Threads and all that

Jeff Chase
Threads

• A thread is a **stream of control**….  
  – Executes a sequence of instructions. 
  – Thread identity is defined by CPU register context (PC, SP, …, page table base registers, …) 
  – Generally: a thread’s **context** is its register values and referenced memory state (stacks, page tables).

• Multiple threads can execute independently: 
  – They can run in parallel on multiple cores...  
    • **physical concurrency** 
  – …or arbitrarily interleaved on some single core.  
    • **logical concurrency**

• A thread is also an OS abstraction to spawn and manage a stream of control.
Thread Abstraction

• Infinite number of processors
• Threads execute with variable speed
  – Programs must be designed to work with any schedule
## 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></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<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></td>
<td>y = y + x</td>
</tr>
<tr>
<td>z = x + 5y;</td>
<td>z = x + 5y;</td>
<td>thread is suspended</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>other thread(s) run</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>thread is resumed</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Possible Execution #1
- `x = x + 1;`
- `y = y + x;`
- `z = x + 5y;`

### Possible Execution #2
- `x = x + 1;`
- `y = y + x;`
- `z = x + 5y;` (thread is suspended)
- `other thread(s) run`
- `thread is resumed`

### Possible Execution #3
- `x = x + 1;`
- `y = y + x;`
- `z = x + 5y;`
- `thread is suspended`
- `other thread(s) run`
- `thread is resumed`
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  

Thread 1  
Thread 2  
Thread 3  

Thread 1  
Thread 2  
Thread 3  

 Thread 1  
Thread 2  
Thread 3  

Thread 1  

 Thread 1  
Thread 2  
Thread 3  

Thread 1  
Thread 2  
Thread 3  

 Thread 1  
Thread 2  
Thread 3  

 These executions are “schedules” chosen by the system.
# 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><strong>Heap</strong></td>
<td><strong>Thread Control Block (TCB)</strong></td>
<td><strong>Thread Control Block (TCB)</strong></td>
</tr>
<tr>
<td><strong>Global Variables</strong></td>
<td><strong>Stack Information</strong></td>
<td><strong>Stack Information</strong></td>
</tr>
<tr>
<td><strong>Code</strong></td>
<td><strong>Saved Registers</strong></td>
<td><strong>Saved Registers</strong></td>
</tr>
<tr>
<td></td>
<td><strong>Thread Metadata</strong></td>
<td><strong>Thread Metadata</strong></td>
</tr>
<tr>
<td></td>
<td><strong>Stack</strong></td>
<td><strong>Stack</strong></td>
</tr>
</tbody>
</table>
Running code can suspend the current thread just by saving its register values in memory. Load them back to resume it at any time.
Drawbridge

Rethinking the Library OS from the Top Down
Drawbridge thread ABI/API

The ABI supports multithreading through five calls to create, sleep, yield the scheduler quantum for, resume execution of, and terminate threads, as well as seven calls to create, signal, and block on synchronization objects:

DKHANDLE DkThreadCreate(Addr, Param, Flags);
DkThreadDelayExecution(Duration);
DkThreadYieldExecution();
DkThreadResume(ThreadHandle);
DkThreadExit();
DKHANDLE DkSemaphoreCreate(InitialCount, MaxCount);
DKHANDLE DkNotificationEventCreate(InitialState);
DKHANDLE DkSynchronizationEventCreate(InitialState);
DkSemaphoreRelease(SemaphoreHandle, ReleaseCount);
BOOL DkEventSet(EventHandler);
DkEventClear(EventHandler);
ULONG DkObjectsWaitAny(Count, HandleArray, Timeout);
Bascule thread ABI (refines Drawbridge)

SemaphoreCreate(InitCount, MaxCount) -> SemaphoreHandle
SemaphoreRelease(SemaphoreHandle, ReleaseCount)
SemaphorePeek(SemaphoreHandle) -> Count
NotificationEventCreate(InitialState) -> EventHandle
SynchronizationEventCreate(InitialState) -> EventHandle
EventSet(EventHandle)
EventClear(EventHandle)
EventPeek(EventHandle) -> State
ObjectReference(Handle)
ObjectClose(Handle)
ObjectsWaitAny(Count, Handles, Timeout) -> Index
ThreadCreate(Routine, Arg, Stack, Params) -> ThreadHandle
ThreadExit()
ThreadYieldExecution()
ThreadRaiseException(ThreadHandle, OpaquePointer)
Bascule/Drawbridge thread ABI

