Acknowledgments: Lecture slides are from the Operating Systems course taught by John Kubiatowicz at Berkeley, with few minor updates/changes. When slides are obtained from other sources, a reference will be noted on the bottom of that slide, in which case a full list of references is provided on the last slide.
Recall: Lifecycle of a Process

- As a process executes, it changes state:
  - **new**: The process is being created
  - **ready**: The process is waiting to run
  - **running**: Instructions are being executed
  - **waiting**: Process waiting for some event to occur
  - **terminated**: The process has finished execution
Recall: Use of Threads

• Version of program with Threads (loose syntax):

```
main() {
    ThreadFork(ComputePI("pi.txt"));
    ThreadFork(PrintClassList("clist.txt"));
}
```

• What does “ThreadFork()” do?
  - Start independent thread running given procedure

• What is the behavior here?
  - Now, you would actually see the class list
  - This should behave as if there are two separate CPUs

<table>
<thead>
<tr>
<th>CPU1</th>
<th>CPU2</th>
<th>CPU1</th>
<th>CPU2</th>
<th>CPU1</th>
<th>CPU2</th>
</tr>
</thead>
</table>

Time
Recall: Thread Abstraction

- Infinite number of processors
- Threads execute with variable speed
  - Programs must be designed to work with any schedule
Recall: Execution Stack Example

A(int tmp) {
    if (tmp<2)
        B();
    printf(tmp);
}
B() {
    C();
}
C() {
    A(2);
}
A(1);

- Stack holds temporary results
- Permits recursive execution
- Crucial to modern languages

Stack Pointer

Stack Growth

A: tmp=1
   ret=exit
B: ret=A+2
C: ret=b+1
A: tmp=2
   ret=C+1
### MIPS: Software conventions for Registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>zero constant 0</td>
</tr>
<tr>
<td>1</td>
<td>at reserved for assembler</td>
</tr>
<tr>
<td>2</td>
<td>v0 expression evaluation &amp;</td>
</tr>
<tr>
<td>3</td>
<td>v1 function results</td>
</tr>
<tr>
<td>4</td>
<td>a0 arguments</td>
</tr>
<tr>
<td>5</td>
<td>a1</td>
</tr>
<tr>
<td>6</td>
<td>a2</td>
</tr>
<tr>
<td>7</td>
<td>a3</td>
</tr>
<tr>
<td>8</td>
<td>t0 temporary: caller saves</td>
</tr>
<tr>
<td>9</td>
<td>(callee can clobber)</td>
</tr>
<tr>
<td>10</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>t7</td>
</tr>
<tr>
<td>16</td>
<td>s0 callee saves</td>
</tr>
<tr>
<td>17</td>
<td>. . . (callee must save)</td>
</tr>
<tr>
<td>18</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td></td>
</tr>
<tr>
<td>21</td>
<td></td>
</tr>
<tr>
<td>22</td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>s7</td>
</tr>
<tr>
<td>24</td>
<td>t8 temporary (cont’d)</td>
</tr>
<tr>
<td>25</td>
<td>t9</td>
</tr>
<tr>
<td>26</td>
<td>k0 reserved for OS kernel</td>
</tr>
<tr>
<td>27</td>
<td>k1</td>
</tr>
<tr>
<td>28</td>
<td>gp Pointer to global area</td>
</tr>
<tr>
<td>29</td>
<td>sp Stack pointer</td>
</tr>
<tr>
<td>30</td>
<td>fp frame pointer</td>
</tr>
<tr>
<td>31</td>
<td>ra Return Address (HW)</td>
</tr>
</tbody>
</table>

**Before calling procedure:**
- Save caller-saves regs
- Save v0, v1
- Save ra

**After return, assume**
- Callee-saves reg OK
- gp,sp,fp OK (restored!)
- Other things trashed
Recall: Multithreaded stack switching

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

- Suppose we have 2 threads:
  - Threads S and T
Use of Timer Interrupt to Return Control

- Solution to our dispatcher problem
  - Use the timer interrupt to force scheduling decisions

- Timer Interrupt routine:
  
  ```c
  void TimerInterrupt() {
    DoPeriodicHouseKeeping();
    run_new_thread();
  }
  ```

- I/O interrupt: same as timer interrupt except that DoHousekeeping() replaced by ServiceIO().
How does Thread get started?

Eventually, `run_new_thread()` will select this TCB and return into beginning of `ThreadRoot()`.
- This really starts the new thread.
What does ThreadRoot() look like?

