CS162
Operating Systems and
Systems Programming
Lecture 3

Abstractions 1: Threads
A quick, programmer’s viewpoint

September 2nd, 2021
Prof. Ion Stoica
http://cs162.eecs.Berkeley.edu
Recall: Four fundamental OS concepts

- **Thread**
  - Single unique execution context
  - Program Counter, Registers, Execution Flags, Stack

- **Address Space** with translation
  - Programs execute in an address space that is distinct from the memory space of the physical machine

- **Process**
  - An instance of an executing program is a process consisting of an address space and one or more threads of control

- **Dual Mode** operation/protection
  - Only the “system” has the ability to access certain resources
  - The OS and the hardware are protected from user programs and user programs are isolated from one another by controlling the translation from program virtual addresses to machine physical addresses
Simple Protection: Base and Bound (B&B)

- **Code**: 0000...
- **Static Data**: 0100...
- **Heap**: 0010...
- **Stack**: Bound
- **Program address**
  - **Base**: 1000...
  - **Bound**: 1100...
  - **Bound**: 1100...

**Diagram**:
- Code: 0000...
- Static Data: 0100...
- Heap: 0010...
- Stack: Bound
- Program address: 1010...
- Base: 1000...
- Bound: 1100...
- FFFF...

**Notes**:
- The diagram illustrates how program addresses are mapped to different sections of memory based on the base and bound addresses.
### Simple Protection: Base and Bound (B&B)

- Requires relocating loader
- Still protects OS and isolates program
- No addition on address path

Addresses translated when program is loaded.
Another idea: Address Space Translation

- Program operates in an address space that is distinct from the physical memory space of the machine
A simple address translation with Base and Bound

- Can the program touch OS?
- Can it touch other programs?
Tying it together: Simple B&B: OS loads process
Simple B&B: OS gets ready to execute process

- Privileged Inst: set special registers
- RTU
• How does kernel switch between processes?
• First question: How to return to system?
3 types of Mode Transfer

• Syscall
  – Process requests a system service, e.g., exit
  – Like a function call, but “outside” the process
  – Does not have the address of the system function to call
  – Like a Remote Procedure Call (RPC) – for later
  – Marshall the syscall id and args in registers and exec syscall

• Interrupt
  – External asynchronous event triggers context switch
  – e.g., Timer, I/O device
  – Independent of user process

• Trap or Exception
  – Internal synchronous event in process triggers context switch
  – e.g., Protection violation (segmentation fault), Divide by zero, …

• All 3 are an UNPROGRAMMED CONTROL TRANSFER
  – Where does it go?
How do we get the system target address of the “unprogrammed control transfer?”
Interrupt Vector

interrupt number (i)

Address and properties of each interrupt handler

intrpHandler_i () {
    ...
}

• Where else do you see this dispatch pattern?
Simple B&B: User => Kernel

- How to return to system?
Simple B&B: Interrupt

- How to save registers and set up system stack?
Simple B&B: Switch User Process

- How to save registers and set up system stack?
Simple B&B: “resume”

- How to save registers and set up system stack?
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
Threads Allow Handling MTAO

- Threads are a unit of *concurrency* provided by the OS
- Each thread can represent one thing or one task
Multiprocessing vs. Multiprogramming

• Some Definitions:
  – Multiprocessing: Multiple CPUs (cores)
  – Multiprogramming: Multiple jobs/processes
  – Multithreading: Multiple threads/processes

• What does it mean to run two threads concurrently?
  – Scheduler is free to run threads in any order and interleaving
  – Thread may run to completion or time-slice in big chunks or small chunks
Concurrency is not Parallelism

- Concurrency is about handling multiple things at once (MTAO)
- Parallelism is about doing multiple things simultaneously

Example: Two threads on a single-core system...
- … execute concurrently …
- … but not in parallel

- Each thread handles or manages a separate thing or task…
- But those tasks are not necessarily executing simultaneously!
Silly Example for Threads

• Imagine the following program:
  ```c
  main() {
    ComputePI("pi.txt");
    PrintClassList("classlist.txt");
  }
  ```

• What is the behavior here?
  • Program would never print out class list
  • Why? `ComputePI` would never finish
Adding Threads

• Version of program with threads (loose syntax):
  ```c
  main() {
    create_thread(ComputePI, "pi.txt");
    create_thread(PrintClassList, "classList.txt");
  }
  ```

• `create_thread`: Spawns a new thread running the given procedure
  – *Should* behave as if another CPU is running the given procedure