ThreadCreate(Routine, Arg, Stack, Params) -> ThreadHandle
ThreadExit()
ThreadYieldExecution()
ThreadRaiseException(ThreadHandle, OpaquePointer)
An Introduction to Programming with C# Threads

Andrew D. Birrell

This paper provides an introduction to writing concurrent programs with "threads". A threads facility allows you to write programs with multiple simultaneous points of execution, synchronizing through shared memory. The paper describes the basic thread and synchronization primitives, then for each primitive provides a tutorial on how to use

```csharp
Thread t = new Thread(new ThreadStart(foo.A));
t.Start();
foo.B();
t.Join();
```
Implementing threads

• Thread_fork(func, args)
  – Allocate thread control block
  – Allocate stack
  – Build stack frame for base of stack (stub)
  – Put func, args on stack
  – Put thread on ready list
  – Will run sometime later (maybe right away!)

• stub(func, args): Pintos switch_entry
  – Call (*func)(args)
  – Call thread_exit()
## Two threads call yield

<table>
<thead>
<tr>
<th>Thread 1’s instructions</th>
<th>Thread 2’s instructions</th>
<th>Processor’s instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>call thread_yield</td>
<td>call thread_yield</td>
<td>call thread_yield</td>
</tr>
<tr>
<td>save state to stack</td>
<td>save state to stack</td>
<td>save state to stack</td>
</tr>
<tr>
<td>save state to TCB</td>
<td>save state to TCB</td>
<td>save state to TCB</td>
</tr>
<tr>
<td>choose another thread</td>
<td>choose another thread</td>
<td>choose another thread</td>
</tr>
<tr>
<td>load other thread state</td>
<td>load other thread state</td>
<td>load other thread state</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>return thread_yield</td>
<td>return thread_yield</td>
<td>return thread_yield</td>
</tr>
<tr>
<td>call thread_yield</td>
<td>call thread_yield</td>
<td>call thread_yield</td>
</tr>
<tr>
<td>save state to stack</td>
<td>save state to stack</td>
<td>save state to stack</td>
</tr>
<tr>
<td>save state to TCB</td>
<td>save state to TCB</td>
<td>save state to TCB</td>
</tr>
<tr>
<td>choose another thread</td>
<td>choose another thread</td>
<td>choose another thread</td>
</tr>
<tr>
<td>load other thread state</td>
<td>load other thread state</td>
<td>load other thread state</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>return thread_yield</td>
<td>return thread_yield</td>
<td>return thread_yield</td>
</tr>
<tr>
<td>call thread_yield</td>
<td>call thread_yield</td>
<td>call thread_yield</td>
</tr>
<tr>
<td>save state to stack</td>
<td>save state to stack</td>
<td>save state to stack</td>
</tr>
<tr>
<td>save state to TCB</td>
<td>save state to TCB</td>
<td>save state to TCB</td>
</tr>
<tr>
<td>choose another thread</td>
<td>choose another thread</td>
<td>choose another thread</td>
</tr>
<tr>
<td>load other thread state</td>
<td>load other thread state</td>
<td>load other thread state</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>return thread_yield</td>
<td>return thread_yield</td>
<td>return thread_yield</td>
</tr>
<tr>
<td>call thread_yield</td>
<td>call thread_yield</td>
<td>call thread_yield</td>
</tr>
<tr>
<td>save state to stack</td>
<td>save state to stack</td>
<td>save state to stack</td>
</tr>
<tr>
<td>save state to TCB</td>
<td>save state to TCB</td>
<td>save state to TCB</td>
</tr>
<tr>
<td>choose another thread</td>
<td>choose another thread</td>
<td>choose another thread</td>
</tr>
<tr>
<td>load other thread state</td>
<td>load other thread state</td>
<td>load other thread state</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>return thread_yield</td>
<td>return thread_yield</td>
<td>return thread_yield</td>
</tr>
<tr>
<td>call thread_yield</td>
<td>call thread_yield</td>
<td>call thread_yield</td>
</tr>
<tr>
<td>save state to stack</td>
<td>save state to stack</td>
<td>save state to stack</td>
</tr>
<tr>
<td>save state to TCB</td>
<td>save state to TCB</td>
<td>save state to TCB</td>
</tr>
<tr>
<td>choose another thread</td>
<td>choose another thread</td>
<td>choose another thread</td>
</tr>
<tr>
<td>load other thread state</td>
<td>load other thread state</td>
<td>load other thread state</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 4.13: Interleaving of instructions when two threads loop and call thread_yield().
**Pthread (posix thread) example**

```c
volatile int counter = 0;
int loops;

void *worker(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        counter++;
    }
    pthread_exit(NULL);
}

int main(int argc, char *argv[]) {
    if (argc != 2) {
        fprintf(stderr, "usage: threads <loops>");
        exit(1);
    }
    loops = atoi(argv[1]);
    pthread_t p1, p2;
    printf("Initial value : %d\n", counter);
    pthread_create(&p1, NULL, worker, NULL);
    pthread_create(&p2, NULL, worker, NULL);
    pthread_join(p1, NULL);
    pthread_join(p2, NULL);
    printf("Final value   : %d\n", counter);
    return 0;
}
```

