user code

Page Table
u/s

Physical Memory
Page
Recall: Running a Program

- Create OS “PCB”, address space, stack and heap
- Load instruction and data segments of executable file into memory
- “Transfer control to program”
- Provide services to program
- While protecting OS and program
Recall: How to \texttt{fork()} efficiently?

- Alias the pages
  - Same physical address!
  - If we stopped here, the data would be shared (not what we want)

- Mark PTEs read-only
  - If a process tries to write → trap to the OS

- On first write to a page after \texttt{fork()}, kernel copies the page, marks PTEs as writeable

- Illusion of separate memory, but really aliased until first write

Pintos doesn’t support \texttt{fork()}, just \texttt{CreateProcess()}
Recall: Reference Counting

- Process 1
  - File Descriptors: 3
    - File: foo.txt
    - Position: 300
    - Reference Count: 2
    - Lock

- Process 2
  - File Descriptors: 3

User Space

Kernel Space

Not shown: Initially contains 0, 1, and 2 (stdin, stdout, stderr)
Before starting synchronization, let’s finish up the previous lecture, “Creating the Process Abstraction.”
How Does the OS Support the Process Abstraction?

- Support for threads and kernel structure
- Memory layout
- Support for process operations
- Support for I/O
- Influence of IPC/RPC on kernel structure
What about `wait()`?

- The parent process needs to get the exit code
- The following events may happen in any order (or concurrently)
  - Parent process calls `wait()` (or `exit()`)  
  - Child process calls `exit()`

- Where should the child put its exit code?
  - Needs to work even if the parent has exited

- Where should the parent search for the exit code?
  - Needs to work even if the child has exited already
How Does the OS Support the Process Abstraction?

• Support for threads and kernel structure
• Memory layout
• Support for process operations
• Support for I/O
• Influence of IPC/RPC on kernel structure
Recall: I/O and Storage Layers

What we’ve covered so far...

- Streams
  - File Descriptors
    - open(), read(), write(), close(), ...
  - Open File Descriptions

What we’ll peek at today

- Files/Directories/Indexes
- Commands and Data Transfers
- Disks, Flash, Controllers, DMA
length = read(input_fd, buffer, BUFFER_SIZE);

ssize_t read(int, void *, size_t){
    marshal args into registers
    issue syscall
    register result of syscall to rtn value
};

void syscall_handler (struct intr_frame *f) {
    unmarshal call#, args from regs
    dispatch : handlers[call#](args)
    marshal results fo syscall ret
}

ssize_t vfs_read(struct file *file, char __user *
*buf, size_t count, loff_t *pos)
{
    User Process/File System relationship
    call device driver to do the work
}

Device Driver
Low-Level Driver

• Associated with particular hardware device
• Registers / Unregisters itself with the kernel
• Handler functions for each of the file operations

```c
struct file_operations {
    struct module *owner;
    loff_t (*llseek)(struct file *, loff_t, int);
    ssize_t (*read)(struct file *, char __user *, size_t, loff_t *);
    ssize_t (*write)(struct file *, const char __user *, size_t, loff_t *);
    ssize_t (*aio_read)(struct kiocb *, const struct iovec *, unsigned long, loff_t);
    ssize_t (*aio_write)(struct kiocb *, const struct iovec *, unsigned long, loff_t);
    int (*readdir)(struct file *, void *, filldir_t);
    unsigned int (*poll)(struct file *, struct poll_table_struct *);
    int (*ioctl)(struct inode *, struct file *, unsigned int, unsigned long);
    int (*mmmap)(struct file *, struct vm_area_struct *);
    int (*open)(struct inode *, struct file *);
    int (*flush)(struct file *, f1_owner_t id);
    int (*release)(struct inode *, struct file *);
    int (*fsync)(struct file *, struct dentry *, int datasync);
    int (*fasync)(int, struct file *, int);
    int (*flock)(struct file *, int, struct file_lock *);
    [...]
};
```
File System: From Syscall to Driver

```c
ssize_t vfs_read(struct file *file, char __user *buf, size_t count, loff_t *pos)
{
    ssize_t ret;
    if (!(file->f_mode & FMODE_READ)) return -EBADF;
    if (!file->f_op || (!file->f_op->read && !file->f_op->aio_read))
        return -EINVAL;
    if (unlikely(!access_ok(VERIFY_WRITE, buf, count))) return -EFAULT;
    ret = rw_verify_area(READ, file, pos, count);
    if (ret >= 0) {
        count = ret;
        if (file->f_op->read)
            ret = file->f_op->read(file, buf, count, pos);
        else
            ret = do_sync_read(file, buf, count, pos);
        if (ret > 0) {
            fsnotify_access(file->f_path.dentry);
            add_rchar(current, ret);
        }
        inc_syscr(current);
    }
    return ret;
}
```

