CS162
Operating Systems and Systems Programming
Lecture 3

Abstractions 1: Threads and Processes
A quick, programmer’s viewpoint

January 23rd, 2024
Prof. John Kubiatowicz
http://cs162.eecs.Berkeley.edu
Goals for Today: The Thread Abstraction

• **What** threads are
  – And what they are not
• **Why** threads are useful (motivation)
• **How** to write a program using threads
• **Alternatives** to using threads
Recall: Four Fundamental OS Concepts

• **Thread: Execution Context**
  – Fully describes program state
  – Program Counter, Registers, Execution Flags, Stack

• **Address space (with or w/o translation)**
  – Set of memory addresses accessible to program (for read or write)
  – May be distinct from memory space of the physical machine
    (in which case programs operate in a virtual address space)

• **Process: an instance of a running program**
  – Protected Address Space + One or more Threads

• **Dual mode operation / Protection**
  – Only the “system” has the ability to access certain resources
  – Combined with translation, isolates programs from each other and
    the OS from programs
Recall: Illusion of Multiple Processors

- Threads are **virtual cores**
- Multiple threads: **Multiplex** hardware in time
- A Thread is **executing** on a processor when it is resident in that processor’s registers

- Each virtual core (thread) has:
  - Program counter (PC), stack pointer (SP)
  - Registers – both integer and floating point

- Where is “it” (the thread)?
  - On the real (physical) core, or
  - Saved in chunk of memory – called the **Thread Control Block (TCB)**
Recall: Process

• Definition: execution environment with Restricted Rights
  – One or more threads in a (protected) Address Space
  – Owns memory (address space), file descriptors, network connections, …

• Instance of a running program
  – When you run an executable, it runs in its own process
  – Application: one or more processes working together

• Why processes?
  – Protected from each other!
  – OS Protected from them

• In modern OS, anything that runs outside of the kernel runs in a process
Recall: Simple address translation with Base and Bound

- Addresses translated on-the-fly

- Can the program touch OS?
- Can it touch other programs?
Simple B&B: User => Kernel

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

- How to save registers and set up system stack?

IntrpVector[i]
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?
Is Simple Base and Bound Enough for General Systems?

• NO: Too simplistic for real systems

• Inflexible/Wasteful:
  – Must dedicate physical memory for potential future use
  – (Think stack and heap!)

• Fragmentation:
  – Kernel has to somehow fit whole processes into contiguous block of memory
  – After a while, memory becomes fragmented!

• Sharing:
  – Very hard to share any data between Processes or between Process and Kernel
  – Need to communicate indirectly through the kernel…
Better: Translation through Page Table (More soon!)

Translation Map 1

Translation Map 2

Physical Address Space
Recall: Dual Mode Operation

- Processes (i.e., programs you run) execute in **user mode**
  - To perform privileged actions, processes request services from the OS kernel
  - Carefully controlled transition from user to kernel mode
- Kernel executes in **kernel mode**
  - Performs privileged actions to support running processes
  - … and configures hardware to properly protect them (e.g., address translation)
- Carefully controlled transitions between user mode and kernel mode
  - System calls, interrupts, exceptions
Adding Protection: CPU Switch From Process A to Process B

- User Mode
- Kernel/System Mode
- User Mode

Process $P_0$ executing, interrupt or system call:
- Save state into PCB$_0$
- ... (omitted for brevity)
- Reload state from PCB$_1$

Process $P_1$ executing, idle:
- ... (omitted for brevity)
- Reload state from PCB$_0$
Running Many Programs

• We have the basic *mechanism* to:
  – switch between user processes and the kernel,
  – the kernel can switch among user processes,
  – Protect OS from user processes and processes from each other

• Questions:
  – How do we represent user processes in the OS?
  – How do we decide which user process to run?
  – How do we pack up the process and set it aside?
  – How do we get a stack and heap for the kernel?
  – Aren’t we wasting a lot of memory?
Multiplexing Processes: The Process Control Block

- Kernel represents each process with a process control block (PCB)
  - Status (running, ready, blocked, …)
  - Register state (when not ready)
  - Process ID (PID), User, Executable, Priority, …
  - Execution time, …
  - Memory space, translation, …
- Kernel Scheduler maintains a data structure containing the PCBs
  - Give out CPU to different processes
  - This is a Policy Decision
- Give out non-CPU resources
  - Memory/IO
  - Another policy decision
Scheduler