[pthread code from OSTEP]
Interleaving matters

In this schedule, $x$ is incremented only once: last writer wins. The program breaks under this schedule. This bug is a race.
pthread_mutex_t m;
volatile int counter = 0;
int loops;

void *worker(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&m);
        counter++;
        Pthread_mutex_unlock(&m);
    }
    pthread_exit(NULL);
}
C# lock (mutex)

class KV {
    string k, v;
    public void SetKV(string nk, string nv) {
        lock (this) {
            this.k = nk; this.v = nv;
        }
    }
    ...
}

An Introduction to Programming with C# Threads
Bascule/Drawbridge semaphore ABI

SemaphoreCreate(InitCount, MaxCount) -> SemaphoreHandle
SemaphoreRelease(SemaphoreHandle, ReleaseCount)
SemaphorePeek(SemaphoreHandle) -> Count

ObjectsWaitAny(Count, Handles, Timeout) -> Index
Semaphore

• A semaphore is a hidden atomic integer counter with only increment (V) and decrement (P) operations.
  – Also called “Up” and “Down” or “release” and “wait”.

• Decrement (P) blocks iff the count is zero.

• “Semaphores handle all of your synchronization needs with one elegant but confusing abstraction.”

```
V-Up

int sem

P-Down

if (sem == 0) then
  wait
  until a V
```
If a thread is in the **ready** state thread, then the system may choose to run it “at any time”. The kernel can switch threads whenever it gains control on a core, e.g., by a timer interrupt. If the current thread takes a fault or system call trap, and blocks or exits, then the scheduler switches to another thread. But it could also preempt a **running** thread. From the point of view of the program, dispatch and preemption are **nondeterministic**: we can’t know the **schedule** in advance.

These **preempt** and **dispatch** transitions are controlled by the kernel scheduler.

**Sleep** and **wakeup** transitions are initiated by calls to internal sleep/wakeup APIs by a running thread.
Thread Lifecycle

Thread Creation
- e.g., sthread_create()

Scheduler Resumes Thread

Thread Exit
- e.g., sthread_exit()

Ready
- Event Occurs
  - e.g., other thread calls sthread_join()

Running
- Thread Yields/
  Scheduler Suspends Thread
  - e.g., sthread_yield()

Waiting
- Thread Waits for Event
  - e.g., sthread_join()
What cores do

scheduler
getNextToRun()
nothing?
pause
idle

got thread

ready queue
(runqueue)

put thread
timer
quantum expired?
sleep? exit?

switch in
run thread

switch out

get thread

getId

Idle loop
A mutex is a binary semaphore

A mutex is just a binary semaphore with an initial value of 1, for which each thread calls **P-V** in strict pairs.

Once a thread **A** completes its **P**, no other thread can **P** until **A** does a matching **V**.
Bascule/Drawbridge event ABI

NotificationEventCreate(InitialState) -> EventHandle
SynchronizationEventCreate(InitialState) -> EventHandle
EventSet(EventHandle)
EventClear(EventHandle)
EventPeek(EventHandle) -> State

ObjectsWaitAny(Count, Handles, Timeout) -> Index
Events (MS/Windows)

