

CS162
Operating Systems and
Systems Programming
Lecture 4

Abstractions 2: Threads (Con't)
Process Management,
A quick programmer's viewpoint

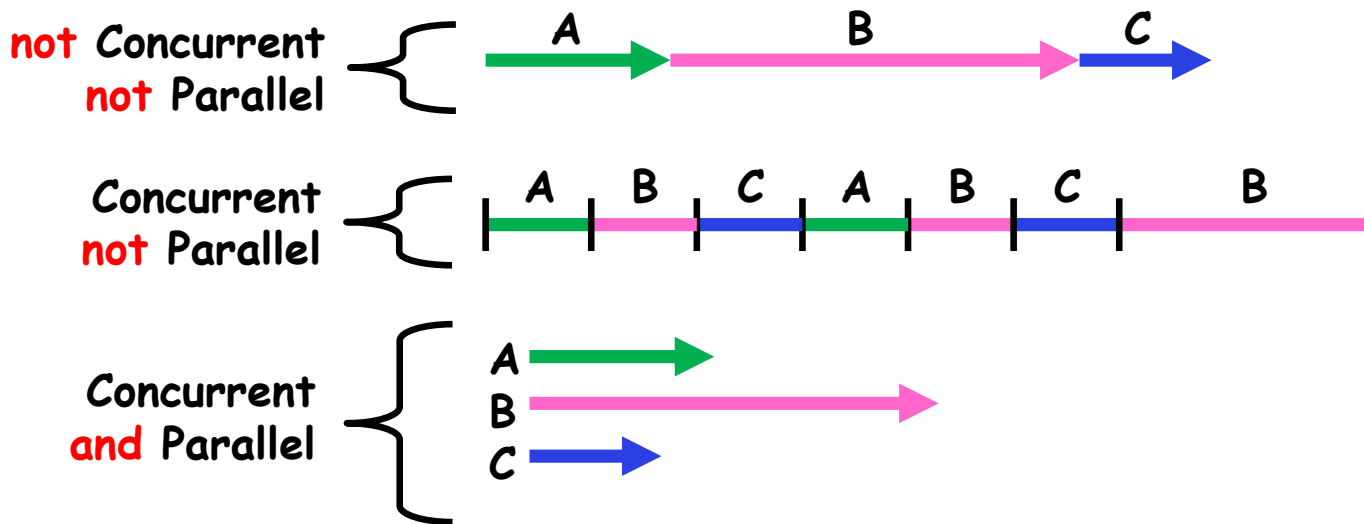
January 29th, 2026

Prof. John Kubiatowicz

<http://cs162.eecs.Berkeley.edu>

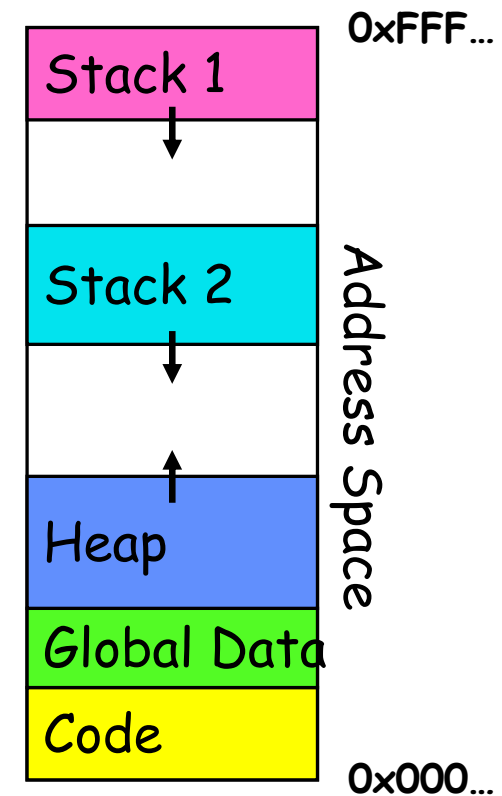
Recall: Concurrency vs. Parallelism

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



Memory Layout with Two Threads

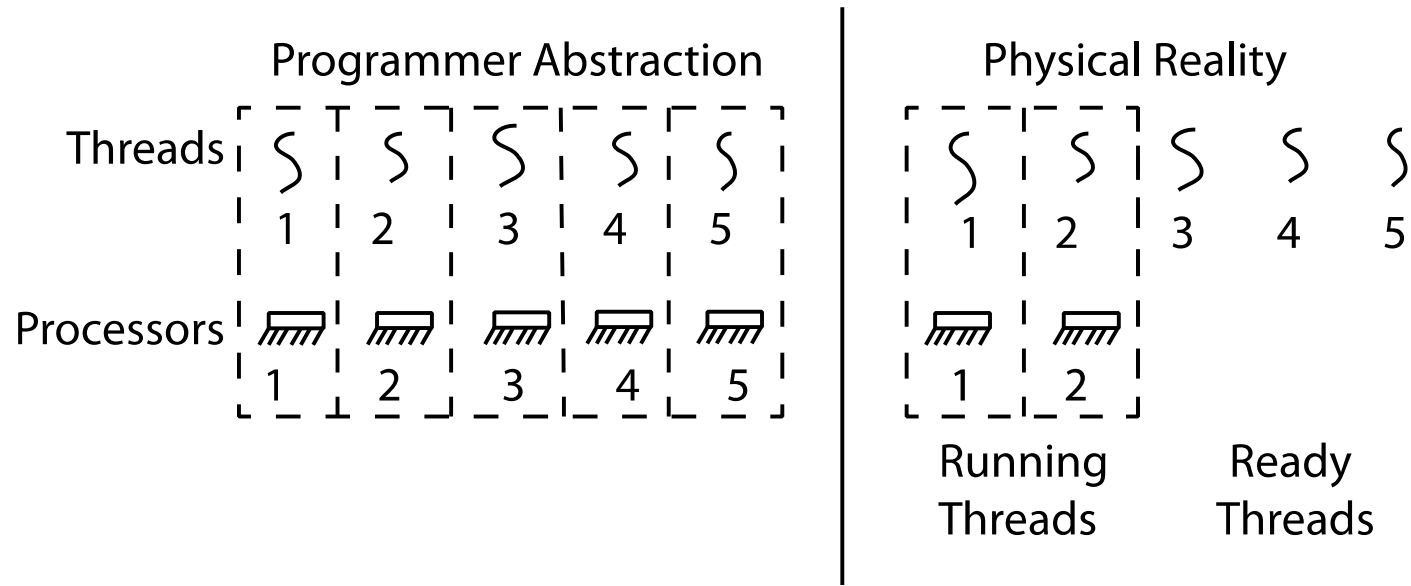
- Two sets of CPU registers
- Two sets of Stacks
- Issues:
 - How do we position stacks relative to each other?
 - What maximum size should we choose for the stacks?
 - What happens if threads violate this?
 - How might you catch violations?



