Recap – Week 3

Pamela Delgado

March 6, 2019

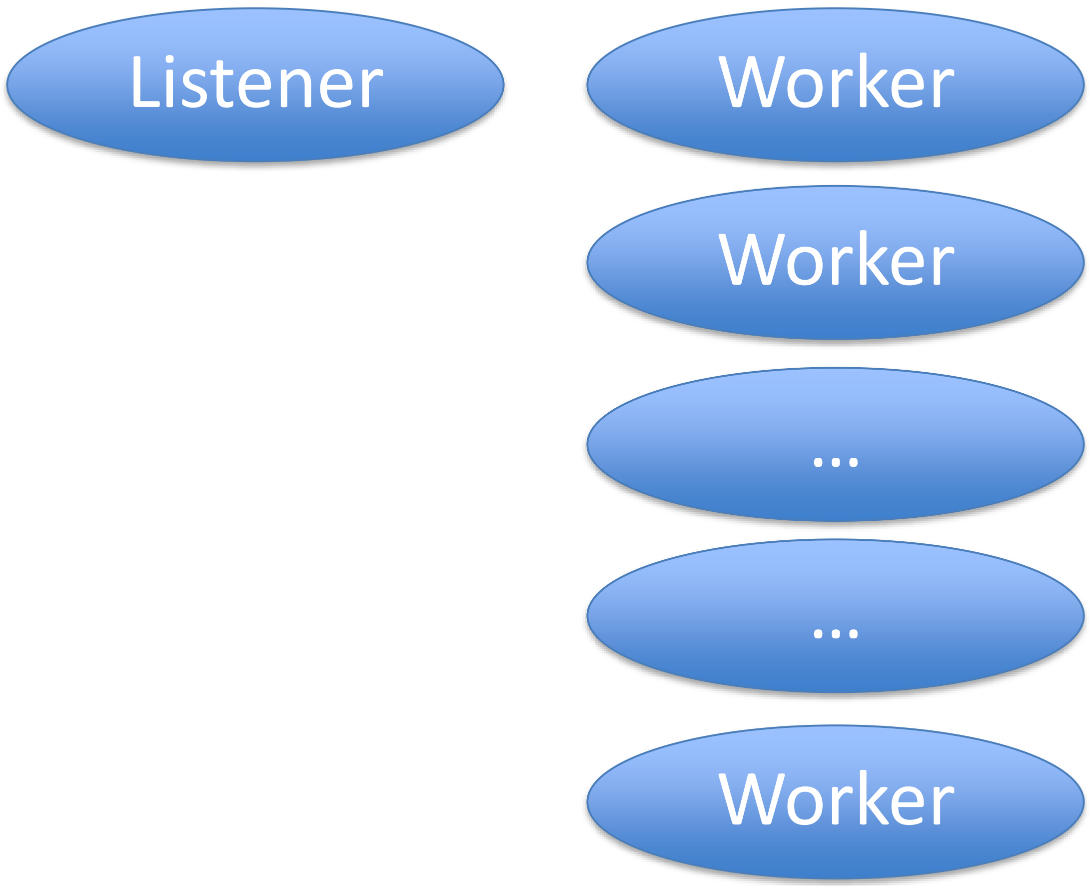
(slides Willy Zwaenepoel)

Application Multiprocess Structuring

- One application = multiple processes
- Example: web server
- Goal: overlap computation with I/O

Application Multiprocess Structuring

Multiprocess Web Server



Interprocess Communication

- Always by value
- No addresses / pointers

Interprocess Communication

- Message passing
- Remote procedure call
  - Client and server stubs

Week 4
Application Multithreading and Synchronization (continued)

