CS162
Operating Systems and
Systems Programming
Lecture 3

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

• **Address space** (with or w/o translation)
  – Set of memory addresses accessible to program

• **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
Recall: Illusion of Multiple Processors

- Threads are *virtual cores*

- Multiple threads: *Multiplex* hardware in time

- Each virtual core (thread) has:
  - Program counter (PC), stack pointer (SP)
  - Registers

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

- Address space ⇒ the set of accessible addresses + state associated with them:
  - For 32-bit processor: \(2^{32} = 4 \text{ billion (}10^9\text{)} \) addresses
  - For 64-bit processor: \(2^{64} = 18 \text{ quintillion (}10^{18}\text{)} \) addresses

- Virtual Address Space ⇒ Processor’s view of memory:
  - Address Space is independent of physical storage
Recall: Process

- **Definition:** execution environment with Restricted Rights
  - One or more threads executing 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
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

• 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
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

• Threads are a mechanism for concurrency and parallelism

• 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
  – Processes, interrupts, background system maintenance

• Networked servers must handle concurrent requests

• Parallel programs must parallelise for performance

• Programs with user interface need threading to ensure responsiveness

• Network and disk bound programs use threading to hide network/disk latency
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
- Parallelism is about doing multiple things *simultaneously*
- Example: Two threads on a single-core system...
  - ... execute concurrently ...
  - ... but *not* in parallel
- Parallel => concurrent, but not the other way round!
Silly Example for Threads

• Imagine the following program:

```java
main() {
    ComputePI("pi.txt");
    PrintClassList("classlist.txt");
}
```

• What is the behaviour here?

• Program would never print out class list

• Why? `ComputePI` would never finish
Adding Threads

• Version of program with threads (loose syntax):
  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
More Practical Motivation: Compute/IO overlap

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

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

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

- Compilers
- Web Browsers
- Email
- Web Servers
- Databases
- Word Processing
- Email
- Web Servers
- Databases
- Portable OS Library
- System Call Interface
- Portable OS Kernel
- Platform support, Device Drivers
- OS
- Application / Service

Hardware
- x86
- PowerPC
- ARM
- PCI
- Ethernet (1Gbs/10Gbs)
- 802.11 a/g/n/ac
- SCSI
- Graphics
- Thunderbolt

Software
- User
- System
OS Library Issues Syscalls

Diagram showing the interaction between processes and the OS library. Processes (Proc 1, Proc 2, ... Proc n) interact with the OS through the OS library. The OS library includes components like libc, Appln, login, and Window Manager.
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`

```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`. 
Peeking Ahead: System Call Example

- What happens when \texttt{pthread\_create(...)} is called in a process?

\begin{verbatim}
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...
    }
\end{verbatim}
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?
Shared vs. Per-Thread State

<table>
<thead>
<tr>
<th>Shared State</th>
<th>Per–Thread State</th>
<th>Per–Thread State</th>
</tr>
</thead>
<tbody>
<tr>
<td>Heap</td>
<td>Thread Control Block (TCB)</td>
<td>Thread Control Block (TCB)</td>
</tr>
<tr>
<td>Global Variables</td>
<td>Stack Information</td>
<td>Stack Information</td>
</tr>
<tr>
<td>Code</td>
<td>Saved Registers</td>
<td>Saved Registers</td>
</tr>
<tr>
<td></td>
<td>Thread Metadata</td>
<td>Thread Metadata</td>
</tr>
<tr>
<td></td>
<td>Stack</td>
<td>Stack</td>
</tr>
</tbody>
</table>
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

```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
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
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);
```

Stack Growth:

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

Stack Pointer:

- 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);
}

Output: >2

- 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() {
    C();
}

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

Output: \textgreater{}2

- 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() {
    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

- 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 1`

Stack Pointer

A: tmp=1
ret=exit
Execution Stack Example

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

Output: \texttt{>2 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
Possible Executions

Thread 1
Thread 2
Thread 3
a) One execution

Thread 1
Thread 2
Thread 3
b) Another execution

c) Another execution
## 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>( 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>( \ldots )</td>
<td>( \ldots )</td>
<td>( \ldots )</td>
<td>( \ldots )</td>
</tr>
<tr>
<td>( \ldots )</td>
<td>( \ldots )</td>
<td>( \ldots )</td>
<td>( \ldots )</td>
</tr>
</tbody>
</table>

\( \text{thread is suspended} \)

\( \text{other thread(s) run} \)

\( \text{thread is resumed} \)

\( \text{thread is suspended} \)

\( \text{other thread(s) run} \)

\( \text{thread is resumed} \)
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 \( y = 0 \)

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

- 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 $y = 0$

  **Thread A**  **Thread B**
  $x = y + 1;$  $y = 2;$
  $y = y \times 2;$

• What are the possible values of $x$ below?
  • 1 or 3 or 5 (non-deterministically)
  • Race Condition: Thread A races against Thread B!
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
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);
}
```
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!
Recall: Life of a Process?