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

Programmer's
View

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

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
.....
y = y + x
z = x + 5y

Possible
Execution
#3

·
·
·
x = x + 1
y = y + x
.....
thread is suspended
other thread(s) run
thread is resumed
.....
z = x + 5y

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.
- Threads are Independent!

Race Conditions: Example 2

- Initially $x == 0$ and $y == 0$

Thread A

$x = y + 1;$

Thread B

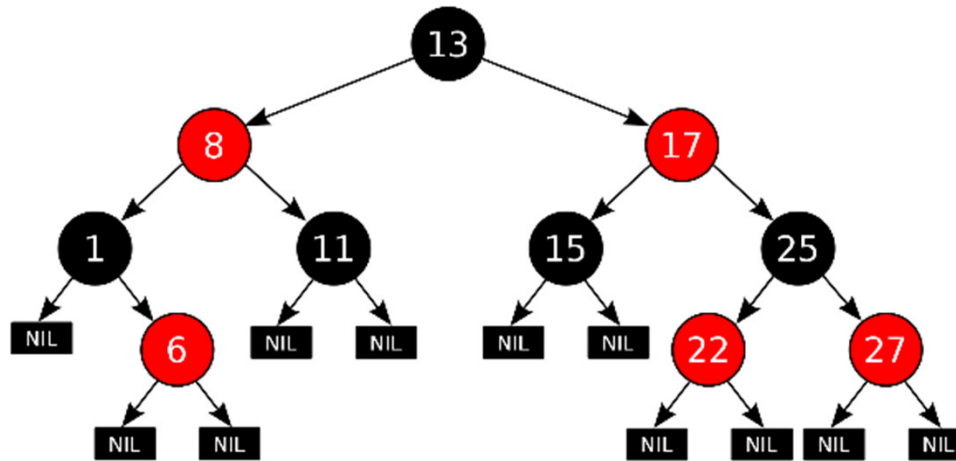
$y = 2;$

$y = y * 2;$

- What are the possible values of x below?
- 1 or 3 or 5 (non-deterministically)
- Race Condition: Thread A races against Thread B!
- Threads are Cooperating / Not Independent!