• Now, you would actually see the class list
Administrivia: Getting started

- Should be working on Homework 0 already! ⇒ Due Friday (9/3)
  - cs162-xx account, Github account, registration survey
  - Get familiar with all the cs162 tools, submit to autograder via git
- Should be working on Project 0 already! ⇒ Due Next Thursday (9/9)
  - To be done on your own – like a homework!
- Slip days: I’d bank these and not spend them right away!
  - 7 slip days for homeworks
  - 7 slip days for projects (slip days don't apply to design reviews + Proj0)
  - No credit when late and run out of slip days
- Friday (9/3) is drop day!
  - Very hard to drop afterwards…
  - Please drop sooner if you are going to anyway ⇒ Let someone else in!
- Group creation form sue on Sunday (9/5)!
Threads Mask I/O Latency

• A thread is in one of the following three states:
  – RUNNING – running
  – READY – eligible to run, but not currently running
  – BLOCKED – ineligible to run

• If a thread is waiting for an I/O to finish, the OS marks it as BLOCKED
• Once the I/O finally finishes, the OS marks it as READY
Threads Mask I/O Latency

- If no thread performs I/O:

- If thread 1 performs a blocking I/O operation:
A Better Example for Threads

• Version of program with threads (loose syntax):

```c
main() {
    create_thread(ReadLargeFile, "pi.txt");
    create_thread(RenderUserInterface);
}
```

• What is the behavior here?
  – Still respond to user input
  – While reading file in the background
Multithreaded Programs

- You know how to compile a C program and run the executable
  - This creates a process that is executing that program

- Initially, this new process has one thread in its own address space
  - With code, global variables, etc. as specified in the executable

- Q: How can we make a multithreaded process?
- A: Once the process starts, it issues system calls to create new threads
  - These new threads are part of the process: they share its address space
System Calls ("Syscalls")

“But, I've never seen a syscall!”
- OS library issues system call
- Language runtime uses OS library…
OS Library Issues Syscalls

OS

Proc 1
Proc 2
Proc n

OS library

AppIn

login

Window Manager

libc

OS
OS Library API for Threads: *pthreads*

```c
int pthread_create(pthread_t *thread, const pthread_attr_t *attr, 
    void **(*start_routine)(void*), void *arg);
    
    – thread is created executing start_routine with arg as its sole argument.
    – return is implicit call to pthread_exit

void pthread_exit(void *value_ptr);
    
    – terminates the thread and makes value_ptr available to any successful join

int pthread_join(pthread_t thread, void **value_ptr);
    
    – suspends execution of the calling thread until the target thread terminates.
    – On return with a non-NULL value_ptr the value passed to pthread_exit() by the 
      terminating thread is made available in the location referenced by value_ptr.
```

prompt% man pthread
https://pubs.opengroup.org/onlinepubs/7908799/xsh/pthread.h.html
Peeking Ahead: System Call Example

• What happens when `pthread_create(…)` is called in a process?

Library:
```
int pthread_create(…) {
    Do some work like a normal fn…

    asm code … syscall # into %eax
    put args into registers %ebx, …
    special trap instruction
```

Kernel:
```
get args from regs
dispatch to system func
Do the work to spawn the new thread
Store return value in %eax

get return values from regs
Do some more work like a normal fn…
};
```
New Idea: Fork-Join Pattern

- Main thread creates (forks) collection of sub-threads passing them args to work on…
- … and then joins with them, collecting results.
pThreads Example

- How many threads are in this program?
- Does the main thread join with the threads in the same order that they were created?
- Do the threads exit in the same order they were created?
- If we run the program again, would the result change?
Thread State

- State shared by all threads in process/address space
  - Content of memory (global variables, heap)
  - I/O state (file descriptors, network connections, etc)

- State “private” to each thread
  - Kept in TCB ≡ Thread Control Block
  - CPU registers (including, program counter)
  - Execution stack – what is this?

- Execution Stack
  - Parameters, temporary variables
  - Return PCs are kept while called procedures are executing
Execution Stack Example

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

```c
A(int tmp) {
    if (tmp<2)
        B();
    printf(tmp);
}
B() {
    C();
}
C() {
    A(2);
}
A(1);
```
Execution Stack Example

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

```
A(int tmp) {
    if (tmp<2)
        B();
    printf(tmp);
}
B() {
    C();
}
C() {
    A(2);
}
A(1);
```
Execution Stack Example

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

```c
A(int tmp) {
    if (tmp<2)
    B();
    printf(tmp);
}
B() {
    C();
}
C() {
    A(2);
}
A(1);
```
Execution Stack Example

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

```c
A(int tmp) {
    if (tmp < 2)
        B();
    printf(tmp);
}

B() {
    C();
}

C() {
    A(2);
}

A(1);
```
Execution Stack Example

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

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

B() {
    C();
}

C() {
    A(2);
}

A(1);
Execution Stack Example

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

```
A(int tmp) {
    if (tmp<2)
        B();
    printf(tmp);
}
B() {
    C();
}
C() {
    A(2);
}
A(1);
```
Execution Stack Example