- Read up to “count” bytes from “file” starting from “pos” into “buf”.
- Return error or number of bytes read.

Linux: fs/read_write.c
File System: From Syscall to Driver

```c
ssize_t vfs_read(struct file *file, char __user *buf, size_t count, loff_t *pos) {
    ssize_t ret;
    if (!(file->f_mode & FMODE_READ)) return -EBADF;
    if (!(file->f_op || (!file->f_op->read && !file->f_op->aio_read)))
        return -EINVAL;
    if (unlikely(!access_ok(VERIFY_WRITE, buf, count))) return -EFAULT;
    ret = rw_verify_area(READ, file, pos, count);
    if (ret >= 0) {
        count = ret;
        if (file->f_op->read)
            ret = file->f_op->read(file, buf, count, pos);
        else
            ret = do_sync_read(file, buf, count, pos);
        if (ret > 0) {
            fsnotify_access(file->f_path.dentry);
            add_rchar(current, ret);
        }
        inc_syscr(current);
    }
    return ret;
}
```

Linux: fs/read_write.c

Make sure we are allowed to read this file
ssize_t vfs_read(struct file *file, char __user *buf, size_t count, loff_t *pos)
{
    ssize_t ret;
    if (!(file->f_mode & FMODE_READ)) return -EBADF;
    if (!file->f_op || (!file->f_op->read && !file->f_op->aio_read))
        return -EINVAL;
    if (unlikely(!access_ok(VERIFY_WRITE, buf, count))) return -EFAULT;
    ret = rw_verify_area(READ, file, pos, count);
    if (ret >= 0) {
        count = ret;
        if (file->f_op->read)
            ret = file->f_op->read(file, buf, count, pos);
        else
            ret = do_sync_read(file, buf, count, pos);
        if (ret > 0) {
            fsnotify_access(file->f_path.dentry);
            add_rchar(current, ret);
        }
        inc_syscr(current);
    }
    return ret;
}
ssize_t vfs_read(struct file *file, char __user *buf, size_t count, loff_t *pos) {
    ssize_t ret;
    if (!(file->f_mode & FMODE_READ)) return -EBADF;
    if (!file->f_op || (!file->f_op->read && !file->f_op->aio_read))
        return -EINVAL;
    if (unlikely(!access_ok(VERIFY_WRITE, buf, count))) return -EFAULT;
    ret = rw_verify_area(READ, file, pos, count);
    if (ret >= 0) {
        count = ret;
        if (file->f_op->read)
            ret = file->f_op->read(file, buf, count, pos);
        else
            ret = do_sync_read(file, buf, count, pos);
        if (ret > 0) {
            fsnotify_access(file->f_path.dentry);
            add_rchar(current, ret);
        }
        inc_syscr(current);
    }
    return ret;
}