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

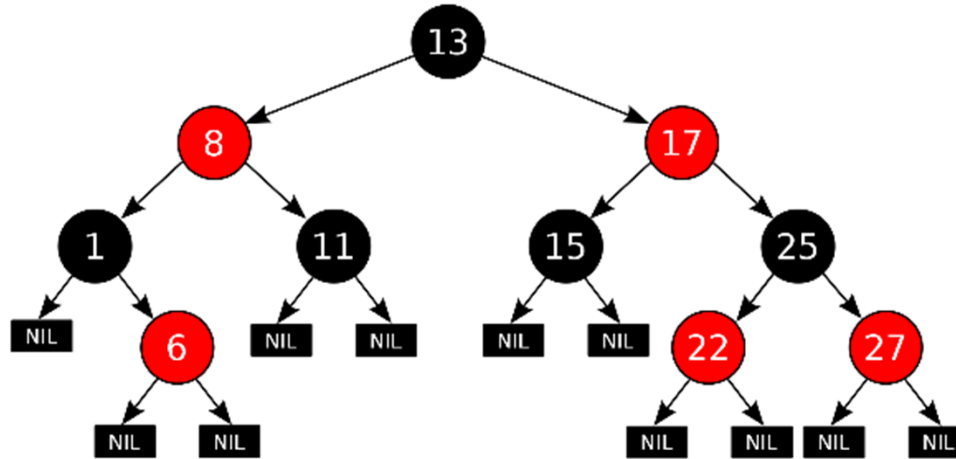
Locks

- 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

Recall: pThreads Example

- How many threads are in this program?
- What function does each thread run?
- One possible result:

```
[(base) CullerMac19:code04 culler$ ./pthread 4
Main stack: 7ffee2c6b6b8, common: 10cf95048 (162)
Thread #1 stack: 70000d83bef8 common: 10cf95048 (162)
Thread #3 stack: 70000d941ef8 common: 10cf95048 (164)
Thread #2 stack: 70000d8beef8 common: 10cf95048 (165)
Thread #0 stack: 70000d7b8ef8 common: 10cf95048 (163)
```
- 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 a variable that is used without locking and there is a race condition!

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <string.h>

int common = 162;

void *threadfun(void *threadid)
{
    long tid = (long)threadid;
    printf("Thread #%lx stack: %lx common: %lx (%d)\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 (%d)\n",
        (unsigned long) &t, (unsigned long) &common, 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 %d\n", rc);
            exit(-1);
        }
    }

    for(t=0; t<nthreads; t++){
        pthread_join(threads[t], NULL);
    }
    pthread_exit(NULL);          /* last thing in the main thread */
}
```

OS Library Locks: *pthread*s