Pamela Delgado

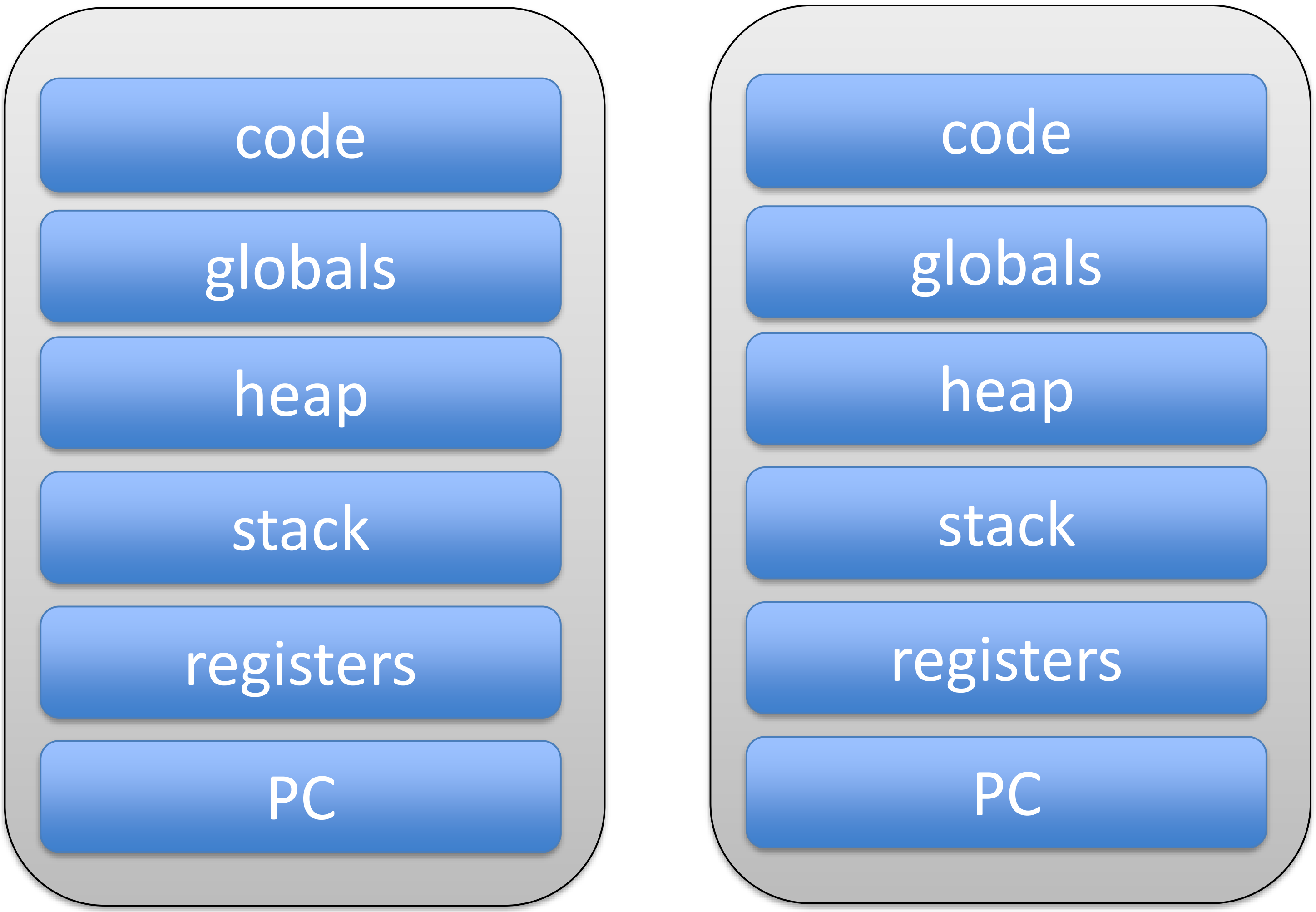
March 13, 2019

(slides Willy Zwaenepoel)