• Check whether we can write to buf (e.g., buf is in the user space range)
• unlikely(): hint to branch prediction this condition is unlikely
File System: From Syscall to Driver

```c
ssize_t vfs_read(struct file *file, char __user *buf, size_t count, loff_t *pos)
{
    ssize_t ret;
    if (!(file->f_mode & FMODE_READ)) return -EBADF;
    if (!file->f_op || (!file->f_op->read && !file->f_op->aio_read))
        return -EINVAL;
    if (unlikely(!access_ok(VERIFY_WRITE, buf, count))) return -EFAULT;
    ret = rw_verify_area(READ, file, pos, count);
    if (ret >= 0) {
        count = ret;
        if (file->f_op->read)
            ret = file->f_op->read(file, buf, count, pos);
        else
            ret = do_sync_read(file, buf, count, pos);
        if (ret > 0) {
            fsnotify_access(file->f_path.dentry);
            add_rchar(current, ret);
        }
        inc_syscr(current);
    }
    return ret;
}
```

Check whether we read from a valid range in the file.
 ssize_t vfs_read(struct file *file, char __user *buf, size_t count, loff_t *pos) {
    ssize_t ret;
    if (!(file->f_mode & FMODE_READ)) return -EBADF;
    if (!(file->f_op || (!file->f_op->read && !file->f_op->aio_read)))
        return -EINVAL;
    if (unlikely(!access_ok(VERIFY_WRITE, buf, count))) return -EFAULT;
    ret = rw_verify_area(READ, file, pos, count);
    if (ret >= 0) {
        count = ret;
        if (file->f_op->read)
            ret = file->f_op->read(file, buf, count, pos);
        else
            ret = do_sync_read(file, buf, count, pos);
        if (ret > 0) {
            fsnotify_access(file->f_path.dentry);
            add_rchar(current, ret);
        }
        inc_syscr(current);
    }
    return ret;
}
ssize_t vfs_read(struct file *file, char __user *buf, size_t count, loff_t *pos)
{
    ssize_t ret;
    if (!((file->f_mode & FMODE_READ)) return -EBADF;
    if (!file->f_op || (!file->f_op->read && !file->f_op->aio_read))
        return -EINVAL;
    if (unlikely(!access_ok(VERIFY_WRITE, buf, count))) return -EFAULT;
    ret = rw_verify_area(READ, file, pos, count);
    if (ret >= 0) {
        count = ret;
        if (file->f_op->read)
            ret = file->f_op->read(file, buf, count, pos);
        else
            ret = do_sync_read(file, buf, count, pos);
        if (ret > 0) {
            fsnotify_access(file->f_path.dentry);
            add_rchar(current, ret);
        }
        inc_syscr(current);
    }
    return ret;
}

Notify the parent of this file that the file was read (see http://www.fieldses.org/~bfields/kernel/vfs.txt)

Linux: fs/read_write.c
File System: From Syscall to Driver

```c
ssize_t vfs_read(struct file *file, char __user *buf, size_t count, loff_t *pos) {
    ssize_t ret;
    if (!(file->f_mode & FMODE_READ)) return -EBADF;
    if (!file->f_op || (!file->f_op->read && !file->f_op->aio_read))
        return -EINVAL;
    if (unlikely(!access_ok(VERIFY_WRITE, buf, count))) return -EFAULT;
    ret = rw_verify_area(READ, file, pos, count);
    if (ret >= 0) {
        count = ret;
        if (file->f_op->read)
            ret = file->f_op->read(file, buf, count, pos);
        else
            ret = do_sync_read(file, buf, count, pos);
        if (ret > 0) {
            fsnotify_access(file->f_path.dentry);
            add_rchar(current, ret);
        }
        inc_syscr(current);
    }
    return ret;
}
```

Update the number of bytes read by “current” task (for scheduling purposes)