```
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

Fixing Our Example

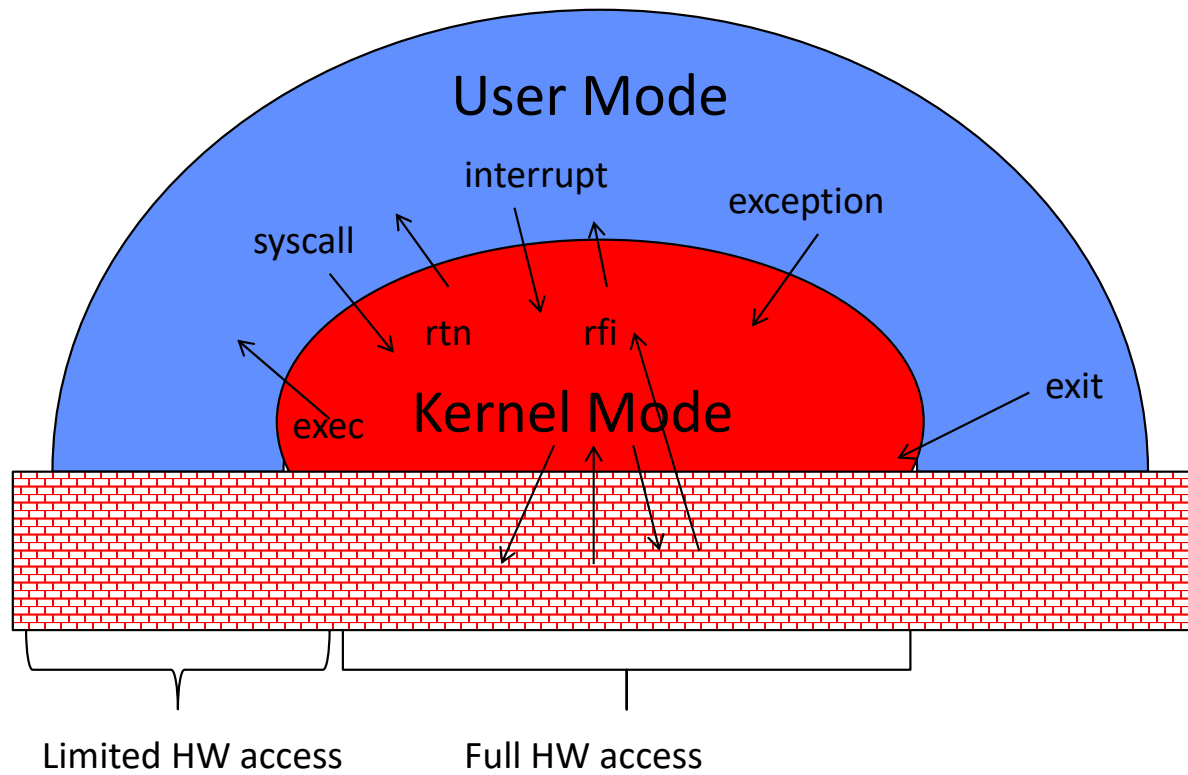
Critical section

```
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: User \Rightarrow Kernel Mode Transfer



3 types of User \Rightarrow Kernel Mode Transfer

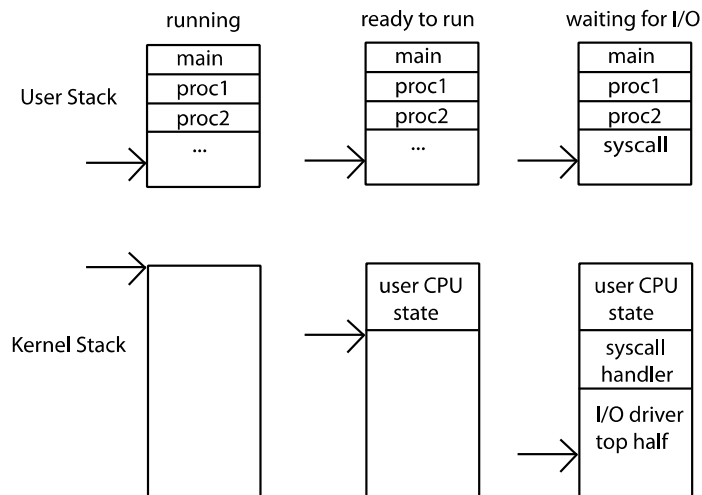
- Syscall
 - Process requests a system service, e.g., exit
 - Like a function call, but “outside” the process
 - Does not have the address of the system function to call
 - Like a Remote Procedure Call (RPC) – for later
 - Marshall the syscall id and args in registers and exec syscall
- Interrupt
 -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

Implementing Safe Kernel Mode Transfers

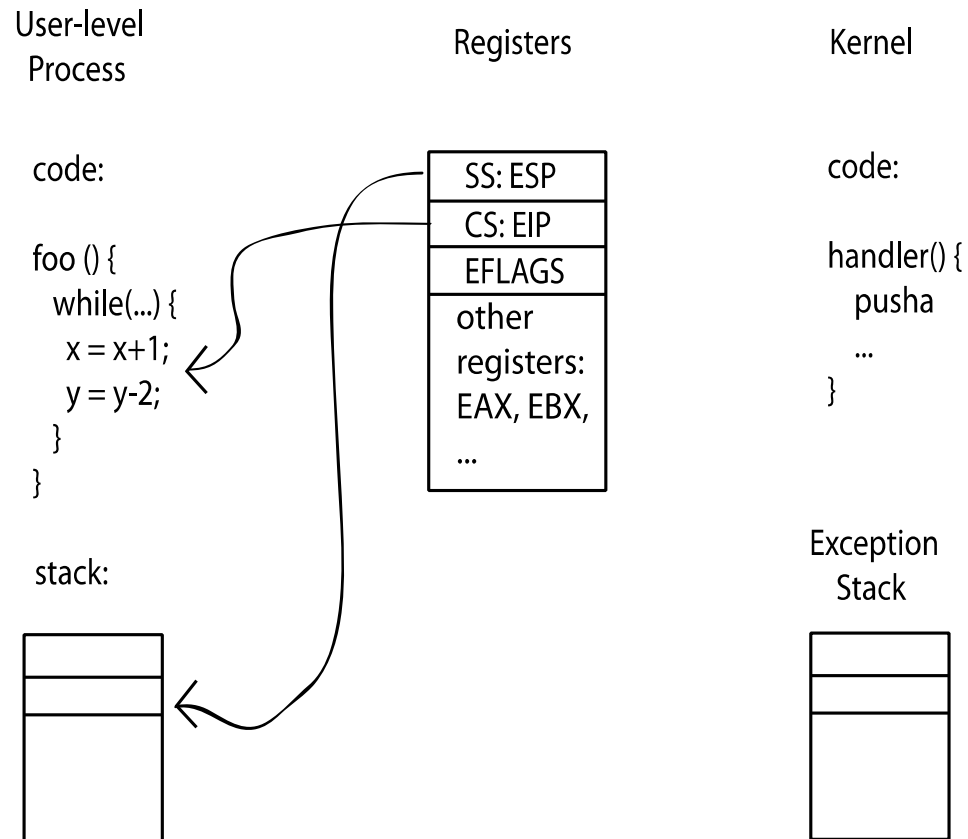
- Should be impossible for buggy or malicious user program to cause the kernel to corrupt itself!
- Important security aspects:
 - Atomic transfer to well defined kernel entry point (handler) while entering kernel mode
 - Separate kernel stack (don't trust user's stack pointer)
- How to restrict kernel entrypoints?