- Multiple kinds of event objects: anything you can wait for.
- Event objects named by handles (safe references).
- All have two basic states: \textit{signaled} and \textit{not-signaled}.
- Unified *\texttt{WaitAny}* call for any/all kinds of event object.
  - Caller blocks iff all objects passed are in \textit{not-signaled} state.
  - Caller wakes up when any of them transitions to \textit{signaled} state.
- API: \texttt{set}, \texttt{clear} or \texttt{pulse} (set+clear)
- Synchronization events: wake up one waiter on signal.
- Notification events: wake up all waiters on signal.
Windows synchronization objects

They all enter a signaled state on some event, and revert to an unsignaled state after some reset condition. Threads block on an unsignaled object, and wakeup (resume) when it is signaled.
C# monitors

```csharp
public sealed class Monitor {
    public static bool Wait(Object obj) { ... }
    public static void Pulse(Object obj) { ... }
    public static void PulseAll(Object obj) { ... }
    ...
}
```

Pulse is also called signal or notify. PulseAll is also called broadcast or notifyAll.

A thread that calls “Wait” must already hold the object’s lock (otherwise, the call of “Wait” will throw an exception). The “Wait” operation atomically unlocks the object and blocks the thread*. A thread that is blocked in this way is said to be “waiting on the object”. The “Pulse” method does nothing unless there is at least one thread waiting on the object, in which case it awakens at least one such waiting thread (but possibly more than one). The “PulseAll” method is like “Pulse”, except that it awakens all the threads currently waiting on the object. When a thread is awoken inside “Wait” after blocking, it re-locks the object, then returns.

*An Introduction to Programming with with C# Threads*
For example, we could add the following “GetFromList” method to the class “KV”. This method waits until the linked list is non-empty, and then removes the top item from the list.

```csharp
public static KV GetFromList() {
    KV res;
    lock (typeof(KV)) {
        while (head == null) Monitor.Wait(typeof(KV));
        res = head; head = res.next;
        res.next = null; // for cleanliness
    }
    return res;
}
```

And the following revised code for the “AddToList” method could be used by a thread to add an object onto “head” and wake up a thread that was waiting for it.

```csharp
public void AddToList() {
    lock (typeof(KV)) {
        /* We’re assuming this.next == null */
        this.next = head; head = this;
        Monitor.Pulse(typeof(KV));
    }
}
```
Part 1. Sleeping late (80 points)

The "missed wakeup problem" occurs when a thread calls an internal sleep() primitive to block, and another thread calls wakeup() to awaken the sleeping thread in an unsafe fashion. For example, consider the following pseudocode snippets for two threads:

Sleeper thread

S1
Thread sleeper = self();
listMx.lock();
list.put(sleeper);
listMx.unlock();
S2
sleeper.sleep();

Waker thread

W1
listMx.lock();
Thread sleeper = list.get();
listMx.unlock();
sleeper.wakeup();
W2

One possible schedule is [S1, S2, W1, W2]. This is the intended behavior: the sleeper puts itself (a reference to its Thread object) on a list and sleeps, and the waker retrieves the sleeping thread from the list and then wakes that sleeper up.

These snippets could also execute in some schedule with W1 < S1 (W1 happens before S1) for the given sleeper. In this case, the waker does not retrieve the sleeper from the list, so it does not try to wake it up. It wakes up some other sleeping thread, or the list is empty, or whatever.

The schedule of interest is [S1, W1, W2, S2]. In this case, the sleeper is on the list, and the waker retrieves that sleeper from the list and issues a wakeup call on that sleeper, as in the first schedule. But the sleeper is not asleep, and so the wakeup call may be lost or it may execute incorrectly. This is the missed wakeup problem.

(a) What could go wrong? Outline how this code is vulnerable to the missed wakeup problem, and illustrate with an example schedule.

Note that these raw sleep/wakeup primitives, as defined, are inherently unsafe and vulnerable to the missed wakeup problem. That is why we have discussed them only as "internal" primitives to illustrate blocking behavior: we have not studied them as part of any useful concurrency API. The point of the question is that monitors and semaphores are designed to wrap sleep/wakeup in safe higher-level abstractions that allow threads to sleep for events and wake other threads when those events occur. Both abstractions address the missed wakeup problem, but they resolve the problem in different ways.
What could go wrong?

Consider schedule [S1, W1, W2, S2]. In this case, the sleeper is on the list, and the waker retrieves that sleeper from the list and issues a wakeup call on that sleeper. But the sleeper is not asleep, and so the wakeup call may be lost or it may execute incorrectly. This is the missed wakeup problem. Condition variables are designed to solve it.
(b) How does blocking with monitors (condition variables) avoid the missed wakeup problem? Illustrate how the code snippets in (a) might be implemented using monitors, and outline why it works.

