Synchronization 1: Concurrency and Mutual Exclusion

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

Read: 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
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

```
<table>
<thead>
<tr>
<th>Address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xffffffff</td>
<td>Kernel space</td>
</tr>
<tr>
<td>0xc00000000</td>
<td>Argv space</td>
</tr>
<tr>
<td>0x00000000</td>
<td>User code space</td>
</tr>
<tr>
<td>0x08048000</td>
<td>User data space</td>
</tr>
</tbody>
</table>
```

CPL: 3 - user

Processor registers:
- `sp`
- `ip`

Page Table:
- u/s

Physical Memory

---

7/2/2020

Kumar CS 162 at UC Berkeley, Summer 2020
Recall: Memory Layout

Process Virtual Address Space

Processor registers

- sp
- ip

CPL:
- 0 - sys

PTBR:

Physical Memory

0xffffffff

kernel
ker data
ker code

argv
stack

heap
user data

user code

0x08048000
0x00000000

Page Table

Page

u/s

u/s
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

```c
int main()
{
    ...;
}
```

Program Source: foo.c

Compiler and Linker:
- Compile to a.out
- Link to executable

OS Loader:
- Load OS, stack, heap, data, instructions
- Load into memory
- Execute: `exec(...)`

Processor:
- PC: 0x000...
- Registers

Memory:
- 0xFFFF...

Executable:
- data
- instructions
- a.out

OS Loader Diagram:
- OS
- stack
- heap
- data
- instructions
- 0x000...
- 0xFFFF...
- PC: 0x000...
- Processor

7/2/2020
Kumar CS 162 at UC Berkeley, Summer 2020
Recall: How to `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 `fork()`, kernel copies the page, marks PTEs as writeable

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

Pintos doesn’t support `fork()`, just `CreateProcess()`
Recall: Reference Counting

User Space

Kernel Space

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

File Descriptors

3

Open File Description

File: foo.txt
Position: 300
Reference Count: 2
Lock

File Descriptors

3
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

Application / Service

High Level I/O

Streams

Low Level I/O

File Descriptors

Syscall

open(), read(), write(), close(), ...

File System

Open File Descriptions

I/O Driver

Commands and Data Transfers

Disks, Flash, Controllers, DMA

What we’ve covered so far...

What we’ll peek at today
length = \texttt{read}(\texttt{input\_fd}, \texttt{buffer}, \texttt{BUFFER\_SIZE});

```c
ssize_t \texttt{read}(int, void *, size_t) {
    // marshaling
    \texttt{marshal} \texttt{args} \texttt{into registers}
    \texttt{issue} \texttt{syscall}
    \texttt{register \ result of syscall to \ rtn value}
}
```

```c
\texttt{Exception U\rightarrow K, interrupt processing}
void \texttt{syscall\_handler} (\texttt{struct intr\_frame *} \texttt{f}) {
    \texttt{unmarshall} \texttt{call\#}, \texttt{args} \texttt{from \ regs}
    \texttt{dispatch: handlers}[\texttt{call\#}](\texttt{args})
    \texttt{marshal results to syscall ret}
}
```

```c
ssize_t \texttt{vfs\_read}(\texttt{struct file *} \texttt{file}, char \_user *
\texttt{buf}, size_t \texttt{count}, loff_t *
\texttt{pos})
{
    \texttt{User Process/File System relationship}
    \texttt{call device driver to do the work}
}
```

**Layers...**

- **User App**
- **User library**
- **File System**
- **I/O Driver**
- **Syscall**
- **Low Level I/O**
- **High Level I/O**

**Application / Service**

- **User App**
- **User library**
- **Device Driver**
- **I/O Driver**
- **File System**
- **Syscall**
- **Low Level I/O**
- **High Level I/O**
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 (*fchdir) (struct file *, struct file_id);
    int (*release) (struct inode *, struct file *);
    int (*fsync) (struct file *, struct dentry *, int data_sync);
    int (*fasync) (int, struct file *, int);
    int (*flock) (struct file *, int, struct file_lock *);
    [...]};
```
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;
}

Make sure we are allowed to read this file

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;
}
```

Check if file has read methods

Linux: fs/read_write.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;
}
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.
```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;
}
```

If driver provide a read function (f_op->read) use it; otherwise use do_sync_read()
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)
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)
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
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: 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
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 $\rightarrow$ AT&T Unix $\rightarrow$ BSD Unix $\rightarrow$ Ultrix, SunOS, NetBSD,…

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

• Linux $\rightarrow$ Android OS

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

• Linux $\rightarrow$ 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

User Space

Kernel Space

Process 1

Thread 1
Regs

Thread 2
Regs

Address Space (Memory)

File Descriptors

3

Open File Description

File: foo.txt
Position: 100

Process 2

Thread 1
Regs

Address Space (Memory)

File Descriptors

3

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

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

```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?
Avoid Mixing FILE* and File Descriptors

char x[10];
char y[10];
FILE* f = fopen("foo.txt", "rb");
int fd = fileno(f);
fwrite(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 `fork()` and `FILE*`
Be Careful Using \texttt{fork()} with \texttt{FILE*}

\begin{verbatim}
FILE* f = fopen("foo.txt", "w");
fwrite("a", 1, 1, f);
fork();
fclose(f);
\end{verbatim}

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

- Thread’s Regs
- `FILE*` Buffer: a
- File Descriptors: 3

Process 2

- Thread’s Regs
- `FILE*` Buffer: a
- Open File Description
  - `FILE`: foo.txt
  - Position: 0

User Space

Kernel Space

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

- Open File Description is aliased
- But the `FILE*` buffer is copied!
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
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></td>
<td></td>
<td>MS/DOS, early Macintosh</td>
<td>Traditional UNIX</td>
</tr>
<tr>
<td>One</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Many</td>
<td></td>
<td>Embedded systems (Geoworks, VxWorks, JavaOS, etc)</td>
<td>Mach, OS/2, Linux Windows 10 Win NT to XP, Solaris, HP-UX, OS X</td>
</tr>
<tr>
<td></td>
<td></td>
<td>JavaOS, Pilot(PC)</td>
<td></td>
</tr>
</tbody>
</table>
How does the OS implement concurrency?
Stack for Yielding Thread

• How do we run a new thread?
  ```
  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:
  ```go
  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)
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

- add $r1,$r2,$r3
- subi $r4,$r1,#4
- slli $r4,$r4,#2

... 

PC saved

- Disable All Ints
- Kernel Mode
- 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"

Restore PC

- Enable All Ints
- User Mode

7/2/2020
Preempting a Thread

• Timer Interrupt routine:

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

- Let 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 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”
  • http://www.cs.washington.edu/research/smt/index.html
  • Intel, SPARC, Power (IBM)

• From the OS perspective, this just looks like multiple cores

Colored blocks show instructions executed
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
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);
```
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