Discussion Week 2

TA: Kyle Dewey

Overview

- Concurrency
  - Process level
  - Thread level
- MIPS - switch.s
- Project #1

Process Level

- UNIX/Linux: `fork()`
- Windows: `CreateProcess()`

`fork()/waitpid()` Example

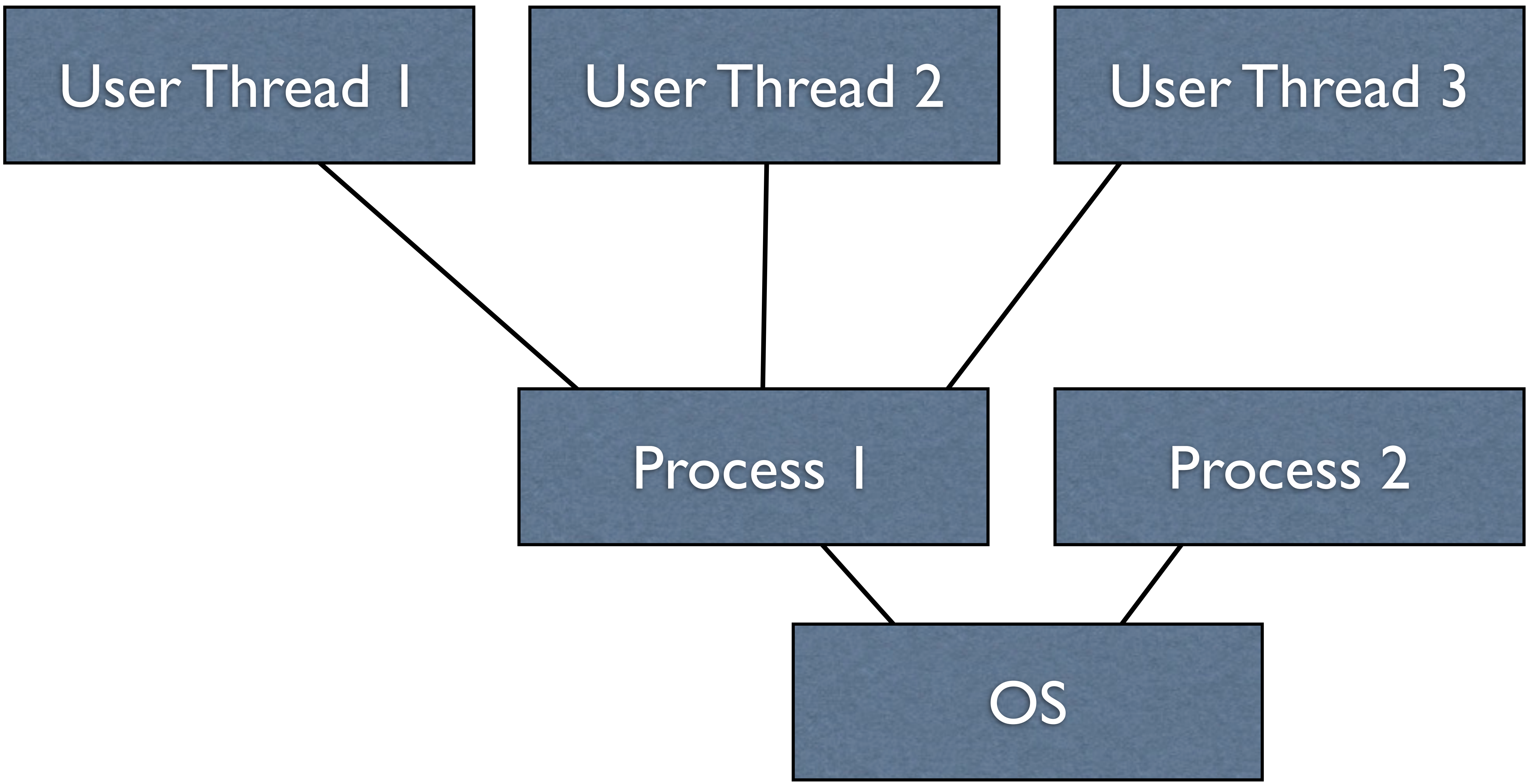