Linux: fs/read_write.c
```c
ssize_t vfs_read(struct file *file, char __user *buf, size_t count, loff_t *pos)
{
    ssize_t ret;
    if (!(file->f_mode & FMODE_READ)) return -EBADF;
    if (!file->f_op || (!file->f_op->read && !file->f_op->aio_read))
        return -EINVAL;
    if (unlikely(!access_ok(VERIFY_WRITE, buf, count))) return -EFAULT;
    ret = rw_verify_area(READ, file, pos, count);
    if (ret >= 0) {
        count = ret;
        if (file->f_op->read)
            ret = file->f_op->read(file, buf, count, pos);
        else
            ret = do_sync_read(file, buf, count, pos);
        if (ret > 0) {
            fsnotify_access(file->f_path.dentry);
            add_rchar(current, ret);
        }
    }
    inc_syscr(current);
    return ret;
}
```

Update the number of read syscalls by “current” task (for scheduling purposes)
Device Drivers

• Device-specific code in the kernel that interacts directly with the device hardware
  • Supports a standard, internal interface
  • Same kernel I/O system can interact easily with different device drivers
  • Special device-specific configuration supported with the ioctl() system call

• Device Drivers typically divided into two pieces:
  • Top half: accessed in call path from system calls
    • implements a set of standard, cross-device calls like open(), close(), read(), write(), ioctl(), strategy()
    • This is the kernel’s interface to the device driver
    • Top half will start I/O to device, may put thread to sleep until finished
  • Bottom half: run as interrupt routine
    • Gets input or transfers next block of output
    • May wake sleeping threads if I/O now complete
Recall: Inter-Process Communication (IPC)

• Mechanism to create communication channel between distinct processes
  • Same or different machines, same or different programming language...

• Requires serialization format understood by both

• Failure in one process isolated from the other
  • Sharing is done in a controlled way through IPC
  • Still have to be careful handling what is received via IPC

• Many uses and interaction patterns
  • Logging process, window management, ...
  • Potentially allows us to move some system functions outside of kernel to userspace
Today: How Does the OS Support the Process Abstraction?

• Support for threads and kernel structure
• Memory layout
• Support for process operations
• Support for I/O
• Influence of IPC/RPC on kernel structure
Recall: Using IPC to Simplify OS

• What if the file system is not local to the machine, but on the network?
• Is there a general mechanism for providing services to other processes?
  • Do the protocols we run on top of IPC generalize as well?
Microkernels

• Split OS into **separate processes**
  • Example: File System, Network Driver are processes outside of the kernel
• Pass messages among these components (e.g., via RPC) instead of system calls
Microkernels

• Microkernel itself provides only essential services
  • Communication
  • Address space management
  • Thread scheduling
  • Almost-direct access to hardware devices (for driver processes)
Why Microkernels?

**Pros**
- Failure Isolation
- Easier to update/replace parts
- Easier to distribute – build one OS that encompasses multiple machines

**Cons**
- More communication overhead and context switching
- Harder to implement?
Flashback: What is an OS?

• Always:
  • Memory Management
  • I/O Management
  • CPU Scheduling
  • Communications
  • Multitasking/multiprogramming

• Maybe:
  • File System?
  • Multimedia Support?
  • User Interface?
  • Web Browser?

Not provided in a strict microkernel
Influence of Microkernels

• Many operating systems provide some services externally, similar to a microkernel
  • OS X and Linux: Windowing (graphics and UI)

• Some currently monolithic OSes started as microkernels
  • Windows family originally had microkernel design
  • OS X: Hybrid of Mach microkernel and FreeBSD monolithic kernel
Operating System Archaeology

• Because of the cost of developing an OS from scratch, most modern OSes have a long lineage:

• Multics → AT&T Unix → BSD Unix → Ultrix, SunOS, NetBSD,…

• Mach (micro-kernel) + BSD → NextStep → XNU → Apple OS X, iPhone iOS

• Linux → Android OS

• CP/M → QDOS → MS-DOS → Windows 3.1 → NT → 95 → 98 → 2000 → XP → Vista → 7 → 8 → phone → …

• Linux → RedHat, Ubuntu, Fedora, Debian, Suse,…
Bonus Material (If Time)
Don’t fork() in a process that already has multiple threads

Unless you plan to call exec() in the child process
fork() in Multithreaded Processes

• The child process always has just a single thread
  • The thread in which fork() was called

• The other threads just vanish
fork() in a Multithreaded Processes

- Only the thread that called fork() exists in the new process

Not shown: Initially contains 0, 1, and 2 (stdin, stdout, stderr)
Possible Problems with Multithreaded fork()

• When you call fork() in a multithreaded process, the other threads (the ones that didn’t call fork()) just vanish
  • What if one of these threads was holding a lock?
  • What if one of these threads was in the middle of modifying a data structure?
  • No cleanup happens!