Monitors (condition variables) provide a higher-level abstraction: instead of using raw sleep and wakeup, we use wait() and signal/notify(). These primitives serve the desired purpose, but the wait() primitive is integrated with the locking, so that the sleeper may hold the mutex until the sleep is complete. The implementation of wait() takes care of releasing the mutex atomically with the sleep. For example:

```
listMx.lock();
sleeper++;
listCv.wait();
sleeper--;
listMx.unlock();
listMx.lock();
if (sleeper > 0)
    listCv.signal();
listMx.unlock();
```

In these snippets we presume that the condition variable listCv is bound to the mutex listMx. Various languages show this with various syntax. I didn't require it for full credit.

In this example, the sleeper's snippet may execute before or after the waker, but it is not possible for the waker to see a sleeper's count (sleeper > 0) and then fail to wake a/the sleeper up. The missed wakeup problem cannot occur.
(d) Next implement sleep() and wakeup() primitives using semaphores. These primitives are used as in the code snippets in part (1a) above. Note that sleep() and wakeup() operate on a specific thread. Your implementation should be "safe" in that it is not vulnerable to the missed wakeup problem.

The idea here is to allocate a semaphore for each thread. Initialize it to 0. The thread sleeps with a P() on its semaphore. Another thread can wake a sleeping thread T up with a V() on T’s semaphore. Thus each call to sleep() consumes a wakeup() before T can run again. If a wakeup on T is scheduled before the corresponding sleep, then the wakeup is “remembered” and T’s next call to sleep simply returns. Note, however, that with this implementation a wakeup is remembered even if the sleep occurs far in the future, and the semaphore records any number of wakeups. Thus it is suitable only if the use of sleep/wakeup is restricted so that a wakeup is issued only after T has declared its intention to sleep, as in the example snippets.

```c
for each thread: thread.s.init(0);
thread.sleep: thread.s.P();
thread.wakeup: thread.s.V();
```

Note that the solution of giving each thread its own semaphore is generally a useful trick: for example, it is the key to the difficult problem of implementing condition variables using semaphores, as discussed at length in the 2003 paper by Andrew Birrell discussing that problem.
Semaphore

void P() {
    s = s - 1;
}

void V() {
    s = s + 1;
}

Step 0.
Increment and decrement operations on a counter.

But how to ensure that these operations are atomic, with mutual exclusion and no races?

How to implement the blocking (sleep/wakeup) behavior of semaphores?
Semaphore

void P() {
    synchronized(this) {
        ....
        s = s - 1;
    }
}

void V() {
    synchronized(this) {
        s = s + 1;
        ....
    }
}

Step 1.
Use a mutex so that increment (V) and decrement (P) operations on the counter are atomic.
Semaphore

synchronized void P() {
    s = s - 1;
}

synchronized void V() {
    s = s + 1;
}

Step 1.
Use a mutex so that increment (V) and decrement (P) operations on the counter are atomic.
Semaphore

synchronized void \textbf{P}() {
    \textbf{while} (s == 0) \textbf{wait}();
    s = s - 1;
}

synchronized void \textbf{V}() {
    s = s + 1;
    \textbf{if} (s == 1) \textbf{notify}();
}

\textbf{Step 2.}
Use a condition variable to add sleep/wakeup synchronization around a zero count.

(This is Java syntax.)
Semaphore

synchronized void $P()$ {
    while ($s == 0$)
        wait();
    $s = s - 1$;
    ASSERT($s >= 0$);
}

synchronized void $V()$ {
    $s = s + 1$;
    signal();
}

This code constitutes a proof that monitors (mutexes and condition variables) are at least as powerful as semaphores.

Loop before you leap!
Understand why the while is needed, and why an if is not good enough.

Wait releases the monitor/mutex and blocks until a signal.

Signal wakes up one waiter blocked in $P$, if there is one, else the signal has no effect: it is forgotten.
The primary I/O mechanism in Drawbridge is an I/O stream. I/O streams are byte streams that may be memory-mapped or sequentially accessed.