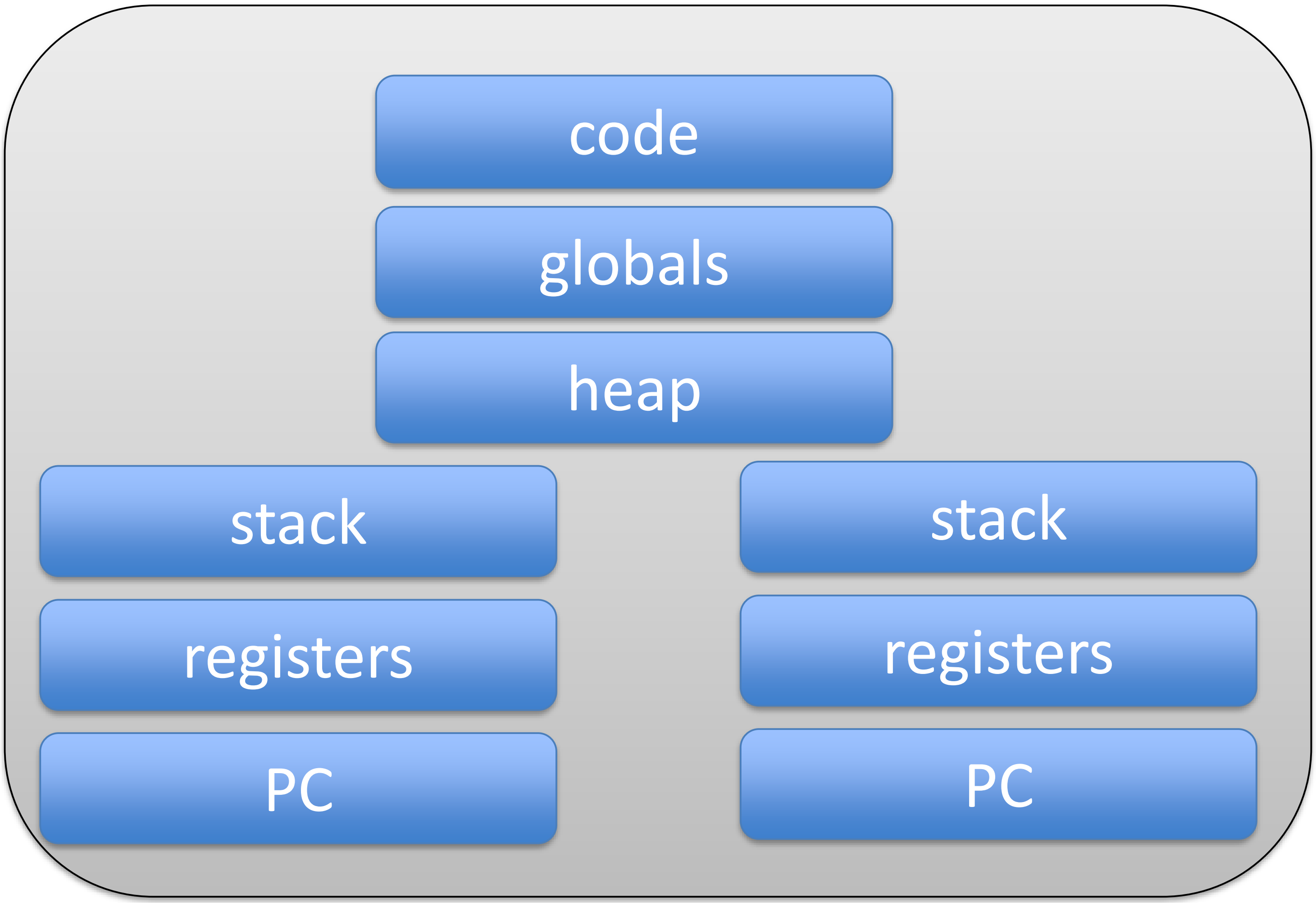
Key Concepts

- Multithreading vs. multiprocessing
- Synchronization
- Pthreads examples

Two Processes



Two Threads in a Process



In General

- Processes provide separation
  - In particular, memory separation (no shared data)
  - Suitable for coarse-grain interaction
- Threads do not
  - In particular, share memory (shared data)
  - Suitable for tighter integration

Most Important Difference

- Process crashes
  - Other processes are not affected
- Thread crashes
  - The entire process, including other threads, crashes

Concrete Example: Web Server

- Serving static content (files)
  - Probably no bugs
  - Can easily be done in a multithreaded process
- Serving dynamic (third-party) content
  - No guarantees about bugs
  - Keep in a different process

Shared Data

- Advantage:
  - Many threads can read/write it
- Disadvantage:
  - Many threads can read/write it
  - Can lead to *data races*

Data Race

- Unexpected/unwanted access to shared data
- Result of *inter-leaving* of thread executions
- Program must be correct for all inter-leavings

Basic Approach to Multithreading

- Divide “work” among multiple threads
- Which data is shared?
  - Globals and heap
  - Not locals
  - Not read-only
- Where is shared data accessed?
- Put shared data access in critical section
  - Only one process at a time can access it

Why this (mostly) works

- Trouble with multithreaded execution:
  - Data races
  - Data changed by another thread
- Critical section:
  - No other thread can change data
- So you are (mostly) ok

Pthreads: Thread Creation and Destruction

- Pthread_create(&threadid, threadcode, arg)
- Pthread_exit(status)
- Pthread_join(threadid, &status)

Pthreads: Thread Creation and Destruction

- Pthread_create(&threadid, threadcode, arg)
  - Create thread
  - Return threadid
  - Run threadcode
  - With argument arg

```
#include <pthread.h>
int
pthread_create(      pthread_t *          thread,
               const pthread_attr_t *     attr,
                     void *               (*start_routine)(void*),
                     void *               arg);
```