 - Use indirection table: Interrupt handler table and system call table
- Kernel handler packs up the user process state and sets it aside before starting
 - Details depend on the machine architecture

Need for Separate Kernel Stacks

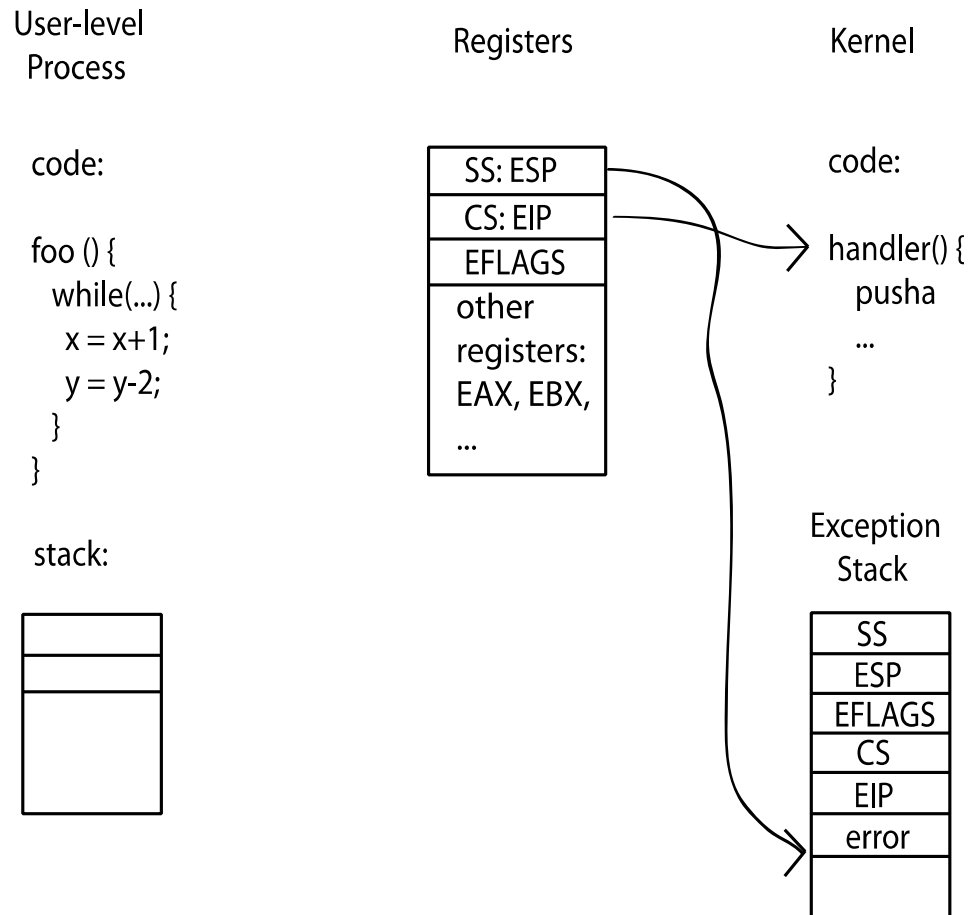
- Kernel needs space to work
- Kernel cannot put anything on the user stack (Why?)
 - Can't trust stack pointer to point at anything useful
 - Can't trust other threads not to mess with information on user stack
- Two-stack model
 - Each thread has both User stack (located in user memory) *and* Kernel stack (located in kernel memory)



Before Interrupt/System Call



During Interrupt/System Call

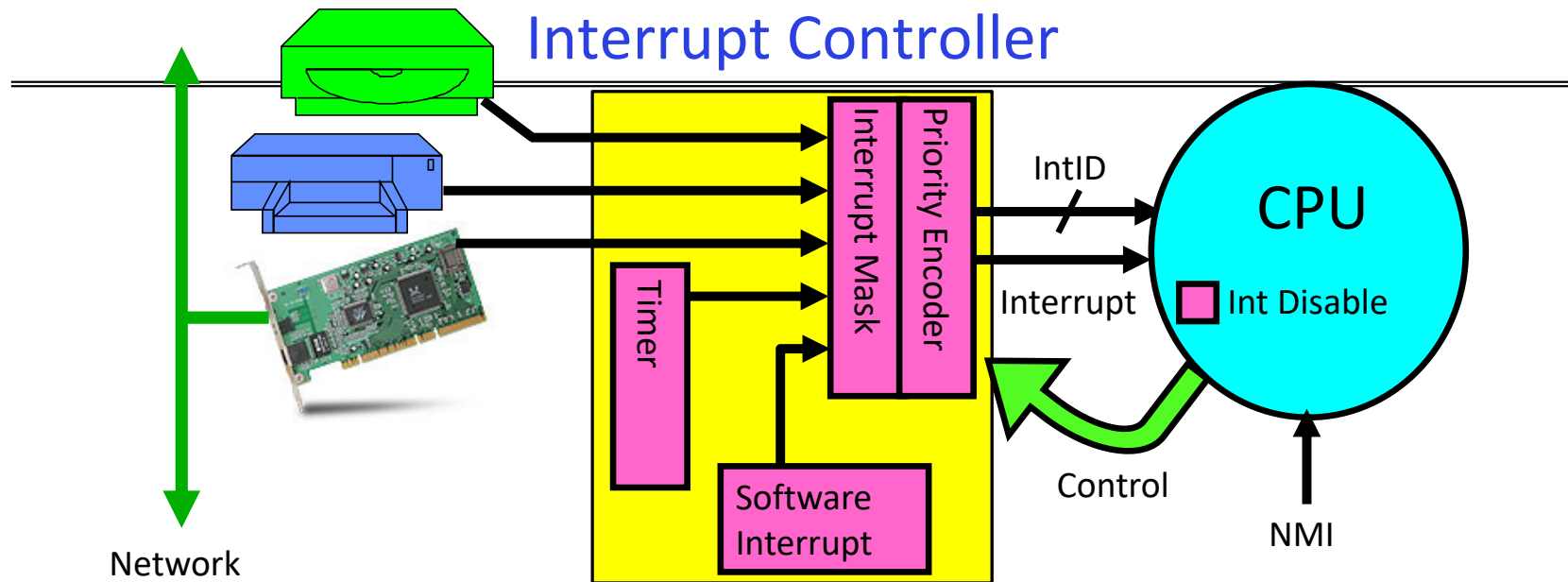


Kernel System Call Handler: Safe Entry into Kernel

- Vector through well-defined syscall entry points!
 - Table mapping system call number to handler
- System call stack
 - System call handler works regardless of state of user SP register
- 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

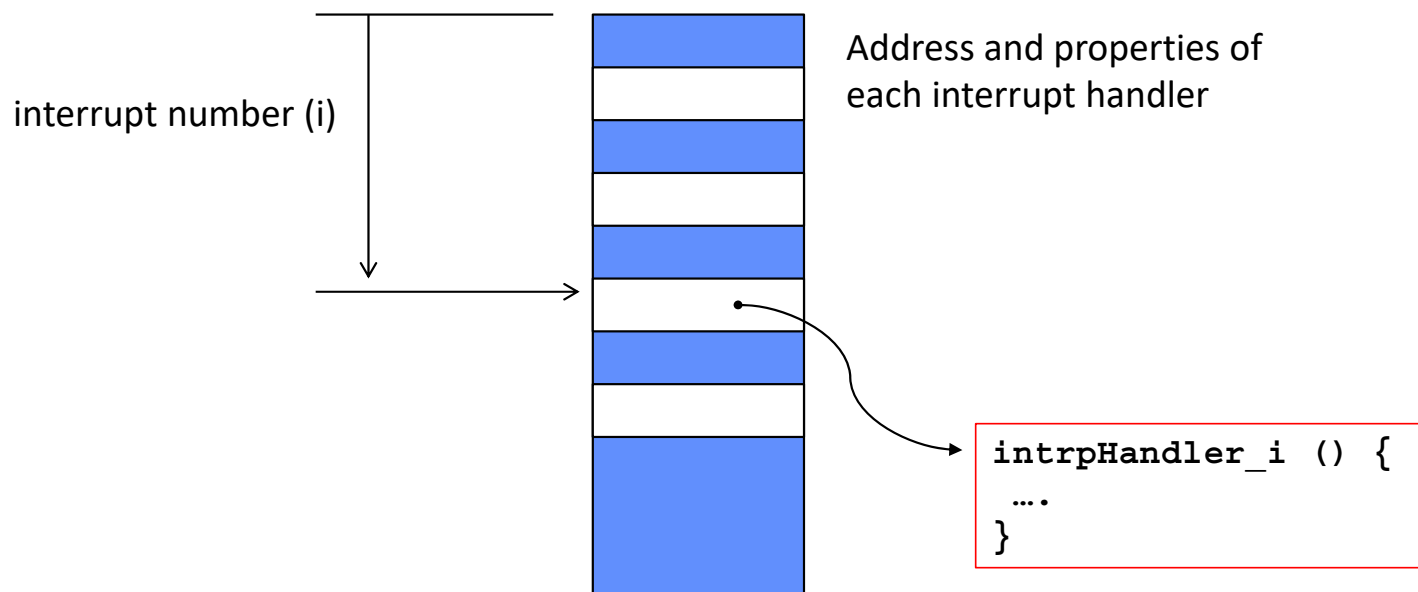
Interrupt Control: Also Safe Entry into Kernel