• It’s safe if you call exec() in the child
  • Replacing the entire address space
Don’t carelessly mix low-level and high-level file I/O
Avoid Mixing FILE* and File Descriptors

char x[10];
char y[10];
FILE* f = fopen("foo.txt", "rb");
int fd = fileno(f);
fread(x, 10, 1, f); // read 10 bytes from f
read(fd, y, 10); // assumes that this returns 10

• Which bytes from the file are read into y?
  A. Bytes 0 to 9
  B. Bytes 10 to 19
  C. None of these?
Avoid Mixing FILE* and File Descriptors

```c
char x[10];
char y[10];
FILE* f = fopen("foo.txt", "rb");
int fd =fileno(f);
fread(x, 10, 1, f); // read 10 bytes from f
read(fd, y, 10); // assumes that this returns 10
```

• Which bytes from the file are read into y?
  A. Bytes 0 to 9
  B. Bytes 10 to 19
  C. None of these?
Be careful with \texttt{fork()} and \texttt{FILE*}
Be Careful Using `fork()` with FILE*

```c
FILE* f = fopen("foo.txt", "w");
fwrite("a", 1, 1, f);
fork();
fclose(f);
```

After all processes exit, what is in `foo.txt`?

Could be `aa`

- Depends on whether this `fwrite` call flushes...
Be Careful Using `fork()` with `FILE*`

Process 1

- User Space
- Thread’s Regs
- FILE* Buffer
- File Descriptors 3
- File: foo.txt
- Position: 0

Process 2

- User Space
- Thread’s Regs
- FILE* Buffer
- File Descriptors 3
- Open File Description
- File: foo.txt
- Position: 0

- Kernel Space

- Open File Description is aliased
- But the FILE* buffer is copied!

Not shown: Initially contains 0, 1, and 2 (stdin, stdout, stderr)
Announcements

• Project 0 due tonight

• **Drop deadline (with refund) tonight!**

• Homework 2 due Monday

• Quiz 1 on Monday
  • Covers material up to this point

• Project 1 Design Doc due Monday
  • Design reviews with TAs on Tuesday
Recall: What Threads Are

• Definition from before: A *single unique execution context*
  • Describes its representation

• It provides the abstraction of: A *single execution sequence that represents a separately schedulable task*
  • Also a valid definition!

• Threads are a mechanism for *concurrency*

• Protection is an orthogonal concept
  • A protection domain can contain one thread or many
Recall: Motivation for Threads

- Operating systems must handle multiple things at once (MTAO)
  - Processes, interrupts, background system maintenance
- Networked servers must handle MTAO
  - Multiple connections handled simultaneously
- Parallel programs must handle MTAO
  - To achieve better performance
- Programs with user interface often must handle MTAO
  - To achieve user responsiveness while doing computation
- Network and disk bound programs must handle MTAO
  - To hide network/disk latency
  - Sequence steps in access or communication
Processes vs. Threads

- **Switch overhead:**
  - Same process: low
  - Different proc.: high

- **Protection**
  - Same proc: low
  - Different proc: high

- **Sharing overhead**
  - Same proc: low
  - Different proc: high
Processes vs. Threads

- **Switch overhead:**
  - Same process: **low**
  - Different proc.: **high**

- **Protection**
  - Same proc: **low**
  - Different proc: **high**

- **Sharing overhead**
  - Same proc: **low**
  - Different proc: **high**
## Classification

