Kernel/userspace separation

- Userspace processes cannot interact directly with hardware (non-privileged mode)

- Attempting to execute a system call instruction causes a trap to the kernel (privileged mode), which handles the request

- Why is it necessary to have both privileged and non-privileged mode?

- How is privileged mode enforced, and how do virtual machine monitors work inside this model?
IO from userspace

- Userspace processes interact with disks and other devices via `open()`, `read()`, `write()`, and other system calls

- Multiple levels of abstraction: kernel presents file system to userspace, and device drivers present a (mostly) unified interface to kernel code

- What are the benefits and drawbacks of designing a system in this way?
Monolithic and microkernels

- Monolithic kernels encapsulate all aspects of functionality aside from hardware and user programs
  - **Pro:** Low communication cost, since everything is in the kernel’s address space
  - **Cons:** Millions of lines of code, continually expanding, no isolation between modules, security

- Microkernels separate functionality into separate modules that each expose an API
  - Services as servers
  - Why? How?
Processes versus threads

- Processes have multiple pieces of state associated with them
  - Program counter, registers, virtual memory, open file handles, mutexes, registered signal handlers, the text and data segment of the program, and so on
  - Total isolation, mediated by the kernel

- Threads are "lightweight" versions of processes
  - Which pieces of state listed above do threads not maintain individually?
Process creation

- **fork()**: create and initialize a new process control block
  - Copy resources of current process but assign a new address space
  - Calls to `fork()` return twice—once to parent (with pid of child process) and once to child
  - What makes this system call fast even for large processes? `vfork()` versus copy-on-write

- **exec()**: stop the current process and begin execution of a new one
  - Existing process image is overwritten
  - No new process is created
  - Is there a reason why `fork()` and `exec()` are separate system calls?
Threads

- How is a kernel thread different from a userspace thread?
  - Kernel thread: managed by OS, can run on a different CPU core than parent process
  - Userspace thread: managed by process/thread library, provides concurrency but no parallelism (can’t have two userspace threads within a process executing instructions at the same time)

- CPU sharing
  - Threads share CPU either implicitly (via preemption) or explicitly via calls to `yield()`
  - What happens when a userspace thread blocks on IO?
Synchronization

- Critical sections are sequences of instructions that may produce incorrect behavior if two threads interleave or execute them at the same time
- E.g. the banking example that everyone loves to use

- Mutexes are constructs that enforce mutual exclusion
  - `mutex.lock() / acquire()`: wait until no other thread holds the lock and then acquire it
  - `mutex.unlock() / release()`: release the Locken!
  - Mutexes rely on hardware support such as an atomic test-and-set instruction or being able to disable interrupts (why?)
Synchronization constructs

* Spinlocks are mutexes where `lock()` spins in a loop until the lock can be acquired
  * High CPU overhead, but no expensive context switches are necessary
  * In what type of scenario are spinlocks useful?

* Semaphores are counters that support atomic increments and decrements
  * `P(sem)` : block until semaphore count is positive, then decrement and continue
  * `V(sem)` : increment semaphore count
  * How are semaphores different from spinlocks?
Synchronization constructs

- Condition variables associated with mutexes allow threads to wait for events and to signal when they have occurred.
  - `cv.wait(mutex* m)`: release mutex `m` and block until the condition variable `cv` is signaled. `m` will be held when `wait()` returns.
  - `cv.signal()`: unblock one of the waiting threads. `m` must be held during the call but released sometime afterward.

- Why is it necessary to associate a mutex with a condition variable?

- What happens if `signal()` is invoked before a call to `wait()`?
Monitors

* Monitors are souped-up condition variables that support `enter()`, `exit()`, `wait()`, `signal()`, `broadcast()` routines.

* When one thread enters a monitor, no other thread can enter until the first thread exits.

* The exception is that a thread can wait on a condition after entering a monitor, permitting another thread to enter (which will potentially signal and unblock the first thread).
  * Hoare monitors: `signal()` causes a waiting thread to run immediately.
  * Mesa monitors: `signal()` returns to the caller and a waiting thread will unblock some time later.
**Deadlock**

Is this deadlock? How do we fix it?

<table>
<thead>
<tr>
<th>Thread 1:</th>
<th>Thread 2:</th>
<th>Thread 3:</th>
</tr>
</thead>
<tbody>
<tr>
<td>lock(A)</td>
<td>lock(B)</td>
<td>lock(C)</td>
</tr>
<tr>
<td>lock(B)</td>
<td>lock(C)</td>
<td>lock(A)</td>
</tr>
<tr>
<td>Do_thing1()</td>
<td>Do_thing2()</td>
<td>Do_thing3()</td>
</tr>
<tr>
<td>unlock(B)</td>
<td>unlock(C)</td>
<td>unlock(A)</td>
</tr>
<tr>
<td>unlock(A)</td>
<td>unlock(B)</td>
<td>unlock(C)</td>
</tr>
</tbody>
</table>
Deadlock

- What is an example of deadlock?

- Methods for preventing and avoiding deadlock
  - Have threads block until all required locks are available
  - Have all threads acquire locks in the same global ordering
  - Run banker’s algorithm to simulate what would happen if this thread and others made maximum requests: no deadlock = continue, deadlock = block and check again later

- Can resolve deadlock by breaking cycles in the dependency graph: choose a thread, kill it, and release its locks
  - What are the potential problems related to doing this?