- Scheduling: Mechanism for deciding which processes/threads receive hardware CPU time, when, and for how long
- Lots of different scheduling policies provide ...
  - Fairness or
  - Realtime guarantees or
  - Latency optimization or ..

```c
if ( readyProcesses(PCBs) ) {
    nextPCB = selectProcess(PCBs);
    run( nextPCB );
} else {
    run_idle_process();
}
```
Also Recall: The World Is Parallel: Saphire Rapids (2023)

- Up to 60 cores, 120 threads/package (socket)
  - Up to 4 “chiplets” bonded together
- Network:
  - On-chip Mesh Interconnect
  - Fast off-chip network (UPI): directly connects 8-chips
  - 480 cores/shared memory domain!
- Each Core Has:
  - 80 KB L1 Cache
  - 2 MB L2 Cache
  - Fraction of up to 112.5 MB L3 Cache
- DRAM/chips
  - Up to 4 TiB of DDR5 memory
- Many Accellerators of different types
  - Graphics, Encryption, AI, Security
Simultaneous MultiThreading/Hyperthreading

- Hardware scheduling technique
  - Avoids software overhead of multiplexing
  - Superscalar processors can execute multiple instructions that are independent.
  - Hyperthreading duplicates register state to make a second “thread,” allowing more instructions to run.
- Can schedule each thread as if were separate CPU
  - But, sub-linear speedup!

- Original technique called “Simultaneous Multithreading”
  - SPARC, Pentium 4/Xeon (“Hyperthreading”), Power 5
Administrivia: Getting started!

• Kubiatowicz Office Hours:
  – Tuesday/Thursday 3-4pm, in 673 Soda Hall

• Homework 0: Due Tomorrow!
  – Get familiar with the cs162 tools
  – configure your VM, submit via git
  – Practice finding out information:
    » How to use GDB? How to understand output of unix tools?
    » We don’t assume that you already know everything!
    » Learn to use “man” (command line), “help” (in gdb, etc), google

• Project 0: Started Yesterday!
  – Learn about Pintos and how to modify and debug kernel
  – Important for getting started on projects!

• Should be going to sections now – Important information there
  – Any section will do until groups assigned
**Administrivia (Con’t)**

- **THIS Friday is Drop Deadline! HARD TO DROP LATER!**
  - If you know you are going to drop, do so now to leave room for others on waitlist!
  - Why do we do this? So that groups aren’t left without members!
- Group sign up via autograder form next week
  - Get finding groups of 4 people ASAP
  - Priority for same section; if cannot make this work, keep same TA
  - Remember: Your TA needs to see you in section!
- **Midterm 1: 2/15**
  - 8-10PM in person
  - We will say more about material when we get closer…
- **Midterm 1 conflicts**
  - We will handle these conflicts after have final class roster
  - Watch for queries by HeadTA to collect information
CS 162 Collaboration Policy

Explaining a concept to someone in another group
Discussing algorithms/testing strategies with other groups
Discussing debugging approaches with other groups
Searching online for generic algorithms (e.g., hash table)

Sharing code or test cases with another group or individual (including HW!)
Copying OR reading another group’s code or test cases
Copying OR reading online code or test cases from prior years
Helping someone in another group to debug their code

• We compare all project and HW submissions against prior year submissions and online solutions and will take actions (described on the course overview page) against offenders
• Don’t put a friend in a bad position by asking for help that they shouldn’t give!
More About Threads: What are they?