```
while( true )
 { fork(); }
```

Threading Overview

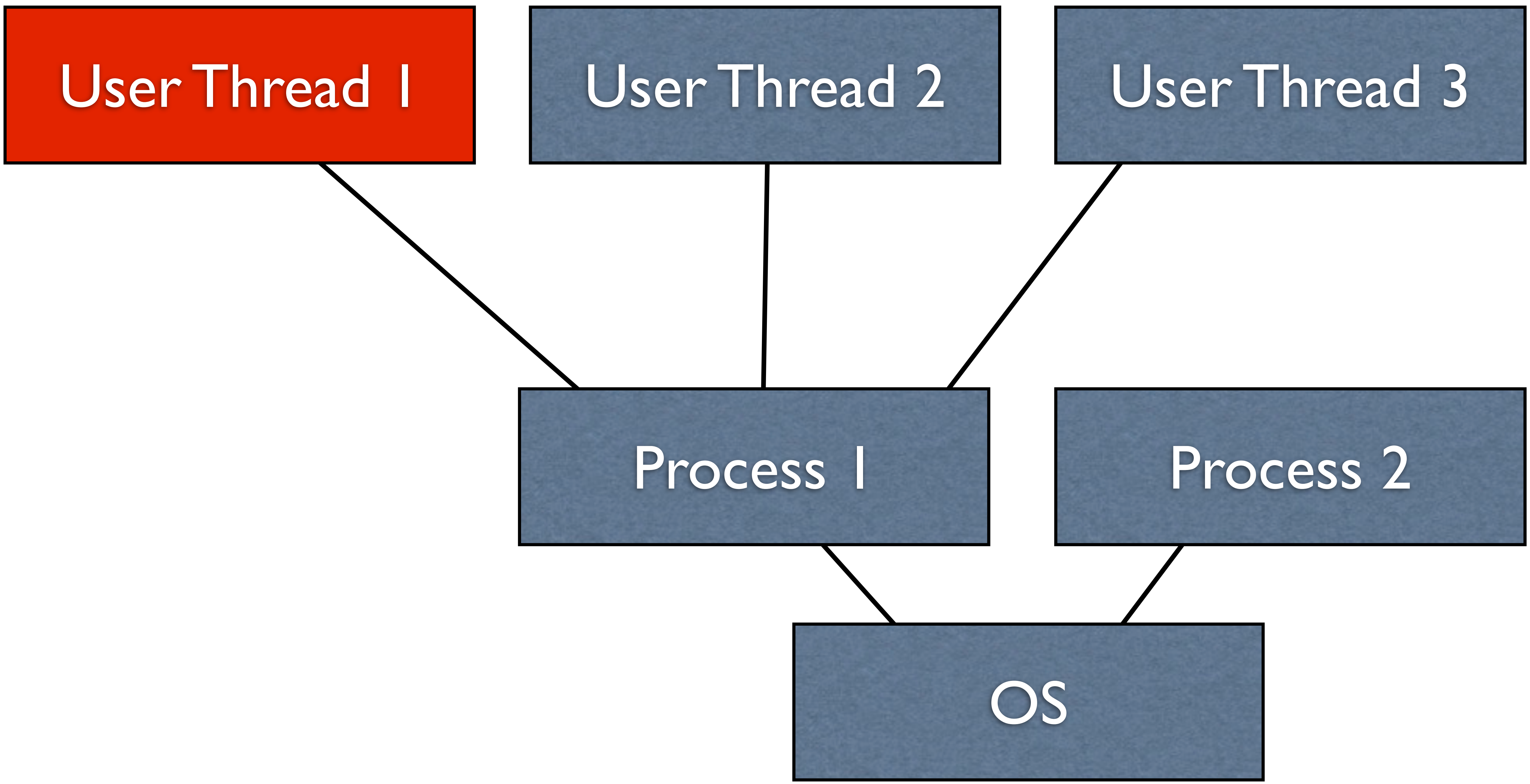
User-space Threads

- OS does not know about them
- Handle their own scheduling
- If one blocks, all block
- Cannot exploit SMP