- Pthread_exit(status)
- Pthread_join(threadid, &status)

Pthreads: Thread Creation and Destruction

- Pthread_create(&threadid, threadcode, arg)
- Pthread_exit(status)
  - Terminate thread
  - Optionally return status
- Pthread_join(threadid, &status)

Pthreads: Thread Creation and Destruction

- Pthread_create(&threadid, threadcode, arg)
- Pthread_exit(status)
- Pthread_join(threadid, &status)
  - Wait for thread threadid to exit
  - Receive status, if any

```
int pthread_join(pthread_t thread, void **value_ptr);
```

Pthreads: Locks

- Pthread_mutex_lock(mutex)
- Pthread_mutex_unlock(mutex)

Pthreads: Locks

- Pthread_mutex_lock(mutex)
  - If mutex is held, block
  - If mutex is not held
    - Acquire mutex
    - Proceed
- Pthread_mutex_unlock(mutex)

Pthreads: Locks

- Pthread_mutex_lock(mutex)
- Pthread_mutex_unlock(mutex)
  - Release mutex

Example: Single-Threaded Code

```
main() {
    int i
    int sum = 0, prod = 1
    for( i=0; i<MAX; i++ ) {
        c = a[i] * b[i]
        sum += c
        prod *= c
    }
}
```

Basic Approach to Multithreading

- Divide “work” among multiple threads
- Which data is shared?
  - Globals and heap
  - Not locals
  - Not read-only
- Where is shared data accessed?
- Define one mutex
- Put lock/unlock around each shared access

Example: Divide Work

- Give each thread equal number of iterations

Example: Divide Work

```
main() {
    int i
         int sum= 0, prod = 1
    for( i=0; i<MAX_THREADS; i++ ) { Pthread_create(…) }
    for( i=0; i<MAX_THREADS; i++ ) { Pthread_join(…) }
    printf( sum )
    printf( prod )
}

Threadcode() {
    int i, c
    for( i=my_min; i<my_max; i++ ) {
        c = a[i] * b[i]
        sum += c
        prod *= c
    }
}
```

Example: Shared Data

- Shared data
  - sum
  - prod
- Shared read-only data
  - a[], b[] read only
- Local data
  - i (loop index), c
- mutex on access to sum and prod

Example: Synchronization

```
Threadcode() {
    int i
    for( i=my_min; i<my_max; i++ ) {
        c = a[i] * b[i]
        Pthread_mutex_lock( biglock )
        sum += c
        prod *= c
        Pthread_mutex_unlock( biglock )
    }
}
```

A Common Mistake/Misunderstanding: A Single Line of Code is not Atomic

- a = a + 1
- Is in reality
  - Load a from memory into register
  - Increment register
  - Store register value in memory
- Instruction sequence may be interleaved
- Some machines have atomic increments

Back to Where We Were

```
Threadcode() {
    int i
    for( i=my_min; i<my_max; i++ ) {
        c = a[i] * b[i]
        Pthread_mutex_lock( biglock )
        sum += c
        prod *= c
        Pthread_mutex_unlock( biglock )
    }
}
```

Why it will not work very well

- Single lock inhibits parallelism
- Two approaches:
  - Fine-grain locking:
    - Multiple locks on individual pieces of shared data
  - Privatization:
    - Make shared data accesses into private data accesses

Fine Grain Locking

- Define separate lock for sum and prod

Example: Finer-Grain Locking

```
Threadcode() {
    int i, c
    for( i=my_min; i<my_max; i++ ) {
        c = a[i] * b[i]
        Pthread_mutex_lock(sumlock)
        sum += c
        Pthread_mutex_unlock(sumlock)
        Pthread_mutex_lock(prodlock)
        prod *= c
        Pthread_mutex_unlock(prodlock)
    }
}
```

Example: Privatization

- Define for each thread
  - A local variable representing its sum
  - A local variable representing its product
- Use those for accesses in the loop
  - Become local accesses
  - No need for lock
- Only access shared data after the loop
  - Use lock there

Example: Privatization

```
Threadcode() {
    int i, c
    local_sum = 0
    local_prod = 1

    for( i=my_min; i<my_max; i++ ) {
        c = a[i] * b[i]
        local_sum += c
        local_prod *= c
    }

    Pthread_mutex_lock(sumlock)
    sum += local_sum
    Pthread_mutex_unlock(sumlock)
    Pthread_mutex_lock(prodlock)
    prod *= local_prod
    Pthread_mutex_unlock(prodlock)
}
```

Example: Privatization

- Only one access to each lock per thread
- Compare to before mymax-mymin accesses

Another Example: Multithreaded Web Server

```
ListenerThread {
    forever {
        Receive( request )
        Pthread_create(…)
     }
}

WorkerThread( request ) {
    read file from disk
    Send( response )
    Pthread_exit()
}
```

Shared Data?