• 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* (overlapping execution)
  – However, they can also run in *parallel* (simultaneous execution)

• Protection is an orthogonal concept
  – A protection domain can contain one thread or many
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

I made this term up!
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:

```java
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):

```cpp
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

```
CPU1  CPU2  CPU1  CPU2  CPU1  CPU2

Time
```
More Practical Motivation: Compute/IO overlap

Back to Jeff Dean’s “Numbers Everyone Should Know”

<table>
<thead>
<tr>
<th>Operation</th>
<th>Time (ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1 cache reference</td>
<td>0.5</td>
</tr>
<tr>
<td>Branch mispredict</td>
<td>5</td>
</tr>
<tr>
<td>L2 cache reference</td>
<td>7</td>
</tr>
<tr>
<td>Mutex lock/unlock</td>
<td>25</td>
</tr>
<tr>
<td>Main memory reference</td>
<td>100</td>
</tr>
<tr>
<td>Compress 1K bytes with Zippy</td>
<td>3,000</td>
</tr>
<tr>
<td>Send 2K bytes over 1 Gbps network</td>
<td>20,000</td>
</tr>
<tr>
<td>Read 1 MB sequentially from memory</td>
<td>250,000</td>
</tr>
<tr>
<td>Round trip within same datacenter</td>
<td>500,000</td>
</tr>
<tr>
<td>Disk seek</td>
<td>10,000,000</td>
</tr>
<tr>
<td>Read 1 MB sequentially from disk</td>
<td>20,000,000</td>
</tr>
<tr>
<td>Send packet CA-&gt;Netherlands-&gt;CA</td>
<td>150,000,000</td>
</tr>
</tbody>
</table>

Handle I/O in separate thread, avoid blocking other progress
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:

  ![Diagram showing thread activity with no I/O]

- If thread 1 performs a blocking I/O operation:

  ![Diagram showing thread activity with I/O operation]
A Better Example for Threads

• Version of program with threads (loose syntax):
  ```
  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, globals, 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...

System Call Interface

Portable OS Library

Application / Service

Portable OS Kernel

Platform support, Device Drivers

User

System

OS

Software

Hardware

x86

PowerPC

ARM

PCI

Ethernet (1Gbs/10Gbs) 802.11 a/g/n/ac SCSI Graphics Thunderbolt

Compilers

Word Processing

Web Browsers

Email

Web Servers

Databases

Email

Word Processing

Portable OS Library

OS

System

User

Software

Hardware

x86

PowerPC

ARM

PCI

Ethernet (1Gbs/10Gbs) 802.11 a/g/n/ac SCSI Graphics Thunderbolt
OS Library Issues Syscalls

Proc 1  Proc 2  ...  Proc n

OS

libc

OS library

Appln

login

Window Manager

OS
OS Library API for Threads: *pthreads*

*Here: the “p” is for “POSIX” which is a part of a standardized API*

```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

```c
void pthread_exit(void *value_ptr);
```

– terminates the thread and makes `value_ptr` available to any successful join

```c
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](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:

```c
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?
• What function does each thread run?
• One possible result:
  ```
  int common = 162;

  void *threadfun(void *threadid)
  {
      long tid = (long)threadid;
      printf("Thread %lx stack: %lx common: %lx \n", tid,
              (unsigned long) &tid, (unsigned long) &common, common++);
      pthread_exit(NULL);
  }

  int main (int argc, char *argv[])
  {
      long t;
      int nthreads = 2;
      if (argc > 1) {
          nthreads = atoi(argv[1]);
      }
      pthread_t *threads = malloc(nthreads*sizeof(pthread_t));
      printf("Main stack: %lx, common: %lx \n", tid, (unsigned long) &common);
      for(t=0; t<nthreads; t++){
          int rc = pthread_create(&threads[t], NULL, threadfun, (void *)t);
          if (rc){
              printf("ERROR; return code from pthread_create() is \n", rc);
              exit(-1);
          }
      }
      for(t=0; t<nthreads; t++){
          pthread_join(threads[t], NULL);
      }
      pthread_exit(NULL);
      /* last thing in the main thread */
  }
  ```

• Does the main thread join with the threads in the same order that they were created?
  • Yes: Loop calls Join in thread order
• Do the threads exit in the same order they were created?
  • No: Depends on scheduling order!
• Would the result change if run again?
  • Yes: Depends on scheduling order!
• Is this code safe/correct???
  • No – threads share are variable that is used without locking and there is a race condition!
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 \equiv \text{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
Shared vs. Per-Thread State