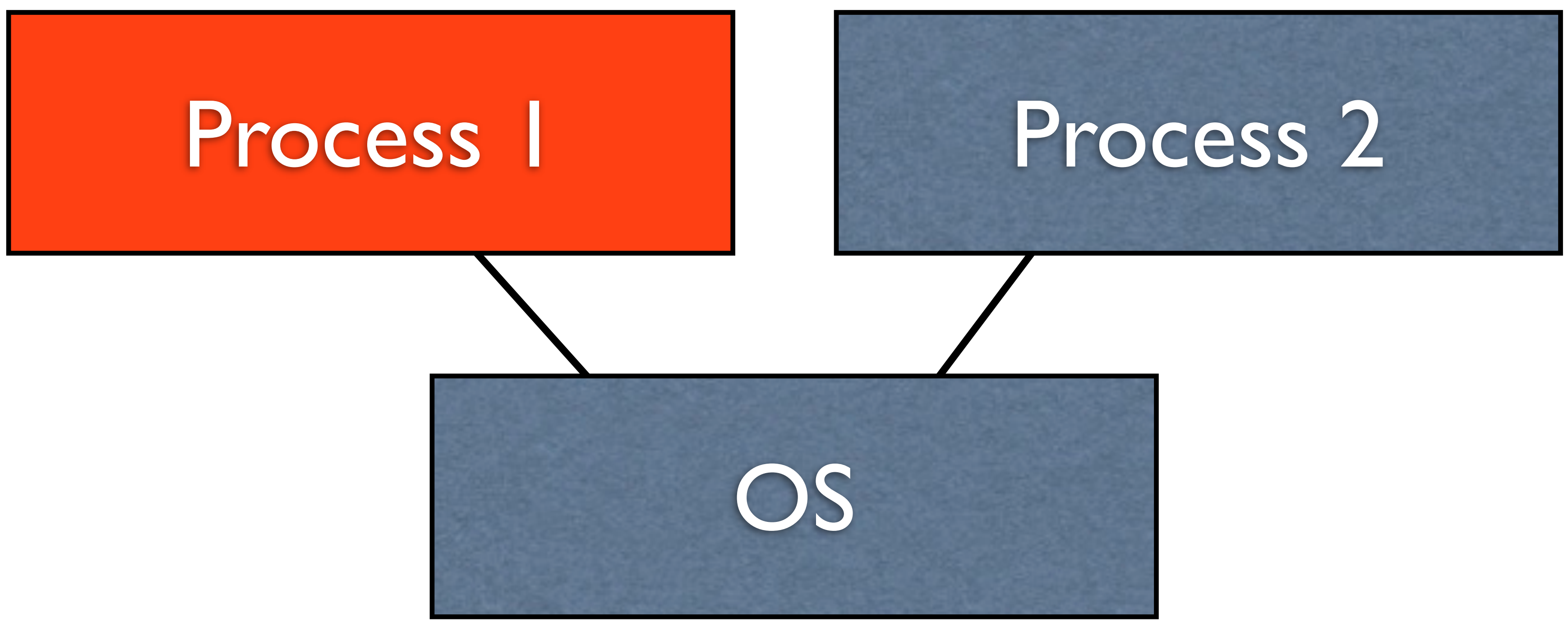
Blocking Example



Thread Standpoint



OS Standpoint

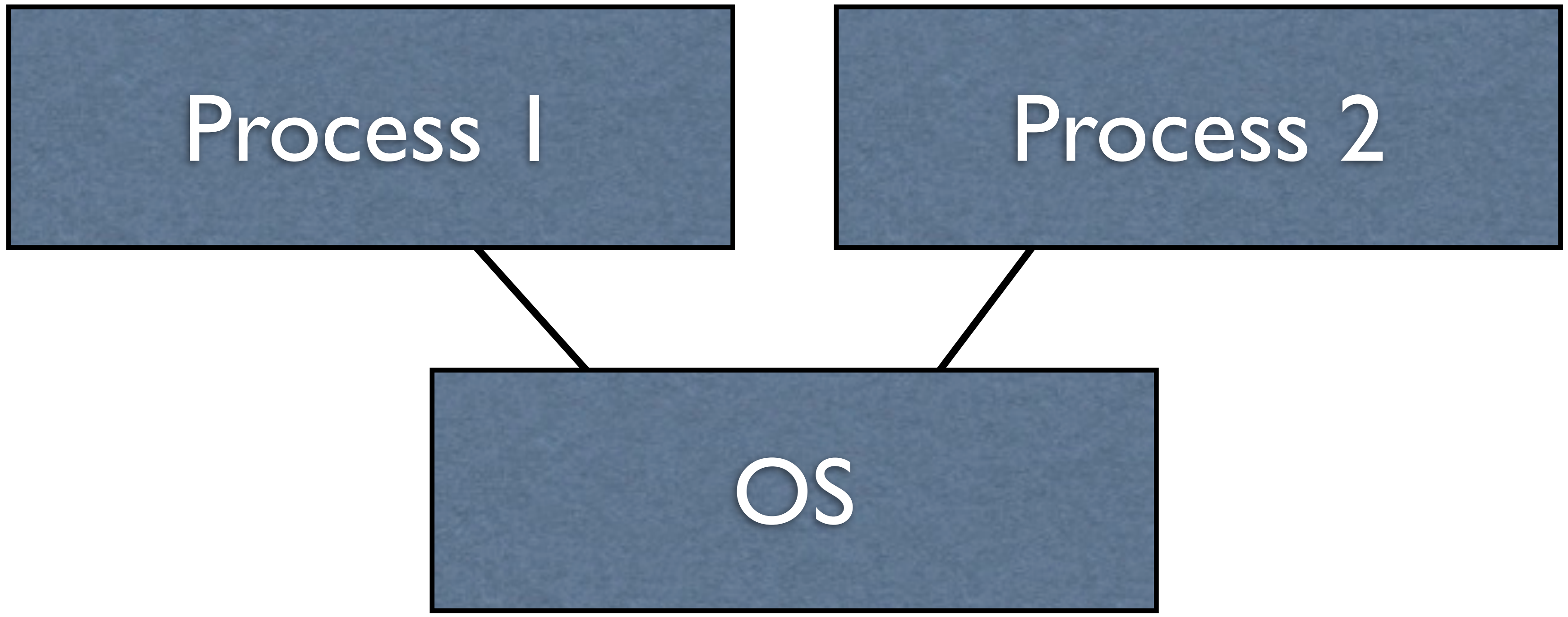


Blocking

- OS only sees a process
- OS blocks the process, in turn blocking all user-space threads