- There is none!
- Process creation serves as synchronization

Multithreaded Web Server with Thread Pool

```
ListenerThread {
    for( i=0; i<MAX_THREADS; i++ ) { Pthread_create(…) }
    forever {
        Receive( request )
        hand request to thread[?]
    }
}

WorkerThread[?] {
    forever {
        wait for available request
        read file from disk
        Send( reply )
    }
}
```

Shared Data?

- We need to create shared data
- Going to be some kind of a queue
- Put lock/unlock around it

Multithreaded Web Server with Thread Pool

```
ListenerThread {
    for( i=0; i<MAX_THREADS; i++ ) thread[i] = Pthread_create(…)
    forever {
        Receive( request )
        Pthread_mutex_lock( queuelock )
        put request in queue
        Pthread_mutex_unlock( queuelock )
    }
}

WorkerThread {
    forever {
        Pthread_mutex_lock( queuelock )
        take request out of queue
        Pthread_mutex_unlock( queuelock )
        read file from disk
        Send( reply )
    }
}
```

It will not work

- Not fork-join parallelism
- You need to tell worker(s) there is something for them to do (i.e., in the queue)
- Sometimes called task parallelism

Pthreads: Condition Variables

- Pthread_cond_wait(cond, mutex)
- Pthread_cond_signal(cond, mutex*)
- Pthread_cond_broadcast(cond, mutex)

```
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
int pthread_cond_signal(pthread_cond_t *cond);
```

* Not strictly correct, but easier to explain

Pthreads: Condition Variables

- Pthread_cond_wait(cond, mutex)
- Pthread_cond_signal(cond, mutex)
- Pthread_cond_broadcast(cond, mutex)

- Must hold mutex when calling any of these!

Pthreads: Condition Variables

- Pthread_cond_wait(cond, mutex)
  - Wait for a signal on cond
  - Release mutex
- Pthread_cond_signal(cond, mutex)
- Pthread_cond_broadcast(cond, mutex)

- Must hold mutex when calling any of these!

Pthreads: Condition Variables

- Pthread_cond_wait(cond, mutex)
- Pthread_cond_signal(cond, mutex)
  - Signal one thread waiting on cond
  - Signaled thread re-acquires mutex
    - At some later time, not necessarily immediately
  - If no thread waiting, no-op
- Pthread_cond_broadcast(cond, mutex)

Pthreads: Condition Variables

- Pthread_cond_wait(cond, mutex)
- Pthread_cond_signal(cond, mutex)
- Pthread_cond_broadcast(cond, mutex)
  - Signal all threads waiting on cond
  - If no thread waiting, no-op

Multithreaded Web Server with Thread Pool

```
ListenerThread {
    for( i=0; i<MAX_THREADS; i++ ) thread[i] = Pthread_create(…)
    forever {
        Receive( request )
        Pthread_mutex_lock( queuelock )
        put request in queue
        Pthread_cond_signal( notempty, queuelock)
        Pthread_mutex_unlock( queuelock )
    }
}
WorkerThread {
    forever {
        Pthread_mutex_lock( queuelock )
        Pthread_cond_wait( notempty, queuelock )
        take request out of queue
        Pthread_mutex_unlock( queuelock )
        read file from disk
        Send( reply )
    }
}
```

Incorrect

- All worker threads busy (none waiting)
- Listener does a signal
- No thread waiting: signal is no-op
- Worker thread finishes what it was doing
  - Will do a wait
  - Although request is waiting in queue

In General

- Signals have no memory
- Forgotten if no thread waiting
- So need an extra variable to remember them

