Abstractions 1: Threads and Processes

Sam Kumar
CS 162: Operating Systems and System Programming
Lecture 3
https://inst.eecs.berkeley.edu/~cs162/su20

Read: A&D 3.1, 5.1-5.3
Recall: Four Fundamental OS Concepts

• **Thread: Execution Context**
  • Program Counter, Registers, Execution Flags, Stack

• **Address Space (with Translation)**
  • Program’s view of memory is distinct from physical machine

• **Process: Instance of a Running Program**
  • Address space + one or more threads + ...

• **Dual-Mode Operation and Protection**
  • Only the “system” can access certain resources
  • Combined with translation, isolates programs from each other
Recall: Thread

• Definition: A single, unique execution context
  • Program counter, registers, stack

• A thread is the OS abstraction for a CPU core
  • A “virtual CPU” of sorts

• Registers hold the root state of the thread:
  • Including program counter – pointer to the currently executing instruction
  • The rest is “in memory”

• Registers point to thread state in memory:
  • Stack pointer to the top of the thread’s (own) stack
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

On a single physical CPU:

- Each virtual core (thread) has PC, SP, Registers
- Where is it?
  - On the real (physical) core, or
  - Saved in memory – called the Thread Control Block (TCB)
Recall: Address Space

- Program operates in an address space that is distinct from the physical memory space of the machine
Recall: Process

• Definition: execution environment with restricted rights
  • One or more threads executing in a single address space
  • Owns 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

• Protected from each other; OS protected from them

• In modern OSes, anything that runs outside of the kernel runs in a process
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)

• Together, address translation and dual-mode operation allow the kernel to protect processes from each other and itself from processes
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
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*
  • Also a valid definition!

• Threads are a mechanism for *concurrency*

• 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
Threads Allow Handling MTAO

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

- Multiprocessing: Multiple 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
Silly Example for Threads

• Imagine the following program:
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

CPU1 CPU2 CPU1 CPU2 CPU1 CPU2

Time
More Practical Motivation

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 (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>
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:
Little 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…
OS Library Issues Syscalls

Proc 1  Proc 2  ...  Proc n

OS

libc

OS library

OS library

OS library

OS

AppIn

login

Window Manager

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`

```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`.

[Man page](man pthread)
[Website](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:**

```c
  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...
};
```
Threads 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?
Fork-Join Pattern

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

• Homework 0 due tomorrow night
  • Start on it soon if you have not done so already

• Quiz 0 available on online exam platform today right after lecture

• C Review Session tomorrow 6-7 PM

• Reminder to look for groups
  • “Search for Teammates” thread on Piazza
  • Work with prospective teammates on Project 0
Interleaving and Nondeterminism
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

Programmer’s View

Possible Execution #1

. .

. .

. .

x = x + 1;
y = y + x;
z = x + 5y;

Possible Execution #2

. .

. .

. .

x = x + 1

thread is suspended
other thread(s) run
thread is resumed

Possible Execution #3

. .

. .

. .

x = x + 1

y = y + x

thread is suspended
other thread(s) run
thread is resumed

y = y + x
z = x + 5y

z = x + 5y
Possible Executions

Thread 1

Thread 2

Thread 3

a) One execution

Thread 1

Thread 2

Thread 3

b) Another execution

Thread 1

Thread 2

Thread 3

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

• What are the possible values of $x$ below after all threads finish?
• Initially $x \ = \ 0$ and $y \ = \ 0$

Thread A  Thread B
$x \ = \ 1; \quad y \ = \ 2;$

• Must be 1. Thread B does not interfere.
Race Conditions

• What are the possible values of x below?
• 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 * 2;</td>
</tr>
</tbody>
</table>

• 1 or 3 or 5 (non-deterministic)

• 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
Example: Shared Data Structure

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: *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);
}
```
Semaphore

• Semaphores are a kind of generalized lock
  • First defined by Dijkstra in late 60s
  • Main synchronization primitive used in original UNIX (& Pintos)
• Definition: a Semaphore has a non-negative integer value and supports the following two operations:
  • P() or down(): atomic operation that waits for semaphore to become positive, then decrements it by 1
  • V() or up(): an atomic operation that increments the semaphore by 1, waking up a waiting P, if any

P() stands for “proberen” (to test) and V() stands for “verhogen” (to increment) in Dutch
Two Important Semaphore Patterns

• **Mutual Exclusion:** (Like lock)
  • Called a "binary semaphore"
    
    ```
    initial value of semaphore = 1;
    semaphore.down();
    // Critical section goes here
    semaphore.up();
    ```

• **Signaling** other threads, e.g. **ThreadJoin**

  ```
  Initial value of semaphore = 0
  ThreadJoin {
    semaphore.down();
  }
  ThreadFinish {
    semaphore.up();
  }
  ```
Break (If Time)
Recall: Process

• Definition: execution environment with restricted rights
  • One or more threads executing in a single address space
  • Owns 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

• Protected from each other; OS protected from them

• In modern OSes, anything that runs outside of the kernel runs in a process
Recall: Life of a Process

User Mode

Kernel Mode

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

Limited HW access

Full HW access
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

• 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
pid.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

• Return value from `fork()`: `pid` (like an integer)
  • When > 0:
    • Running in (original) `Parent` process
    • return value is `pid` of new child
  • When = 0:
    • Running in new `Child` process
  • When < 0:
    • Error! Must handle somehow
    • Running in original process

• 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");
    }
}

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");
    }
}
fork1.c

```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");
    }
}
```
fork_race.c

```c
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);
    }
}
```

Recall: a process consists of one or more threads executing in an address space
- In this case, each process has a single thread
- These threads execute concurrently

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

```
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);
}
...
```
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
fork3.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
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.
Common POSIX Signals

• SIGINT – control-C
• SIGTERM – default for kill shell command
• SIGSTP – control-Z (default action: stop process)

• SIGKILL, SIGSTOP – terminate/stop process
  • Can’t be changed with sigaction
  • Why?
Shell

- A shell is a job control system
  - Allows programmer to create and manage a set of programs to do some task

- You will build your own shell in Homework 2...
  - ... using fork and exec system calls to create new processes...
  - ... and the File I/O system calls we’ll see next time to link them together
Process vs. Thread APIs

• Why have `fork()` and `exec()` system calls for processes, but just a `pthread_create()` function for threads?
  • Convenient to fork without exec: put code for parent and child in one executable instead of multiple
  • It will allow us to programmatically control child process’ state
    • By executing code before calling `exec()` in the child
    • We’ll see this in the case of File I/O next time

• Windows uses `CreateProcess()` instead of `fork()`
  • Also works, but a more complicated interface
Threads vs. Processes

• If we have two tasks to run concurrently, do we run them in separate threads, or do we run them in separate processes?

• Depends on how much isolation we want
  • Threads are lighter weight [why?]
  • Processes are more strongly isolated
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