<table>
<thead>
<tr>
<th># threads Per AS:</th>
<th># of addr spaces:</th>
<th>One</th>
<th>Many</th>
</tr>
</thead>
<tbody>
<tr>
<td>One</td>
<td></td>
<td>MS/DOS, early Macintosh</td>
<td>Traditional UNIX</td>
</tr>
<tr>
<td>Many</td>
<td></td>
<td>Embedded systems (Geoworks, VxWorks, JavaOS, etc) JavaOS, Pilot(PC)</td>
<td>Mach, OS/2, Linux Windows 10 Win NT to XP, Solaris, HP-UX, OS X</td>
</tr>
</tbody>
</table>
How does the OS implement concurrency?
Stack for Yielding Thread

- How do we run a new thread?
  ```c
  run_new_thread() {
    newThread = PickNewThread();
    switch(curThread, newThread);
    ThreadHouseKeeping(); /* Do any cleanup */
  }
  ```
- How does dispatcher switch to a new thread?
  - Save anything next thread may trash: PC, regs, stack pointer
  - Maintain isolation for each thread
Switching Threads

- Consider the following code blocks:
  ```
  func A() {
    B();
  }
  func B() {
    while(TRUE) {
      yield();
    }
  }
  ```

- Two threads, S and T, each run A

Thread S's switch returns to Thread T's (and vice versa)

Pintos: switch.S
Recall: Illusion of Multiple Processors

- At T1: vCPU1 on real core
- At T2: vCPU2 on real core

- How did the OS get to run?
  - Earlier, OS configured a hardware timer to periodically generate an interrupt
  - On the interrupt, the hardware switches to kernel mode and the OS’s timer interrupt handler runs
  - Timer interrupt handler decides whether to switch threads or not **according to a policy**
Interrupt Management

- Interrupt controller chooses interrupt request to honor
  - Interrupt identity specified with ID line
  - Software Interrupt Set/Cleared by Software
- CPU can disable all interrupts with internal flag
- Non-Maskable Interrupt line (NMI) can’t be disabled
Example: Network Interrupt

External Interrupt

Pipeline Flush

lw $r2,0($r4)
lw $r3,4($r4)
add $r2,$r2,$r3
sw 8($r4),$r2
...

raise priority (set mask)
Reenable All Ints
Save registers
Dispatch to Handler...
Transfer Network Packet from hardware to Kernel Buffers...
Restore registers
Clear current Int
Disable All Ints
Restore priority (clear Mask)
RTI

"Interrupt Handler"

PC Saved
Disable All Ints
Kernel Mode

"Interrupt Handler"

Enable All Ints
Disable All Ints
User Mode
Preempting a Thread

• Timer Interrupt routine:

```c
TimerInterrupt()
{
    DoPeriodicHouseKeeping();
    run_new_thread();
}
```
Creating a New Thread

- Let \texttt{ThreadRoot} be the routine that the thread should start out running.
- We need to set up the thread state so that, another thread can “return” into the beginning of \texttt{ThreadRoot}.
  - This really starts the new thread.
Bootstrapping Threads

ThreadRoot() {
    DoStartupHousekeeping();
    UserModeSwitch(); /* enter user mode */
    call fcnPtr(fcnArgPtr);
    ThreadFinish();
}

• Stack will grow and shrink with execution of thread
• ThreadRoot() never returns
  • ThreadFinish() destroys thread, invokes scheduler
Aside: SMT/Hyperthreading

- Hardware technique
  - Superscalar processors try to execute multiple independent instructions in parallel
  - Hyperthreading allows a single core to process multiple instructions streams at once
  - But, sub-linear speedup

- Original called “Simultaneous Multithreading”
  - Intel, SPARC, Power (IBM)

- From the OS perspective, this just looks like multiple cores
Aside: SMT/Hyperthreading

- Switch overhead:
  - Same process: low
  - Different proc.: high
- Protection
  - Same proc: low
  - Different proc: high
- Sharing overhead
  - Same proc: low
  - Different proc: high

Process 1

- threads
- Mem.
- IO state
- CPU state
- CPU state

Process N

- threads
- Mem.
- IO state
- CPU state
- CPU state

CPU sched.