- Vector through table of Interrupt handlers
 - Table mapping *interrupt* number to handler
- Kernel interrupt stack
 - Interrupt handler works regardless of state of user SP register
- 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



- Hardware Interrupts invoked with Interrupt lines from devices (including Timer)
- Software Interrupt Set/Cleared by Software
- Interrupt controller chooses interrupt request to honor
 - Interrupt identity specified with ID line
 - Mask enables/disables interrupts
 - Priority encoder picks highest enabled interrupt
- CPU can disable all interrupts with internal flag
- Except: Non-Maskable Interrupt line (NMI) can't be disabled

Interrupt Vector



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

Administrivia

- Kubiawicz Office Hours
 - 1pm-2pm, Tuesday/Thursday
- **TOMORROW (Friday) is Drop Deadline! VERY HARD TO DROP LATER!**
- Recommendation: Read assigned readings *before* lecture

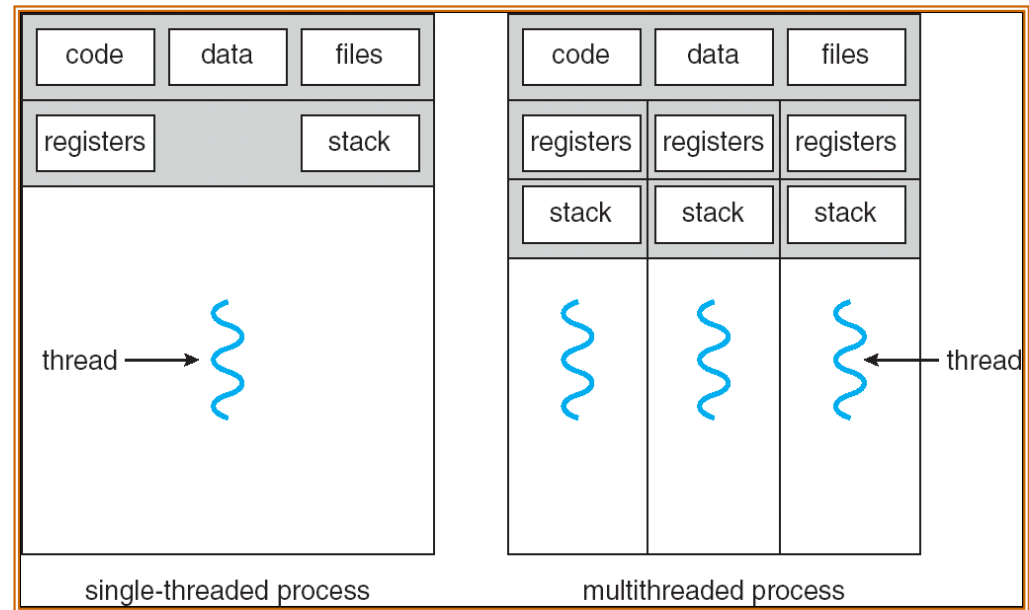
- You should be going to sections – Important information covered in section
 - Any section will do until groups assigned
- Get finding groups of 4 people ASAP
 - Priority for same section; try to make this work!
 - Remember: Your TA needs to see you in section!

Administrivia (Con't)

- Starting next week, we will be adhering to strict slip-day policies for non-DSP students
 - Slip days are no-questions asked (or justification needed) extensions
 - Anything beyond this requires documentation (i.e. doctor's note, etc)
 - If you run out of slip days, assignments will be discounted
 - » 10% first day, 20% second day, 40% third day, 80% fourth day
- You get 5 slip days for homework and 5 slip days for group projects
 - No project extensions on design documents, since we need to keep design reviews on track
 - Conserve your slip days!
- Midterm 1 will be on 2/24 from 7-10pm
 - No class on day of midterm (extra office hours!)
 - Closed book
 - One page of *handwritten* notes – both sides

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

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...

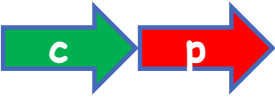
fork1.c