Streams are named by URIs...Supported URI schemes include file:, pipe:, http:, https:, tcp:, udp:, pipe.srv:, http.srv, tcp.srv:, and udp.srv:. The latter four schemes are used to open inbound I/O streams for server applications:

```c
DKHANDLE DkStreamOpen(URI, AccessMode, ShareFlags, CreateFlags, Options);
ULONG DkStreamRead(StreamHandle, Offset, Size, Buffer);
ULONG DkStreamWrite(StreamHandle, Offset, Size, Buffer);
DkStreamMap(StreamHandle, Addr, ProtFlags, Offset, Size);
DkStreamUnmap(Addr);
DkStreamSetLength(StreamHandle, Length);
DkStreamFlush(StreamHandle);
DkStreamDelete(StreamHandle);
DkStreamWaitForClient(StreamHandle);
DkStreamGetName(StreamHandle, Flags, Buffer, Size);
DkStreamAttributesQuery(URI, DK_STREAM_ATTRIBUTES *Attr);
DkStreamAttributesQueryByHandle(StreamHandle,
   DK_STREAM_ATTRIBUTES *Attr);
```
File abstraction

Program A

Library

open “/a/b”
write (“abc”)
read

Program B

Library

open “/a/b”
read
write (“def”)

system call trap/return

OS kernel
Unix Pipes

Example: cat | cat

cat pseudocode (user mode)
while (until EOF) {
  read(0, buf, count);
  compute/transform data in buf;
  write(1, buf, count);
}

Kernel pseudocode for pipes: Producer/consumer bounded buffer

Pipe write: copy in bytes from user buffer to in-kernel pipe buffer, blocking if k-buffer is full.

Pipe read: copy bytes from pipe’s k-buffer out to u-buffer. Block while k-buffer is empty, or return EOF if empty and pipe has no writer.
Pipes

Kernel-space pseudocode
System call internals to read/write N bytes for buffer size B.

```c
read(buf, N)
{
    for (i = 0; i++; i<N) {
        move one byte into buf[i];
    }
}
```
Pipes

```c
read(buf, N) {
    pipeMx.lock();
    for (i = 0; i++ < N) {
        while (no bytes in pipe)
            dataCv.wait();
        move one byte from pipe into buf[i];
        spaceCV.signal();
    }
    pipeMx.unlock();
}
```

Read N bytes from the pipe into the user buffer named by `buf`. Think of this code as deep inside the implementation of the `read` system call on a pipe. The `write` implementation is similar.
Pipes

\[
\text{read}(\text{buf, N}) \\
\begin{aligned}
&\{ \\
&\quad \text{readerMx.lock();} \\
&\quad \text{pipeMx.lock();} \\
&\quad \text{for (i = 0; i++; i<N) \{} \\
&\quad\quad \text{while (no bytes in pipe)} \\
&\quad\quad \quad \text{dataCv.wait();} \\
&\quad\quad \quad \text{move one byte from pipe into buf[i];} \\
&\quad\quad \text{spaceCV.signal();} \\
&\quad \}\} \\
&\quad \text{pipeMx.unlock();} \\
&\quad \text{readerMx.unlock();} \\
&\\}
\]

In Unix, the \textbf{read/write} system calls are “atomic” in the following sense: no \textbf{read} sees interleaved data from multiple \textbf{writes}. The extra lock here ensures that all read operations occur in a serial order, even if any given operation blocks/waits while in progress.
Why exactly does Pipe (bounded buffer) require a nested lock?

First: remember that this is the exception that proves the rule. Nested locks are generally not necessary, although they may be useful for performance. Correctness first: always start with a single lock.

Second: the nested lock is not necessary even for Pipe if there is at most one reader and at most one writer, as would be the case for your typical garden-variety Unix pipe.

The issue is what happens if there are multiple readers and/or multiple writers. The nested lock is needed to meet a requirement that read/write calls are atomic. Understanding this requirement is half the battle.

Consider an example. Suppose three different writers \{A, B, C\} write 10 bytes each, each with a single write operation, and a reader reads 30 bytes with a single read operation. The read returns the 30 bytes, so the read will “see” data from multiple writes. That’s OK. The atomicity requirement is that the reader does not observe bytes from different writes that are interleaved (mixed together).

A necessary condition for atomicity is that the writes are serialized: the system chooses some order for the writes by A, B, and C, even if they request their writes "at the same time". The data returned by the read reflects this ordering. Under no circumstances does a read see an interleaving, e.g.: 5 bytes from A, then 5 bytes from B, then 5 more bytes from A,… (Note: if you think about it, you can see that a correct implementation must also serialize the reads.)