**Shared State**
- Heap
- Global Variables
- Code

**Per–Thread State**
- Thread Control Block (TCB)
  - Stack Information
  - Saved Registers
  - Thread Metadata
- Stack

**Per–Thread State**
- Thread Control Block (TCB)
  - Stack Information
  - Saved Registers
  - Thread Metadata
- Stack
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
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();
  }
}
B()
C()
A(1);
```

Stack Pointer

A: tmp=1
   ret=exit
B: ret=A+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);
```

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

```
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

- 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);
}
```

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);
}
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);
A(1);
```

Output: >2 1

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

```c
A(int tmp) {
    if (tmp<2) { B();
    printf(tmp);
}
B() {
B:   C();
B+1: }
C() {
C:   A(2);
C+1: }
exit: A(1);
A:   tmp=1
ret=exit
}
```

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);

- Stack holds temporary results
- Permits recursive execution
- Crucial to modern languages
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></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></td>
<td>y = y + x</td>
</tr>
<tr>
<td>z = x + 5y;</td>
<td>z = x + 5y;</td>
<td>thread is suspended</td>
<td>thread is suspended</td>
</tr>
<tr>
<td></td>
<td></td>
<td>other thread(s) run</td>
<td>other thread(s) run</td>
</tr>
<tr>
<td></td>
<td></td>
<td>thread is resumed</td>
<td>thread is resumed</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>y = y + x</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: Example 1

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

  **Thread B**  
  $y = 2;$

- What are the possible values of $x$ below after all threads finish?
- Must be 1. Thread B does not interfere
Race Conditions: Example 2

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

<table>
<thead>
<tr>
<th>Thread A</th>
<th>Thread B</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x = y + 1; )</td>
<td>( y = 2; )</td>
</tr>
<tr>
<td></td>
<td>( y = y \times 2; )</td>
</tr>
</tbody>
</table>

• 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()
OS Library Locks: *pthreads*

```c
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);
```

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);
}
```
Recall: UNIX System Structure

<table>
<thead>
<tr>
<th>User Mode</th>
<th>Kernel Mode</th>
<th>Hardware</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Applications</strong></td>
<td><strong>Standard Libs</strong></td>
<td><strong>Terminal Controllers</strong></td>
</tr>
<tr>
<td>(the users)</td>
<td>shells and commands</td>
<td>terminals</td>
</tr>
<tr>
<td></td>
<td>compilers and interpreters</td>
<td></td>
</tr>
<tr>
<td></td>
<td>system libraries</td>
<td></td>
</tr>
<tr>
<td><strong>Kernel Mode</strong></td>
<td><strong>Signals</strong></td>
<td><strong>Device Controllers</strong></td>
</tr>
<tr>
<td><strong>Kernel</strong></td>
<td>handling</td>
<td>disks and tapes</td>
</tr>
<tr>
<td></td>
<td>character I/O system</td>
<td></td>
</tr>
<tr>
<td></td>
<td>terminal drivers</td>
<td></td>
</tr>
<tr>
<td><strong>Kernel Interface to the Hardware</strong></td>
<td>signals handling</td>
<td><strong>Memory Controllers</strong></td>
</tr>
<tr>
<td></td>
<td>file system</td>
<td>physical memory</td>
</tr>
<tr>
<td></td>
<td>swapping block I/O system</td>
<td></td>
</tr>
<tr>
<td></td>
<td>system</td>
<td></td>
</tr>
<tr>
<td></td>
<td>disk and tape drivers</td>
<td></td>
</tr>
<tr>
<td><strong>System-call Interface to the Kernel</strong></td>
<td>CPU scheduling</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>page replacement</td>
</tr>
<tr>
<td></td>
<td></td>
<td>demand paging</td>
</tr>
<tr>
<td></td>
<td></td>
<td>virtual memory</td>
</tr>
</tbody>
</table>
Recall: Life of a Process

User Mode
- syscall
- interrupt
- exception
- exec
- rtn
- rfi
- exit

Kernel Mode
- Limited HW access
- Full HW access
3 types of Kernel 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
  - eg. 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, …
Implementing Safe Kernel Mode Transfers

• Important aspects:
  – Controlled transfer into kernel (e.g., syscall table)
  – Separate kernel stack!