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

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

A: tmp=1
   ret=exit
B: ret=A+2
C: ret=B+1

Stack Pointer
Execution Stack Example

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

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

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

**Output:** 

```
>2
```
Execution Stack Example

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

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

B() {
  C();
}

C() {
  A(2);
}

A(1);

Output: >2

Stack Growth

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

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

Output: >2

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

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

B() {
    C();
}

C() {
    A(2);
}

Output: >2

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

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

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

Output: \texttt{>2 1}
Execution Stack Example

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

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

B() {
    C();
}

C() {
    A(2);
}
A(1);
```

Output: \(2 1\)

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

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

Output: >2 1

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

A(int tmp) {
    if (tmp<2)
        B();
    printf(tmp);
}
B() {
    C();
}
C() {
    A(2);
}
A(1);
Memory Layout with Two Threads

- Two sets of CPU registers
- Two sets of Stacks
- Issues:
  - How do we position stacks relative to each other?
  - What maximum size should we choose for the stacks?
  - What happens if threads violate this?
  - How might you catch violations?
INTERLEAVING AND NONDETERMINISM
(The beginning of a long discussion!)
Thread Abstraction

- Illusion: Infinite number of processors
- Reality: Threads execute with variable “speed”
  - Programs must be designed to work with any schedule
## 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>$\ldots\ldots\ldots\ldots$</td>
<td>$y = y + x$;</td>
</tr>
<tr>
<td>$z = x + 5y$;</td>
<td>$z = x + 5y$;</td>
<td>thread is suspended</td>
<td>$\ldots\ldots\ldots\ldots$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>other thread(s) run</td>
<td>thread is suspended</td>
</tr>
<tr>
<td></td>
<td></td>
<td>thread is resumed</td>
<td>other thread(s) run</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\ldots\ldots\ldots\ldots$</td>
<td>thread is resumed</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$y = y + x$</td>
<td>$\ldots\ldots\ldots\ldots$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$z = x + 5y$</td>
<td>$z = x + 5y$</td>
</tr>
</tbody>
</table>
Possible Executions

Thread 1  Thread 1
Thread 2  Thread 2
Thread 3  Thread 3

a) One execution  b) Another execution

c) Another execution
Correctness with Concurrent Threads

• Non-determinism:
  – Scheduler can run threads in **any order**
  – Scheduler can switch threads **at any time**
  – This can make testing very difficult

• *Independent Threads*
  – No state shared with other threads
  – Deterministic, reproducible conditions

• *Cooperating Threads*
  – Shared state between multiple threads

• **Goal: Correctness by Design**
Race Conditions

- Initially $x = 0$ and $y = 0$
  
  **Thread A** | **Thread B**
  
  $x = 1$; | $y = 2$;

- What are the possible values of $x$ below after all threads finish?
- Must be 1. Thread B does not interfere
Race Conditions

• Initially \( x = 0 \) and \( y = 0 \)

\[
\text{Thread A} & \\
\text{Thread B}
\]

\[
x = y + 1; & \quad y = 2; \\
y = y \ast 2;
\]

• What are the possible values of \( x \) below?
• 1 or 3 or 5 (non-deterministically)
• Race Condition: Thread A races against Thread B!
Example: Shared Data Structure

Thread A
Insert(3)

Thread B
Insert(4)
Get(6)

Tree-Based Set Data Structure
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
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

• For now, don’t worry about how to implement locks!
  – We’ll cover that in substantial depth later on in the class
Thread A
Insert(3)
- Lock.acquire()
- Insert 3 into the data structure
- Lock.release()

Thread B
Insert(4)
- Lock.acquire()
- Insert 4 into the data structure
- Lock.release()

Get(6)
- Lock.acquire()
- Check for membership
- Lock.release()

Tree-Based Set Data Structure
OS Library Locks: \textit{pthreads}

\begin{verbatim}
int pthread_mutex_init(pthread_mutex_t *mutex,
                const pthread_mutexattr_t *attr)

int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
\end{verbatim}

You'll get a chance to use these in Homework 1
Our Example

```c
int common = 162;
pthread_mutex_t common_lock = PTHREAD_MUTEX_INITIALIZER;

void *threadfun(void *threadid)
{
    long tid = (long)threadid;
pthread_mutex_lock(&common_lock);
    int my_common = common++;
pthread_mutex_unlock(&common_lock);

    printf("Thread %lx stack: %lx common: %lx (%d)\n", tid,
           (unsigned long) &tid,
           (unsigned long) &common, my_common);
    pthread_exit(NULL);
}
```
Conclusion

• Threads are the OS unit of concurrency
  – Abstraction of a virtual CPU core
  – Can use `pthread_create`, etc., to manage threads within a process
    – They share data → need synchronization to avoid data races

• We saw the role of the OS library
  – Provide API to programs
  – Interface with the OS to request services