OS

8 threads at a time

Core 1
Core 2
Core 3
Core 4

8 threads at a time
Recall: Race Conditions

• What are the possible values of x below?
• Initially x == 0 and y == 0

Thread A      Thread B
x = y + 1;  y = 2;
y = y * 2;

• 1 or 3 or 5 (non-deterministic)
• Race Condition: Thread A races against Thread B
Recall: Relevant Definitions

• Synchronization: Coordination among threads, usually regarding shared data

• Mutual Exclusion: Ensuring only one thread does a particular thing at a time (one thread excludes the others)
  • Type of synchronization

• Critical Section: Code exactly one thread can execute at once
  • Result of mutual exclusion

• Lock: An object only one thread can hold at a time
  • Provides mutual exclusion
Recall: Locks

• Locks provide two **atomic** operations:
  • Lock.acquire() – wait until lock is free; then mark it as busy
    • After this returns, we say the calling thread *holds* the lock
  • Lock.release() – mark lock as free
    • Should only be called by a thread that currently holds the lock
    • After this returns, the calling thread no longer holds the lock

• Provides *mutual exclusion* between two or more threads
Mutual Exclusion between Thread and Interrupt Handler

• Interrupt handler runs to completion
• Can’t acquire a lock in an interrupt handler (why?)

• Solution: Disable interrupts and restore them afterwards

```c
int state = intr_disable();
<code manipulating shared data>
intr_restore(state);
```
How to implement mutual exclusion?

For now, just consider locks *inside the kernel*.
Recall: Race Conditions

• What are the possible values of x below?
• Initially x == 0

```
Thread A       Thread B
x += 1;        x += 1;
```

• 1 or 2 (non-deterministic)
Recall: Race Conditions

• What are the possible values of $x$ below?
  • Initially $x == 0$

Thread A         Thread B
$x = 1;$          $x = 2;$

• 1 or 2 (non-deterministic)
• Maybe even 3 for serial processors!
Atomic Operations

• To understand a concurrent program, we need to know what the underlying indivisible operations are!

• **Atomic Operation**: an operation that always runs to completion or not at all
  • It is *indivisible*: it cannot be stopped in the middle and state cannot be modified by someone else in the middle
  • Fundamental building block – if no atomic operations, then have no way for threads to work together

• On most machines, memory references and assignments (i.e. loads and stores) of words are atomic
  • Consequently – weird example that produces “3” on previous slide can’t happen
Concurrency is Hard!

- Even for practicing engineers trying to write mission-critical, bulletproof code!
  - Threaded programs must work for all interleavings of thread instruction sequences
  - Cooperating threads inherently non-deterministic and non-reproducible
  - Really hard to debug unless carefully designed!

- Therac-25: Radiation Therapy Machine with Unintended Overdoses (reading on course site)
- Mars Pathfinder Priority Inversion (JPL Account)
- Toyota Uncontrolled Acceleration (CMU Talk)
  - 256.6K Lines of C Code, ~9-11K global variables
  - Inconsistent mutual exclusion on reads/writes
Motivating Example: “Too Much Milk”

• Analogy between problems in OS and problems in real life
• Example: People need to coordinate:

<table>
<thead>
<tr>
<th>Time</th>
<th>Person A</th>
<th>Person B</th>
</tr>
</thead>
<tbody>
<tr>
<td>3:00</td>
<td>Look in Fridge. Out of milk</td>
<td></td>
</tr>
<tr>
<td>3:05</td>
<td>Leave for store</td>
<td></td>
</tr>
<tr>
<td>3:10</td>
<td>Arrive at store</td>
<td>Look in Fridge. Out of milk</td>
</tr>
<tr>
<td>3:15</td>
<td>Buy milk</td>
<td>Leave for store</td>
</tr>
<tr>
<td>3:20</td>
<td>Arrive home, put milk away</td>
<td>Arrive at store</td>
</tr>
<tr>
<td>3:25</td>
<td></td>
<td>Buy milk</td>
</tr>
<tr>
<td>3:30</td>
<td></td>
<td>Arrive home, put milk away</td>
</tr>
</tbody>
</table>
Too Much Milk: Correctness

