SE350: Operating Systems

Lecture 5: Multithreaded Kernels
Outline

• Use cases for multithreaded programs
• Kernel vs. user-mode threads
• Concurrency’s problems
Recall: Why Processes & Threads?

Goals:

• Multiprogramming: Run multiple applications concurrently
• Protection: Don’t want bad applications to crash system!

Solution:

• Process: unit of execution and allocation
• Virtual Machine abstraction: give process illusion it owns machine (i.e., CPU, Memory, and IO device multiplexing)

Challenge:

• Process creation & switching expensive
• Need concurrency within same app (e.g., web server)

Solution:

• Thread: Decouple allocation and execution
• Run multiple threads within same process
Multithreaded Processes

- PCBs could point to multiple TCBs

- Switching threads within one block is simple thread switch
- Switching threads across blocks requires changes to memory and I/O address tables
Examples Multithreaded Programs

• Embedded systems
  • Elevators, planes, medical systems, smart watches
  • Single program, concurrent operations

• Most modern OS kernels
  • Internally concurrent to deal with concurrent requests by multiple users/applications
  • 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 (cont.)

• 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

• Some multiprocessors are actually uniprogrammed
  • Multiple threads in one address space but one program at a time
A Typical Use Case

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

Web Server
- fork process for each client connection
- create threads to get request and issue response
- create threads to read data, access DB, etc.
- join and respond
Kernel Use Cases

- Thread for each user process
- Thread for sequence of steps in processing I/O
- Threads for device drivers
- …
Device Drivers

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

- Device drivers are 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()`, etc.
    - This is kernel's interface to 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
Life Cycle of An I/O Request

User Program

Kernel I/O Subsystem

Device Driver Top Half

Device Driver Bottom Half

Device Hardware
Multithreaded Kernel

- User programs use syscalls to create, join, yield, exit threads
- Kernel handles scheduling and context switching
- Simple, but a lot of transitions between user and kernel mode
Kernel vs. User-Mode Threads

- We have been talking about kernel supported threads
  - Each user-level thread maps to one kernel thread
  - Every thread can run or block independently
  - One process may have several threads waiting on different events
  - Examples: Windows, Linux

- Downside of kernel supported threads: a bit expensive
  - Need to make crossing into kernel mode to schedule
  - Solution: user supported threads
Basic Cost of System Calls

- Min syscall has ~ 25x cost of function call
- Scheduling could be many times more
- Streamline system processing as much as possible
- Other optimizations seek to process as much of syscall in user space as possible (e.g., Linux vDSO)
**User-Mode Threads**

- Lighter weight option
  - Many user-level threads are mapped to single kernel thread
  - User program provides scheduler and thread package
  - Examples: Solaris Green Threads, GNU Portable Threads

- Downside of user-mode threads
  - Multiple threads may not run in parallel on multicore
  - When one thread blocks on I/O, all threads block
  - Option: *Scheduler Activations*
    - Have kernel inform user level when thread blocks …
### Classification

- Most operating systems have either
  - One or many address spaces
  - One or many threads per address space

<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>One</td>
<td>MS/DOS, early Macintosh</td>
<td>Traditional UNIX</td>
</tr>
<tr>
<td>Many</td>
<td>Embedded systems (Geoworks, VxWorks, JavaOS, Pilot(PC), etc.)</td>
<td>Mach, OS/2, Linux Windows 10, Win NT to XP, Solaris, HP-UX, OS X</td>
<td></td>
</tr>
</tbody>
</table>
Putting it Together: Process

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

Sequential stream of instructions

(Unix) Process

- Memory
- Stack
- I/O State (e.g., file, socket contexts)
- CPU state (PC, SP, registers..)

Resources

Stored in OS
Putting it Together: Processes

- **Switch overhead:** high
  - CPU state: low
  - Memory/IO state: high
- **Process creation:** high
- **Protection**
  - CPU: yes
  - Memory/IO: yes
- **Sharing overhead:** high
  (involves at least one context switch)
Putting it Together: Threads

- Switch overhead: medium
  - CPU state: low
- Thread creation: medium
- Protection
  - CPU: yes
  - Memory/IO: no
- Sharing overhead: low(ish)
  (thread switch overhead low)
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!)