• Carefully constructed kernel code packs up the user process state and sets it aside
  – Details depend on the machine architecture
  – More on this next time

• Should be impossible for buggy or malicious user program to cause the kernel to corrupt itself!
Recall: System Call Interface: A Narrow Waist

- Compilers
- Word Processing
- Email
- Databases
- Web Browsers
- Web Servers
- Portable OS Library
- System Call Interface
- Portable OS Kernel
- Platform support, Device Drivers
- User Mode
- System Mode
- OS
- Software
- Hardware
- x86
- PowerPC
- ARM
- Ethernet (1Gbs/10Gbs)
- 802.11 a/g/n/ac/ax
- SCSI
- Graphics Thunderbolt
Kernel System Call Handler

- Vector through well-defined syscall entry points!
  - Table mapping system call number to handler
- Locate arguments
  - In registers or on user (!) stack
- Copy arguments
  - From user memory into kernel memory
  - Protect kernel from malicious code evading checks
- Validate arguments
  - Protect kernel from errors in user code
- Copy results back
  - Into user memory
Hardware support: Interrupt Control

• Interrupt processing not visible to the user process:
  – Occurs between instructions, restarted transparently
  – No change to process state
  – What can be observed even with perfect interrupt processing?

• Interrupt Handler invoked with interrupts ‘disabled’
  – Re-enabled upon completion
  – Non-blocking (run to completion, no waits)
  – Pack up in a queue and pass off to an OS thread for hard work
    » wake up an existing OS thread
Interrupt Controller

- Interrupts invoked with interrupt lines from devices
- Interrupt controller chooses interrupt request to honor
  - Interrupt identity specified with ID line
  - Mask enables/disables interrupts
  - Priority encoder picks highest enabled interrupt
  - Software Interrupt Set/Cleared by Software
- CPU can disable all interrupts with internal flag
- Non-Maskable Interrupt line (NMI) can’t be disabled
How do we take interrupts safely?

- **Interrupt vector**
  - Limited number of entry points into kernel

- **Kernel interrupt stack**
  - Handler works regardless of state of user code

- **Interrupt masking**
  - Handler is non-blocking

- **Atomic transfer of control**
  - “Single instruction”-like to change:
    - Program counter
    - Stack pointer
    - Memory protection
    - Kernel/user mode

- **Transparent restartable execution**
  - User program does not know interrupt occurred
Interrupt Vector

- Where else do you see this dispatch pattern?
  - System Call
  - Exceptions

intrpHandler_i () {
    ...
}

interrupt number (i)

Address and properties of each interrupt handler
Need for Separate Kernel Stacks

- Kernel needs space to work
- Cannot put anything on the user stack (Why?)
- Two-stack model
  - OS thread has interrupt stack (located in kernel memory) plus User stack (located in user memory)
  - Syscall handler copies user args to kernel space before invoking specific function (e.g., open)
  - Interrupts (???)
Before

User-level Process

code:

foo () {
    while(...) {
        x = x+1;
        y = y-2;
    }
}

stack:

Registers

SS: ESP
CS: EIP
EFLAGS
other
 registers: EAX, EBX,
...

Kernel

code:

handler() {
    pusha
    ...
}

Exception Stack
During Interrupt/System Call

User-level Process

code:

foo () {
    while(...) {
        x = x+1;
        y = y-2;
    }
}

stack:

Registers

Kernel

code:

handler() {
    pusha
    ...
}

Exception Stack

SS
ESP
EFLAGS
CS
EIP
error
Managing Processes

• How to manage process state?
  – How to create a process?
  – How to exit from a process?

• Remember: Everything outside of the kernel is running in a process!
  – Including the shell! (Homework 2)

• Processes are created and managed… by processes!
Bootstrapping

• If processes are created by other processes, how does the first process start?

• First process is started by the kernel
  – Often configured as an argument to the kernel *before* the kernel boots
  – Often called the “init” process

• After this, all processes on the system are created by other processes
Next time: Process Management API

- exit – terminate a process
- fork – copy the current process
- exec – change the program being run by the current process
- wait – wait for a process to finish
- kill – send a signal (interrupt-like notification) to another process
- sigaction – set handlers for signals
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

• Processes consist of one or more threads in an address space
  – Abstraction of the machine: execution environment for a program
  – Can use `fork`, `exec`, etc. to manage threads within a process

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