- ThreadRoot() is the root for the thread routine:

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

- Startup Housekeeping
  - Includes things like recording start time of thread
  - Other Statistics

- Stack will grow and shrink with execution of thread

- Final return from thread returns into ThreadRoot() which calls ThreadFinish()
  - ThreadFinish() wake up sleeping threads
Examples multithreaded programs

- **Embedded systems**
  - Elevators, Planes, Medical systems, Wristwatches
  - Single Program, concurrent operations

- **Most modern OS kernels**
  - Internally concurrent because have to deal with concurrent requests by multiple users
  - But no protection needed within kernel

- **Database Servers**
  - Access to shared data by many concurrent users
  - Also background utility processing must be done
Example multithreaded programs (con't)

• Network Servers
  - Concurrent requests from network
  - Again, single program, multiple concurrent operations
  - File server, Web server, and airline reservation systems

• Parallel Programming (More than one physical CPU)
  - Split program into multiple threads for parallelism
  - This is called Multiprocessing
A typical use case

Client Browser
- process for each tab
- thread to render page
- GET in separate thread
- multiple outstanding GETs
- as they complete, render portion

Web Server
- fork process for each client connection
- thread to get request and issue response
- fork threads to read data, access DB, etc
- join and respond
Some Numbers

- Many process are multi-threaded, so thread context switches may be either **within-process** or **across-processes**.
Kernel Use Cases

• Thread for each user process
• Thread for sequence of steps in processing I/O
• Threads for device drivers
• ...
Administrivia

• Group formation: should be completed by tonight!
  - Will handle stragglers tonight

• Group HW #1: Released!
  - Starts today
  - All design reviews will be conducted by TAs

• HW1 due Thursday
  - Must be submitted via the recommended “push” mechanism through git
  - “commit as you make progress” is essential!
Dennis Richie, Unix V6, slp.c:

```
2230  /* If the new process paused because it was
2231  * swapped out, set the stack level to the last call
2232  * to savu(u_ssav). This means that the return
2233  * which is executed immediately after the call to aretu
2234  * actually returns from the last routine which did
2235  * the savu.
2236  *
2237  * You are not expected to understand this.
2238  */
```

“If the new process paused because it was swapped out, set the stack level to the last call to savu(u_ssav). This means that the return which is executed immediately after the call to aretu actually returns from the last routine which did the savu.”

“You are not expected to understand this.”

Source: Dennis Ritchie, Unix V6 slp.c (context-switching code) as per The Unix Heritage Society (tuhs.org); gif by Eddie Koehler.

Included by Ali R. Butt in CS3204 from Virginia Tech
Putting it together: Process

(Unix) Process

A(int tmp) {
    if (tmp<2)
        B();
    printf(tmp);
}
B() {
    C();
}
C() {
    A(2);
}
A(1);
...

Sequential stream of instructions

<table>
<thead>
<tr>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack</td>
</tr>
</tbody>
</table>

I/O State
(e.g., file, socket contexts)

CPU state
(PC, SP, registers..)

Resources

Stored in OS
Putting it together: Processes

- **Switch overhead:** high
  - Kernel entry: low (ish)
  - CPU state: low
  - Memory/IO state: high
- **Process creation:** high
- **Protection**
  - CPU: yes
  - Memory/IO: yes
- **Sharing overhead:** high
  (involves at least a context switch)
Putting it together: Threads

Switch overhead: medium
- Kernel entry: low(ish)
- CPU state: low

Thread creation: medium

Protection
- CPU: yes
- Memory/IO: No

- Sharing overhead: low(ish)
  (thread switch overhead low)
Kernel versus User-Mode threads

- We have been talking about Kernel threads
  - Native threads supported directly by the kernel
  - Every thread can run or block independently
  - One process may have several threads waiting on different things
- Downside of kernel threads: a bit expensive
  - Need to make a crossing into kernel mode to schedule
- Lighter weight option: User Threads
  - User program provides scheduler and thread package
  - May have several user threads per kernel thread
  - User threads may be scheduled non-preemptively relative to each other (only switch on yield())
  - Cheap
- Downside of user threads:
  - When one thread blocks on I/O, all threads block
  - Kernel cannot adjust scheduling among all threads
- Option: Scheduler Activations
  » Have kernel inform user level when thread blocks...
Some Threading Models

- **Simple One-to-One Threading Model**

- **Many-to-One**

- **Many-to-Many**
Threads in a Process

- Threads are useful at user-level
  - Parallelism, hide I/O latency, interactivity
- Option A (early Java): user-level library, within a single-threaded process
  - Library does thread context switch
  - Kernel time slices between processes, e.g., on system call I/O
- Option B (SunOS, Linux/Unix variants): green Threads
  - User-level library does thread multiplexing
- Option C (Windows): scheduler activations
  - Kernel allocates processors to user-level library
  - Thread library implements context switch
  - System call I/O that blocks triggers upcall
- Option D (Linux, MacOS, Windows): use kernel threads
  - System calls for thread fork, join, exit (and lock, unlock,...)
  - Kernel does context switching
  - Simple, but a lot of transitions between user and kernel mode
Putting it together: Multi-Cores