This atomicity requirement exists because applications may depend on it: e.g., if the writers are writing records to the pipe, then a violation of atomicity would cause a record to be “split”.

This is particularly important when the size of a read or write (N) exceeds the size of the bounded buffer (B), i.e., N>B. A read or write with N>B is legal. But such an operation can’t be satisfied with a single buffer’s worth of data, so it can’t be satisfied without alternating execution of a reader and a writer (“ping-pong style”). On a single core, the reader or writer is always forced to block at least once to wait for its counterparty to place more bytes in the buffer (if the operation is a read) or to drain more bytes out of the buffer (if the operation is a write). In this case, it is crucial to block any other readers or writers from starting a competing operation. Otherwise, atomicity is violated and at least one of the readers will observe an interleaving of data.

The nested lock ensures that at most one reader and at most one writer are moving data in the “inner loop” at any given time.
Spinlock: a first try

```c
int s = 0;

lock() {
    while (s == 1) {
    }
    ASSERT (s == 0);
    s = 1;
}

unlock () {
    ASSERT(s == 1);
    s = 0;
}
```

Spinlocks provide mutual exclusion among cores without blocking.

Global spinlock variable

Busy-wait until lock is free.

Spinlocks are useful for lightly contended critical sections where there is no risk that a thread is preempted while it is holding the lock, i.e., in the lowest levels of the kernel.
Spinlock: what went wrong

```c
int s = 0;

lock() {
    while (s == 1) {
    }
    s = 1;
}

unlock();
    s = 0;
}
```

Race to acquire.
Two (or more) cores see $s == 0$. 
We need an atomic “toehold”

• To implement safe mutual exclusion, we need support for some sort of “magic toehold” for synchronization.
  – The lock primitives themselves have critical sections to test and/or set the lock flags.

• Safe mutual exclusion on multicore systems requires some hardware support: atomic instructions
  – These instructions perform an atomic read-modify-write of a memory location. We use them to implement locks.
  – If we have any of those, we can build higher-level synchronization objects like monitors or semaphores.
  – Note: we also must be careful of interrupt handlers….
  – They are expensive, but necessary.
<table>
<thead>
<tr>
<th>Spin_Lock:</th>
<th>Get_Lock:</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMP lockvar, 0 ;Check if lock is free</td>
<td></td>
</tr>
<tr>
<td>JE Get_Lock</td>
<td></td>
</tr>
<tr>
<td>PAUSE ; Short delay</td>
<td></td>
</tr>
<tr>
<td>JMP Spin_Lock</td>
<td></td>
</tr>
<tr>
<td>MOV EAX, 1</td>
<td></td>
</tr>
<tr>
<td>XCHG EAX, lockvar ; Try to get lock</td>
<td></td>
</tr>
<tr>
<td>CMP EAX, 0 ; Test if successful</td>
<td></td>
</tr>
<tr>
<td>JNE Spin_Lock</td>
<td></td>
</tr>
</tbody>
</table>

**Idle the core for a contended lock.**

**Atomic exchange to ensure safe acquire of an uncontended lock.**

**XCHG** is a variant of **compare-and-swap**: compare \( x \) to value in memory location \( y \); if \( x \neq *y \) then exchange \( x \) and \( *y \). Determine success/failure from subsequent value of \( x \).
Locking and blocking

If thread $T$ attempts to acquire a lock that is busy (held), $T$ must spin and/or block (sleep) until the lock is free. By sleeping, $T$ frees up the core for some other use. Just sitting and spinning is wasteful!

*Note:* $H$ is the lock holder when $T$ attempts to acquire the lock.
Threads in a Process

• Threads are useful at user-level
  – Parallelism, hide I/O latency, interactivity
• Option A (early Java): user-level library, within a single-threaded process
  – Library does thread context switch
  – Kernel time slices between processes, e.g., on system call I/O
• Option B (Linux, MacOS, Windows): use kernel threads
  – System calls for thread fork, join, exit (and lock, unlock,...)
  – Kernel does context switching
  – Simple, but a lot of transitions between user and kernel mode
• Option C (Windows): scheduler activations
  – Kernel allocates processors to user-level library
  – Thread library implements context switch
  – System call I/O that blocks triggers upcall
• Option D: Asynchronous I/O