Multithreaded Web Server with Thread Pool

```
ListenerThread {
    for( i=0; i<MAX_THREADS; i++ ) thread[i] = Pthread_create(…)
    forever {
        Receive( request )
        Pthread_mutex_lock( queuelock )
        put request in queue
        avail++
        Pthread_cond_signal( notempty, queuelock)
        Pthread_mutex_unlock( queuelock )
    }
}
WorkerThread {
    forever {
        Pthread_mutex_lock( queuelock )
        if( avail <= 0 ) Pthread_cond_wait( notempty, queuelock )
        take request out of queue
        avail--
        Pthread_mutex_unlock( queuelock )
        read file from disk
        Send( reply )
    }
}
```

Note

- Should now be clear why mutex must be held
- Avail is a shared data item
- Without mutex could have data race

Imagine Solution Without Locks

```
ListenerThread {
    for( i=0; i<MAX_THREADS; i++ ) thread[i] = Pthread_create(…)
    forever {
        Receive( request )
        Pthread_mutex_lock( queuelock )
        put request in queue
        avail++
        Pthread_cond_signal( notempty, queuelock)
        Pthread_mutex_unlock( queuelock )
    }
}
WorkerThread {
    forever {
        Pthread_mutex_lock( queuelock )
        if( avail <= 0 ) Pthread_cond_wait( notempty, queuelock )
        take request out of queue
        avail--
        Pthread_mutex_unlock( queuelock )
        read file from disk
        Send( reply )
    }
}
```

Example: One Worker Thread

- Worker checks avail and finds it to be 0
- Worker interrupted and listener runs
- Listener sets avail to 1 and signals
- No thread is waiting, so signal is no-op
- Listener interrupted and worker runs
- Worker does a wait
- Incorrect: worker waits with request in queue

Back to Solution With Locks

```
ListenerThread {
    for( i=0; i<MAX_THREADS; i++ ) thread[i] = Pthread_create(…)
    forever {
        Receive( request )
        Pthread_mutex_lock( queuelock )
        put request in queue
        avail++
        Pthread_cond_signal( notempty, queuelock)
        Pthread_mutex_unlock( queuelock )
    }
}
WorkerThread {
    forever {
        Pthread_mutex_lock( queuelock )
        if( avail <= 0 ) Pthread_cond_wait( notempty, queuelock )
        take request out of queue
        avail--
        Pthread_mutex_unlock( queuelock )
        read file from disk
        Send( reply )
    }
}
```

Still not quite correct

- Q is empty, thread W1 waits
- Thread L puts request in Q
  - Sets avail to 1
  - Signals
  - W1 is unblocked
- Thread W2 runs and takes something out of Q
  - Sets avail to 0
- Now W1 runs
  - It must check the value of avail

Pthreads: Condition Variables

- Pthread_cond_wait(cond, mutex)
  - Wait for a signal on cond
  - Release mutex
- Pthread_cond_signal(cond, mutex)
- Pthread_cond_broadcast(cond, mutex)

- Must hold mutex when calling any of these!

Pthreads: Condition Variables

- Pthread_cond_wait(cond, mutex)
- Pthread_cond_signal(cond, mutex)
  - Signal one thread waiting on cond
  - Signaled thread re-acquires mutex
    - At some later time, not necessarily immediately
  - If no thread waiting, no-op
- Pthread_cond_broadcast(cond, mutex)