- Switch overhead: low (only CPU state)
- Thread creation: low
- Protection
  - CPU: yes
  - Memory/IO: No
- Sharing overhead: low (thread switch overhead low, may not need to switch at all!)
Putting it together: Hyper-Threading

- Switch overhead between hardware-threads: very-low (done in hardware)
- Contention for ALUs/FPUs may hurt performance

Process 1

- threads
- CPU state
- Mem.
- IO state

Process N

- threads
- CPU state
- Mem.
- IO state

8 threads at a time

CPU sched.

CPU

core 1

core 2

core 3

core 4

hardware-threads (hyperthreading)

OS

9/16/15
Kubiatowicz CS162 ©UCB Fall 2015
Multiprocessing vs Multiprogramming

- Remember Definitions:
  - Multiprocessing \(\equiv\) Multiple CPUs
  - Multiprogramming \(\equiv\) Multiple Jobs or Processes
  - Multithreading \(\equiv\) Multiple threads per Process

- What does it mean to run two threads “concurrently”?
  - Scheduler is free to run threads in any order and interleaving: FIFO, Random, ...
  - Dispatcher can choose to run each thread to completion or time-slice in big chunks or small chunks
Supporting 1T and MT Processes

![Diagram showing single-threaded and multithreaded processes](image-url)
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>One</td>
<td>MS/DOS, early Macintosh</td>
<td>Traditional UNIX</td>
</tr>
<tr>
<td></td>
<td>Many</td>
<td>Embedded systems (Geoworks, VxWorks, JavaOS, etc) JavaOS, Pilot(PC)</td>
<td>Mach, OS/2, Linux Windows 9x???, Win NT to XP, Solaris, HP-UX, OS X</td>
</tr>
</tbody>
</table>

- Real operating systems have either
  - One or many address spaces
  - One or many threads per address space
- Did Windows 95/98/ME have real memory protection?
  - No: Users could overwrite process tables/System DLLs
You are here... why?

- Processes
  - Thread(s) + address space
- Address Space
- Protection
- Dual Mode
- Interrupt handlers
  - Interrupts, exceptions, syscall
- File System
  - Integrates processes, users, cwd, protection
- Key Layers: OS Lib, Syscall, Subsystem, Driver
  - User handler on OS descriptors
- Process control
  - fork, wait, signal, exec
- Communication through sockets
  - Integrates processes, protection, file ops, concurrency
- Client-Server Protocol
- Concurrent Execution: Threads
- Scheduling
Historically, OS was the most complex software
- Concurrency, synchronization, processes, devices, communication, ...
- Core systems concepts developed there

Today, many “applications” are complex software systems too
- These concepts appear there
- But they are realized out of the capabilities provided by the operating system

Seek to understand how these capabilities are implemented upon the basic hardware.

See concepts multiple times from multiple perspectives
- Lecture provides conceptual framework, integration, examples, ...
- Book provides a reference with some additional detail
- Lots of other resources that you need to learn to use
  » man pages, google, reference manuals, includes (.h)

Homework and Group Homework provides detail down to the actual code AND direct hands-on experience
Starting today: Pintos Homeworks

- Groups almost all formed
- Work as one!
- more work than homework!

- P1: threads & scheduler
- P2: user process
- P3: file system
MT Kernel 1T Process ala Pintos/x86

- Each user process/thread associated with a kernel thread, described by a 4kb Page object containing TCB and kernel stack for the kernel thread
In User thread, w/ k-thread waiting

- x86 proc holds interrupt SP high system level
- During user thread exec, associate kernel thread is “standing by”
In Kernel thread

- Kernel threads execute with small stack in thread struct
- Scheduler selects among ready kernel and user threads
Thread Switch (switch.S)

- switch_threads: save regs on current small stack, change SP, return from destination threads call to switch_threads
Switch to Kernel Thread for Process

Kernel

User

Proc Regs

IP

SP

K SP

PL: 0

code

data

heap

User stack

code

data

heap

User stack

magic #

list

priority

stack

status

tid

Proc Regs

SP

K SP

PL: 0
Kernel \rightarrow \text{User}

- \text{iret restores user stack and PL}

![Diagram showing the interaction between kernel and user processes]

- Memory regions: code, data, heap, user stack

- Processor registers: IP, SP, Proc Regs, PL: 3

- Structure: magic #, priority, stack, status, tid

- Context:...