```
#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

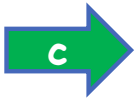
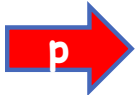
```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

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

```
#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");
    }
}
```



Mystery: fork_race.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

- Here, 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

Starting new Program: variants of exec

```
...
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);

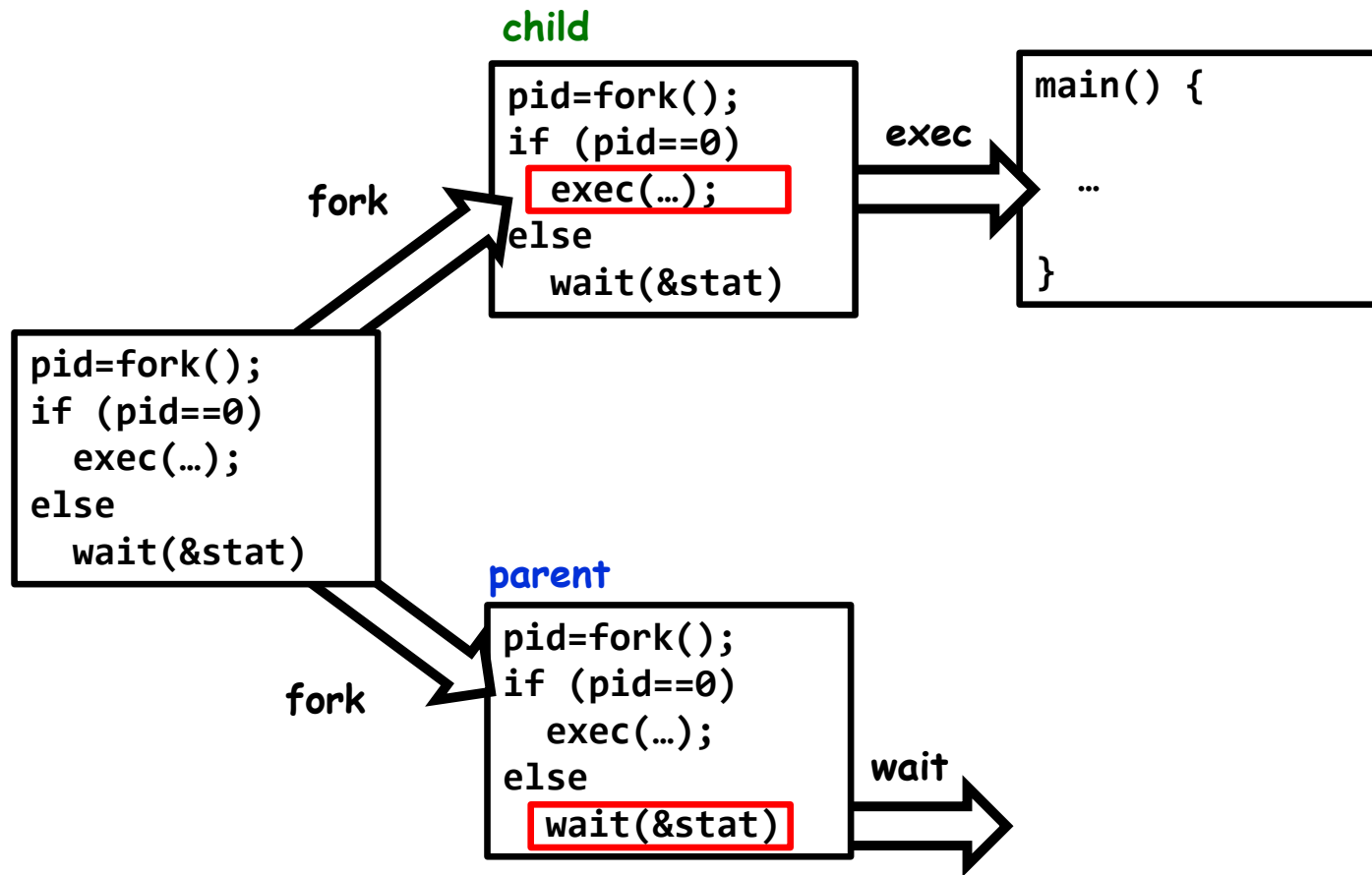
    /* execv doesn't return when it works.
       So, if we got here, it failed! */

    perror("execv");
    exit(1);
}
...
```

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);
    exit(42);
}
...
```

Process Management: The Shell pattern



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?

Conclusion

- Lock: An object only one thread can hold at a time
 - Provides mutual exclusion
 - Threads attempting to acquire lock forced to wait if other thread has lock
- Safe transfer from User \Rightarrow Kernel involves:
 - Atomic transfer to well defined kernel entry point (handler) while entering kernel mode
 - Separate kernel stack (don't trust user's stack pointer)
- 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
- Fork: State of original process duplicated in *both* Parent and Child!
 - Address Space (Memory), File Descriptors (covered later), etc...