1. Safety: At most one person buys milk.

2. Liveness: If milk is needed, at least one person buys it.
Attempt #1

• Leave a note
  • Place on fridge before buying
  • Remove after buying
  • Don’t go to store if there’s already a note

• Leaving/checking a note is atomic (word load/store)

```java
if (noMilk) {
    if (noNote) {
        leave Note;
        buy milk;
        remove Note;
    }
}
```
Attempt #1 in Action

Thread A
if (noMilk) {
  if (noNote) {
    leave Note;
    buy milk;
    remove Note;
  }
}

Thread B
if (noMilk) {
  if (noNote) {
    leave Note;
    buy milk;
    remove Note;
  }
}

Achieves liveness but not safety
Attempt #1.5

• Idea: leave note, then check for milk

```java
leave Note;
if (noMilk) {
  if (noNote) {
    leave Note;
    buy milk;
  }
}
remove Note;
```

But there’s always a note – you just left one!
Attempt #2: Use Named Notes

Thread A
leave note A
if (noMilk) {
  if (noNote B) {
    buy milk
  }
}
remove note A

Thread B
leave note B
if (noMilk) {
  if (noNote A) {
    buy milk
  }
}
remove note B
Attempt #2 in Action

Thread A
leave note A
if (noMilk) {
    if (noNote B) {
        buy milk
    }
}
remove note A

Thread B
leave note B

Achieves safety but not liveness

if (noMilk) {
    if (noNote A) {
        buy milk
    }
    remove note B
Attempt #3: Wait

Thread A
leave note A
while (note B) {
    do nothing
}
if (noMilk) {
    buy milk
}
remove note A

Thread B
leave note B
if (noNote A) {
    if (noMilk) {
        buy milk
    }
}
remove note B

This is a correct solution!
This Generalizes to $n$ Threads...

- Leslie Lamport’s “Bakery Algorithm” (1974)

- Allows one to protect a critical section like:

```java
if (noMilk) {
    buy milk;
}
```
Solution #3 Discussion

• Solution #3 works, but it’s not great
  • Really complex – even for this simple an example
    • Hard to convince yourself that this really works
  • While A is waiting, it is consuming CPU time
    • This is called “busy-waiting”

• There’s a better way
  • Have hardware provide higher-level primitives than atomic load & store
  • Build even higher-level programming abstractions on this hardware support
  • Make sure the OS scheduler never allows another thread to enter the critical section
    • The other thread becomes blocked if it tries to enter
Where are we going with Synchronization?

- Building an efficient, easy-to-use API
Implementing Locks: Single Core

• How can we make lock.Acquire() and lock.Release() appear atomic to other threads?

• Idea: A context switch can only happen (assuming threads don’t yield) if there’s an interrupt

• “Solution”: Disable interrupts while holding lock

• x86 has cli and sti instructions that only operate in system mode (PL=0)
  • Interrupts enabled bit in FLAGS register
Naïve Interrupt Enable/Disable

```plaintext
Acquire() {
    disable interrupts;
}

Release() {
    enable interrupts;
}
```

• Problem: can stall the entire system
  
  Lock.Acquire()
  While (1) {} 

• Problem: What if we want to do I/O?
  
  Lock.Acquire()
  Read from disk
  /* OS waits for (disabled) interrupt)! */
Conclusion

• We saw how device drivers fit into the OS

• We saw how the OS implements concurrency

• We saw how we can implement mutual exclusion with atomic loads/stores

• We motivated how we might implement a lock more efficiently