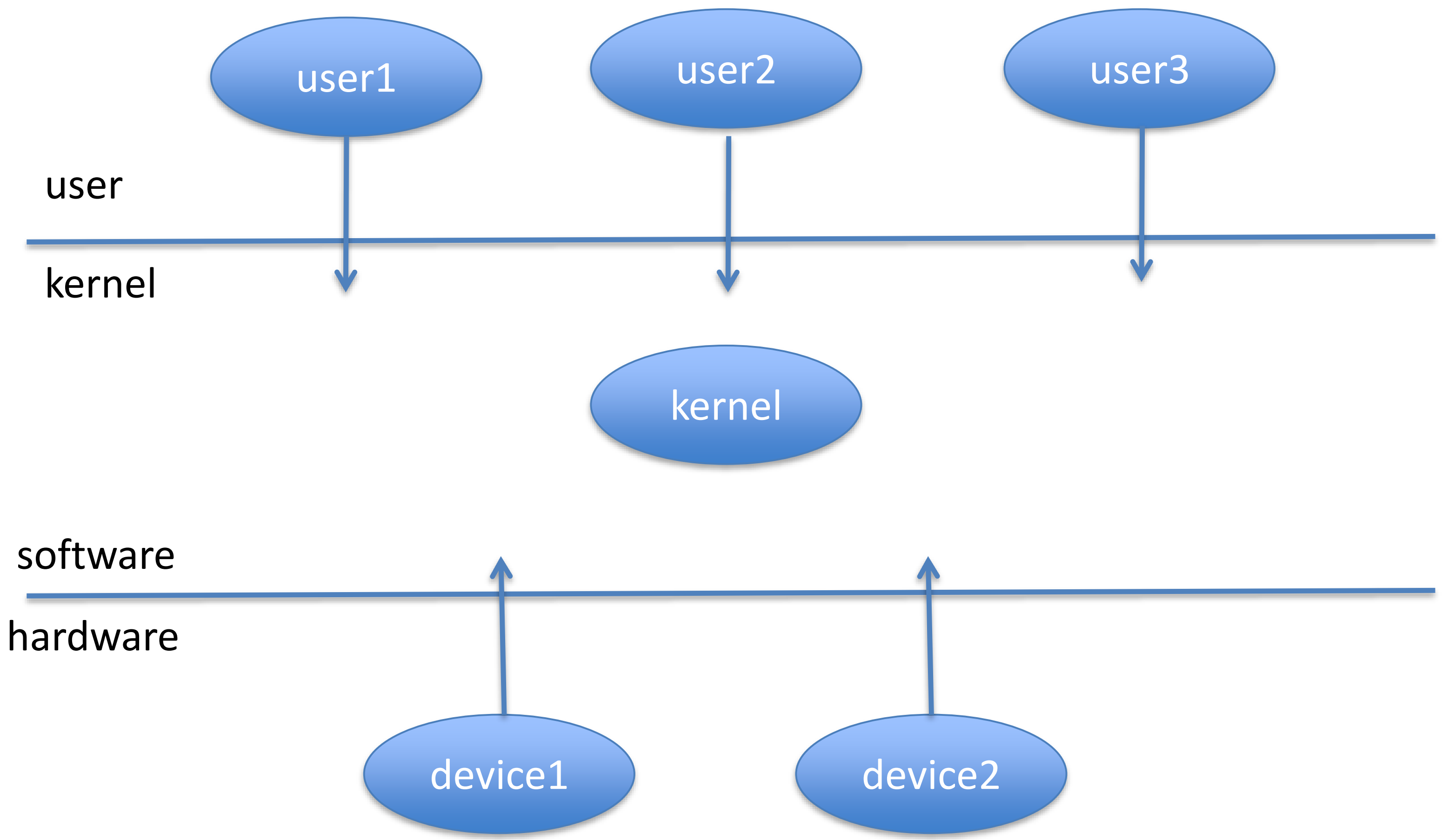
Multithreaded Web Server with Thread Pool

```
ListenerThread {
    for( i=0; i<MAX_THREADS; i++ ) thread[i] = Pthread_create(…)
    forever {
        Receive( request )
        Pthread_mutex_lock( queuelock )
        put request in queue
        avail++
        Pthread_cond_signal( notempty, queuelock)
        Pthread_mutex_unlock( queuelock )
    }
}
WorkerThread {
    forever {
        Pthread_mutex_lock( queuelock )
        while( avail <= 0 ) Pthread_cond_wait( notempty, queuelock )
        take request out of queue
        avail--
        Pthread_mutex_unlock( queuelock )
        read file from disk
        Send( reply )
    }
}
```

Kernel Multithreading: Kernel is a Server

- Requests from users
  - System calls
  - Traps
- Requests from devices
  - Interrupts