User Mode

Kernel Mode

exec
syscall
rtn
rfi
interrupt
exception
exit

Limited HW access

Full HW access
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
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
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
```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>

int main(int argc, char *argv[]) {
    /* get current processes PID */
    pid_t pid = getpid();
    printf("My pid: %d\n", pid);

    exit(0);  
}
```

Q: What if we let main return without ever calling exit?

- The OS Library calls exit() for us!
- The entrypoint of the executable is in the OS library
- OS library calls main
- If main returns, OS library calls exit
- You’ll see this in Project 0: init.c
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
Creating Processes

- `pid_t fork()` – copy the current process
  - New process has different pid
  - New process contains a single thread

State of original process duplicated in both Parent and Child!
Address Space (Memory), File Descriptors (covered later), etc...
fork1.c

#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

int main(int argc, char *argv[]) {
    pid_t cpid, mypid;
    pid_t pid = getpid();            /* get current processes PID */
    printf("Parent pid: %d\n", pid);
    cpid = fork();
    if (cpid > 0) { /* Parent Process */
        mypid = getpid();
        printf("[%d] parent of [%d]\n", mypid, cpid);
    } else if (cpid == 0) { /* Child Process */
        mypid = getpid();
        printf("[%d] child\n", mypid);
    } else {
        perror("Fork failed");
    }
}
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

int main(int argc, char *argv[]) {
    pid_t cpid, mypid;
    pid_t pid = getpid();            /* get current processes PID */
    printf("Parent pid: %d\n", pid);
    cpid = fork();
    if (cpid > 0) { /* Parent Process */
        mypid = getpid();
        printf("[%d] parent of [%d]\n", mypid, cpid);
    } else if (cpid == 0) { /* Child Process */
        mypid = getpid();
        printf("[%d] child\n", mypid);
    } else {
        perror("Fork failed");
    }
}
```c
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

int main(int argc, char *argv[]) {
    pid_t cpid, mypid;
    pid_t pid = getpid();            /* get current processes PID */
    printf("Parent pid: %d\n", pid);
    cpid = fork();
    if (cpid > 0) { /* Parent Process */
        mypid = getpid();
        printf("[%d] parent of [%d]\n", mypid, cpid);
    } else if (cpid == 0) { /* Child Process */
        mypid = getpid();
        printf("[%d] child\n", mypid);
    } else {
        perror("Fork failed");
    }
}
```
int i;
pid_t cpid = fork();
if (cpid > 0) {
    for (i = 0; i < 10; i++) {
        printf("Parent: %d\n", i);
        // sleep(1);
    }
} else if (cpid == 0) {
    for (i = 0; i > -10; i--) {
        printf("Child: %d\n", i);
        // sleep(1);
    }
}

• What does this print?
• Would adding the calls to sleep() matter?

Recall: a process consists of one or more threads executing in an address space
• Here, each process has a single thread
• These threads execute concurrently
Running Another Program

• With threads, we could call `pthread_create` to create a new thread executing a separate function

• With processes, the equivalent would be spawning a new process executing a different program

• How can we do this?
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
```c
... cpid = fork();
if (cpid > 0) { /* Parent Process */
    tcpid = wait(&status);
} else if (cpid == 0) { /* Child Process */
    char *args[] = {"ls", "-l", NULL};
    execv("/bin/ls", args);
    /* execv doesn’t return when it works.
     * So, if we got here, it failed! */
    perror("execv");
    exit(1);
}
...
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
fork2.c – parent waits for child to finish

```c
int status;
pid_t tcpid;
...
cpid = fork();
if (cpid > 0) {               /* Parent Process */
    mypid = getpid();
    printf("[%d] parent of [%d]\n", mypid, cpid);
    tcpid = wait(&status);
    printf("[%d] bye %d(%d)\n", mypid, tcpid, status);
} else if (cpid == 0) {      /* Child Process */
    mypid = getpid();
    printf("[%d] child\n", mypid);
    exit(42);
}
...
```
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
inf_loop.c

#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <signal.h>

void signal_callback_handler(int signum) {
    printf("Caught signal!\n");
    exit(1);
}

int main() {
    struct sigaction sa;
    sa.sa_flags = 0;
    sigemptyset(&sa.sa_mask);
    sa.sa_handler = signal_callback_handler;
    sigaction(SIGINT, &sa, NULL);
    while (1) {}
}

Q: What would happen if the process receives a SIGINT signal, but does not register a signal handler?
A: The process dies!

For each signal, there is a default handler defined by the system
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
**Administrivia: Getting started**

- Should be working on Homework 0 already! ⇒ Due Wednesday (27/1)
  - cs162-xx account, Github account, registration survey
  - Vagrant and VirtualBox – VM environment for the course
    » Consistent, managed environment on your machine
  - Get familiar with all the cs162 tools, submit to autograder via git

- Start working on Project 0 now! ⇒ Due Monday 01/02
  - To be done on your own – like a homework!

- Friday (29/1) is drop day!
  - Very hard to drop afterwards…
  - Please drop sooner if you are going to anyway ⇒ Let someone else in!