[Diagram showing multi-core system with process 1 and process N, CPU scheduler, and 4 threads at a time]
Hyperthreading

- Superscalar processors can execute multiple instructions that are independent
- Multiprocessors can execute multiple independent threads
- Fine-grained multithreading executes two independent threads by switches between them
- Hyperthreading duplicates register state to make second (hardware) “thread” (virtual core)
  - From OS’s point of view, virtual cores are separate CPUs
  - OS can schedule as many threads at a time as there are virtual cores (but, sub-linear speedup!)
Putting it Together: Hyperthreading

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

- **Illusion:** Infinite number of processors
  - Each thread runs on dedicated virtual processor
- **Reality:** few processors, multiple threads running at variable speed
  - To map arbitrary set of threads to fixed set of cores, kernel implements scheduler

![Diagram showing thread abstraction](image)
## Programmer vs. Processor View

<table>
<thead>
<tr>
<th>Programmer’s View</th>
<th>Possible Execution #1</th>
<th>Possible Execution #2</th>
<th>Possible Execution #3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x = x + 1;</td>
<td>x = x + 1;</td>
<td>x = x + 1;</td>
<td>x = x + 1;</td>
</tr>
<tr>
<td>y = y + x;</td>
<td>y = y + x;</td>
<td>y = y + x;</td>
<td>y = y + x;</td>
</tr>
<tr>
<td>z = x + 5y;</td>
<td>z = x + 5y;</td>
<td>z = x + 5y;</td>
<td>z = x + 5y;</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Thread is suspended.</td>
<td></td>
<td>Other thread(s) run.</td>
<td>Other thread(s) run.</td>
</tr>
<tr>
<td>Thread is resumed.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>y = y + x;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>z = x + 5y;</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Possible Interleavings

One Execution

Thread 1
Thread 2
Thread 3

Another Execution

Thread 1
Thread 2
Thread 3

Another Execution

Thread 1
Thread 2
Thread 3
Correctness with Concurrent Threads

- If threads can be scheduled in any way, programs must work under all conditions
  - Can you test for this?
  - How can you know if your program works?

- Independent Threads
  - No state shared with other threads
  - Deterministic ⇒ Input state determines results
  - Reproducible ⇒ 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 file system, OS resources, network, etc.
  • E.g., buggy device driver makes thread 1 crash “independent thread” 2

• Non-deterministic errors are extremely difficult to find
  • E.g., memory layout of kernel + user programs
    • Depends on scheduling, which depends on timer/other things
    • Original UNIX had bunch of non-deterministic errors
  • E.g., something which does interesting I/O
    • User typing of letters used to help generate secure keys
Why Allow Cooperating Threads?

• Advantage 1: **Sharing resources**
  • One computer, many users
  • One bank balance, many ATMs
    • What if ATMs were only updated at night?
    • Embedded systems (robot control: coordinate arm & hand)

• Advantage 2: **Speedup**
  • Overlap I/O and computation
    • Many different file systems do read-ahead
    • Multiprocessors – chop up program into parallel pieces

• Advantage 3: **Modularity**
  • More important than you might think
  • Chop large problem up into simpler pieces
High-level Example: Web Server

- Server must handle many requests
- Non-cooperating version:
  ```
  serverLoop() {
    connection = AcceptCon();
    fork(ServiceWebPage, connection);
  }
  ```
- What are some disadvantages of this technique?
Threaded Web Server

• Instead, use single process

• Multithreaded (cooperating) version:
  
  ```c
  serverLoop() {
    connection = AcceptCon();
    thread_create(ServiceWebPage, connection);
  }
  ```

• Looks almost the same, but has many advantages
  • Can share file caches kept in memory
  • Threads are cheaper to create than processes (lower per-request overhead)

• What about Denial of Service (DoS) attacks?
Thread Pools

- Problem with previous version: unbounded number of threads
  - When web-site becomes too popular, throughput sinks
- Instead, allocate bounded “pool” of worker threads, representing maximum level of multiprogramming

```java
master() {
    allocThreads(worker, queue);
    while(TRUE) {
        con=AcceptCon();
        Enqueue(queue, con);
        wakeUp(queue);
    }
}

worker(queue) {
    while(TRUE) {
        con=Dequeue(queue);
        if (con==null)
            sleepOn(queue);
        else
            ServiceWebPage(con);
    }
}
```
ATM Bank Server