SMP

- Processes have only a single thread
- Without kernel assistance, this cannot be changed
- Only one thread means only one CPU



Kernel-Assisted

- OS has knowledge of threads
- OS schedules them
- Act like individual processes sharing an address space

General Pros/Cons

- Kernel threads can exploit SMP
- Kernel threads will not cause all threads to block
- User-space threads are lightweight
  - Context switch is cheap
  - Likely far less code

These are the concepts!

Then implementation happened...

Question: Do Pthreads threads run in user-space or are they kernel-assisted?

Answer: Yes.

Pthreads

- Really just a standard with a number of possible implementations
- Implementation can be kernel-assisted or in user-space
- Most OSes are kernel-assisted

Pthreads Example

Java Threads

- Again, merely a standard
- Most implement as kernel-assisted threads

Java Example

Kernel Thread Implementation

- OS can implement threads however it likes
- Pthreads and Java are libraries built on top of the threading primitives provided by the OS

Linux vs. Windows

- Linux provides the `clone()` system call
  - Threads are actually processes
- Windows provides `CreateThread()`
  - Referred to as "lightweight processes"

NACHOS Threads

- Kernel-assisted
- Cannot currently handle interrupts or preemption correctly
- Similar to MS-DOS...until project 2

MS-DOS/NACHOS

- One thread of execution
- One process can run
- OS is more like a large, complex software library

Thread Primitives

- `Fork()` - acts much like `pthread_create`
- `Yield()` - gives up the CPU for any other available threads
- `Sleep()` - like yield, but calling thread is blocked
- `Finish()` - terminates calling thread

For Project 1

- `Fork()` creates, but does not immediately start running, a new thread
- Though there is no I/O, `Sleep()` can still be called to block on waiting for a critical region to clear

NACHOS Threads

Concurrency

- Looks easy
- Really hard to get right
    - Really hard
    - No seriously, borderline impossible

Race Condition

- Different results are possible based on different process/thread orderings
- Ordering may be correct 99.999% of the time

Deadlock

- Two processes/threads wait for each other to do something
- While they wait, they do not do whatever it is they are waiting for
- Potential outcome of a race condition

Critical Region

- A point in code where the ordering matters
- Almost always this is some state that is shared between processes/threads

Client

```
connect to server:port1
connect to server:port2
do something with both
```

Server

```
accept from port1
accept from port2
do something with both
```

Fixing the Problem

- Do not share state
- Only share read-only state
- **Carefully** regulate write access to shared state

Regulation

- A critical region can be manipulated by only one thread at a time
- Need a way to enforce that at most one thread at any time point is in such a region

Solving in Java

- Java provides the `synchronized` keyword for blocks
- Only one thread at a time may access a block marked with the `synchronized` keyword

```
int x = 0;
public synchronized  void
set( int y ) {x = y;}
public int get() {return x;}
```

Who cares about Java?

- Many concurrency primitives work **exactly** like this, just with a little more work
- One call upon entrance to critical region, another upon exit
- The entrance and exit are implicit through blocks with Java

Semaphores

- Simply a shared integer
- One call decrements, another increments
- By convention, 0 is locked, and values > 0 are unlocked
  - Values < 0 mean the semaphore is not working!

Semaphores

- Increment/decrement are **atomic** - they are uninterruptible
- The highest possible number it can hold is equal to the max number of callers to the region it protects

Example

```
int x = 0;
Semaphore s;
public void set( int y ) {
 s.decrement(); // wait/P/down
 x = y;
 s.increment(); } // signal/V/up
public int get() {return x;}
```

Project 1 Task 1

- Experiment according to instructions
- Explain the execution of multithreaded code
- Add semaphores and contrast the difference

Project 1 Task 2

- Implement locks - essentially semaphores with a maximum of one caller at a time
- Given all the semaphore code to look at
- Hint hint it is a special case of a semaphore

Project 1 Task 3

- Implement conditions
- Require a correct `Lock` implementation
- Allows a group of threads to synchronize on a given section of code
  - Can enforce that all must be at the same point of execution
  - Block until this is true

Project 1 Task 4

- Identify and describe a race condition in a given section of code
- Fix the race condition using semaphores
- Fix it another way using locks and/or conditions