Mechanism to resume k-thread goes through interrupt vector
User->Kernel via interrupt vector

- Interrupt transfers control through the IV (IDT in x86)
- iret restores user stack and PL
Pintos Interrupt Processing

**Hardware interrupt vector**

```assembly
0x20

stubs

push 0x20 (int #)
jmp intr_entry

push 0x21 (int #)
jmp intr_entry

intr_entry:
save regs as frame
set up kernel env.
call intr_handler

intr_exit:
restore regs
iret

stubs.S
```

**Wrapper for generic handler**

```assembly
0

push 0x20 (int #)
jmp intr_entry

push 0x21 (int #)
jmp intr_entry

***

***
```
Pintos Interrupt Processing

**Hardware interrupt vector**

```
0x20 0
0x21 255
```

**stubs**

```
push 0x20 (int #)
jmp intr_entry
```

**intr_entry:**
- save regs as frame
- set up kernel env.
- call intr_handler

```
intr_exit:
- restore regs
- iret
```

**Wrapper for generic handler**

```
Intr_handler(*frame)
- classify
- dispatch
- ack IRQ
- maybe thread yield
```

**intr_handlers**

```
timer_intr(*frame)
tick++
thread_tick()
```

```
stubs.S
```

**interrupt.c**

```
timer.c
```
Timer may trigger thread switch

- **thread_tick**
  - Updates thread counters
  - If quanta exhausted, sets yield flag
- **thread_yield**
  - On path to rtn from interrupt
  - Sets current thread back to READY
  - Pushes it back on ready_list
  - Calls schedule to select next thread to run upon iret
- **Schedule**
  - Selects next thread to run
  - Calls switch_threads to change regs to point to stack for thread to resume
  - Sets its status to RUNNING
  - If user thread, activates the process
  - Returns back to intr_handler
Pintos Return from Processing

hardware interrupt vector

stubs

push 0x20 (int #)
jmp intr_entry

push 0x21 (int #)
jmp intr_entry

intr_entry:
save regs as frame
set up kernel env.
call intr_handler

intr_exit:
restore regs
iret

Wrapper for generic handler

interrupt.c

Intr_handler(*frame)
- classify
- dispatch
- ack IRQ
- maybe thread yield

intr Handlers

12

Pintos

intr Handlers

0x20

timer.c

timer_intr(*frame)
tick++
thread_tick()

thread_yield()
- schedule

Schedule()
- switch
Correctness for systems with concurrent threads

- If dispatcher can schedule threads in any way, programs must work under all circumstances
  - Can you test for this?
  - How can you know if your program works?
- Independent Threads:
  - No state shared with other threads
  - Deterministic $\Rightarrow$ Input state determines results
  - Reproducible $\Rightarrow$ Can recreate Starting Conditions, I/O
  - Scheduling order doesn't matter (if `switch()` works!!!)
- Cooperating Threads:
  - Shared State between multiple threads
  - Non-deterministic
  - Non-reproducible
- Non-deterministic and Non-reproducible means that bugs can be intermittent
  - Sometimes called “Heisenbugs”
Interactions Complicate Debugging

• Is any program truly independent?
  - Every process shares the file system, OS resources, network, etc
  - Extreme example: buggy device driver causes thread A to crash “independent thread” B

• You probably don’t realize how much you depend on reproducibility:
  - Example: Evil C compiler
    » Modifies files behind your back by inserting errors into C program unless you insert debugging code
  - Example: Debugging statements can overrun stack

• Non-deterministic errors are really difficult to find
  - Example: Memory layout of kernel+user programs
    » depends on scheduling, which depends on timer/other things
    » Original UNIX had a bunch of non-deterministic errors
Summary (1 of 2)

- Processes have two parts
  - Threads (Concurrency)
  - Address Spaces (Protection)

- Concurrency accomplished by multiplexing CPU Time:
  - Unloading current thread (PC, registers)
  - Loading new thread (PC, registers)
  - Such context switching may be voluntary (`yield()`, I/O operations) or involuntary (timer, other interrupts)

- Protection accomplished restricting access:
  - Memory mapping isolates processes from each other
  - Dual-mode for isolating I/O, other resources

- Various Textbooks talk about processes
  - When this concerns concurrency, really talking about thread portion of a process
  - When this concerns protection, talking about address space portion of a process
Summary (2 or 2)

- Concurrent threads are a very useful abstraction
  - Allow transparent overlapping of computation and I/O
  - Allow use of parallel processing when available

- Concurrent threads introduce problems when accessing shared data
  - Programs must be insensitive to arbitrary interleavings
  - Without careful design, shared variables can become completely inconsistent

- Important concept: Atomic Operations
  - An operation that runs to completion or not at all
  - These are the primitives on which to construct various synchronization primitives

- Showed how to protect a critical section with only atomic load and store ⇒ pretty complex!