ATM server requirements:
- Service a set of requests
- Do so without corrupting database
- Don’t hand out too much money
ATM bank server example

- Suppose we wanted to implement server process to handle requests from ATM network

```c
BankServer() {
    while (TRUE) {
        ReceiveRequest(&op, &acctId, &amount);
        ProcessRequest(op, acctId, amount);
    }
}
ProcessRequest(op, acctId, amount) {
    if (op == deposit) Deposit(acctId, amount);
    else if ...
}
Deposit(acctId, amount) {
    acct = GetAccount(acctId);
    acct->balance += amount;
    StoreAccount(acct);
}
```

- How could we speed this up?
  - More than one request being processed at once
  - Event driven (overlap computation and I/O)
  - Multiple threads (multi-proc, or overlap comp and I/O)
Can Threads Make This Easier?

- Threads yield overlapped I/O and computation without having to “deconstruct” code into non-blocking fragments
  - One thread per request
- Requests proceeds to completion, blocking as required

```c
Deposit(acctId, amount) {
    acct = GetAccount(acctId);    /* May use disk I/O */
    acct->balance += amount;
    StoreAccount(acct);           /* Involves disk I/O */
}
```

- Unfortunately, shared state can get corrupted:
  
<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>load r1, acct-&gt;balance</td>
<td>load r1, acct-&gt;balance</td>
</tr>
<tr>
<td>add r1, amount1</td>
<td>add r1, amount2</td>
</tr>
<tr>
<td>store r1, acct-&gt;balance</td>
<td>store r1, acct-&gt;balance</td>
</tr>
</tbody>
</table>
Problem Is At The Lowest Level

• When threads work on separate data, order of scheduling does not change results

  Thread A
  \[ x = 1; \]
  \[ y = 2; \]

  Thread B

• Scheduling order matters when threads work on shared data

  Thread A
  \[ x = 1; \]
  \[ x = y + 1; \]
  \[ y = y \times 2; \]

  Thread B

• What are possible values of \( x \)? (initially, \( y = 12 \))

  Thread A
  \[ x = 1; \]
  \[ x = y + 1; \]

  Thread B
  \[ y = 2; \]
  \[ y = y \times 2; \]

  \[ x = 13 \]
Problem Is At The Lowest Level

- When threads work on separate data, order of scheduling does not change results
  
  **Thread A**
  
  ```
  x = 1;
  y = 2;
  ```

- Scheduling order matters when threads work on shared data
  
  **Thread A**
  
  ```
  x = 1;
  x = y + 1;
  ```

  **Thread B**
  
  ```
  y = 2;
  y = y * 2;
  ```

- What are possible values of $x$? (initially, $y = 12$)
  
  **Thread A**
  
  ```
  x = 1;
  x = y + 1;
  ```

  **Thread B**
  
  ```
  y = 2;
  y = y * 2;
  ```

  $x = 5$
Problem Is At The Lowest Level

• When threads work on separate data, order of scheduling does not change results
  
  **Thread A**
  
  \[
  \begin{align*}
  x &= 1; \\
  y &= 2;
  \end{align*}
  
  **Thread B**
  
  \[
  \begin{align*}
  x &= 1; \\
  y &= 2;
  \end{align*}
  
• Scheduling order matters when threads work on shared data
  
  **Thread A**
  
  \[
  \begin{align*}
  x &= 1; \\
  y &= 2; \\
  x &= y + 1;
  \end{align*}
  
  **Thread B**
  
  \[
  \begin{align*}
  x &= 1; \\
  y &= y * 2;
  \end{align*}
  
• What are possible values of \(x\)? (initially, \(y = 12\))
  
  **Thread A**
  
  \[
  \begin{align*}
  x &= 1; \\
  y &= 2; \\
  x &= y + 1;
  \end{align*}
  
  **Thread B**
  
  \[
  \begin{align*}
  x &= 1; \\
  y &= y * 2;
  \end{align*}
  
\[x = 3\]
Summary

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

- Various textbooks talk about processes
  - When this concerns concurrency, talking about thread portion
  - When this concerns protection, talking about address space portion

- 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
Questions?
Acknowledgment

- Slides by courtesy of Anderson, Culler, Stoica, Silberschatz, Joseph, and Canny