Kernel as a Server



Kernel is Event-Driven Program

- Nothing to do } Do nothing

- Interrupt (from device)
- Trap (from process)
- System call (from process}

} Start running

Kernel Code

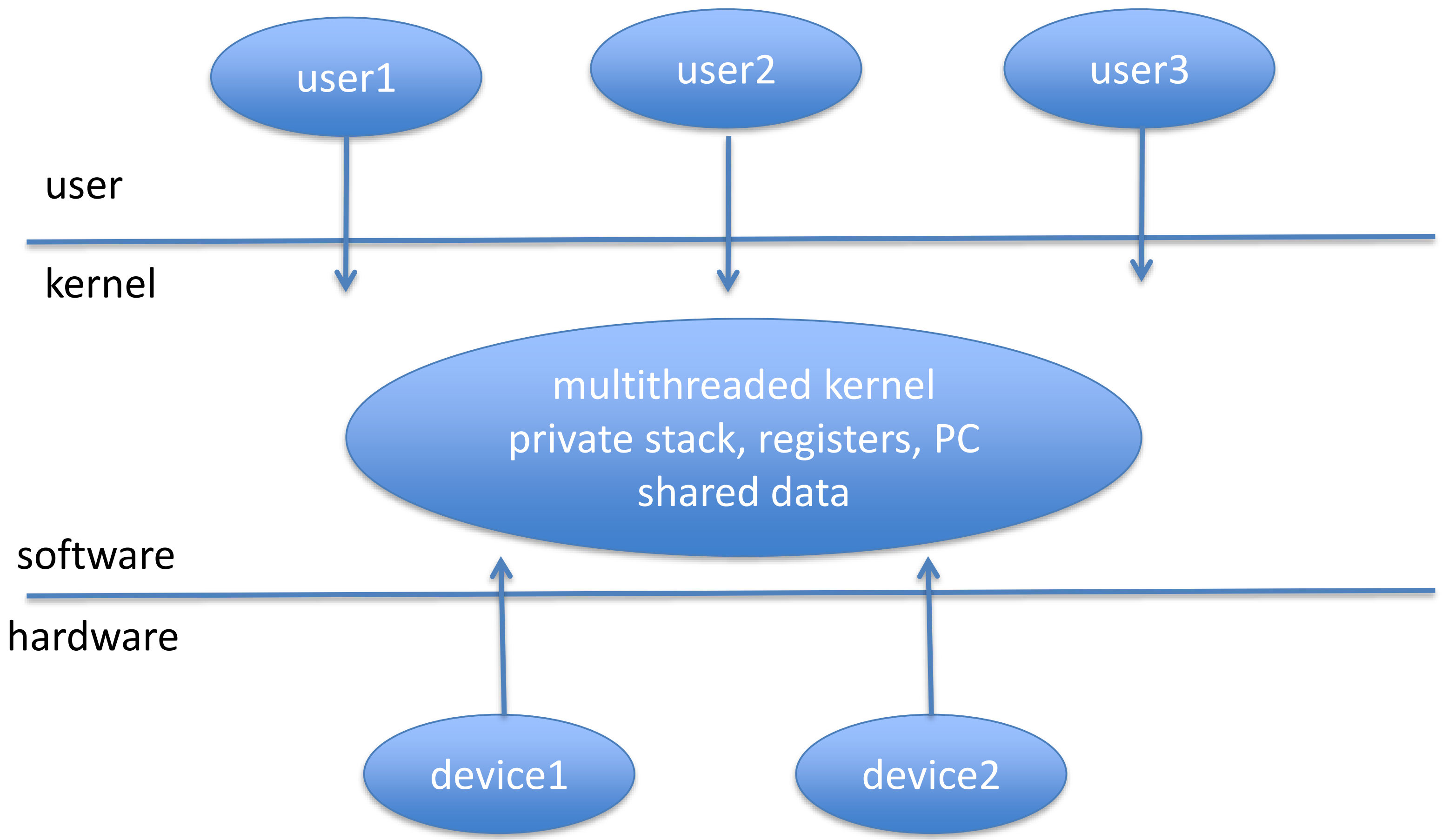
```
InterruptVector[1] = address of interrupt 1 handler routine
InterruptVector[2] = address of interrupt 2 handler routine
…

TrapVector[1] = address of trap 1 handler routine
TrapVector[2] = address of trap 2 handler routine
…

SystemCallVector[1] = address of system call 1 handler routine
SystemCallVector[2] = address of system call 2 handler routine
….


forever {
    wait for something to happen
}
```

Kernel as a Server



For Simplicity

- One kernel thread for each user thread
- Called 1-to-1 mapping
- Is the case in Linux
- Not in other OSs

How does it work? User to Kernel

- User thread makes system call
- Switch to kernel mode
- PC = system call handler routine
- SP = kernel stack of kernel thread

How does it work? Kernel to User

- SP = stack of user thread
- PC = user thread PC (after system call)
- Return from kernel mode
- Run in user thread

Note: Separate Stack

- User thread and corresponding kernel thread have separate stacks
- Why? Because of security
  - while one thread of a process in kernel
  - other thread could modify stack

Kernel Synchronization

- Different kernel threads access shared data
- Must be synchronized
- As in any multithreaded program
- Using a kernel synchronization library
  - Not Pthreads (is a user-level library)

What Makes Kernel Different?

- In addition to kernel threads
- Also interrupts

How does it work?

- Device interrupt
- PC = interrupt handler
- SP = interrupt thread stack
- Run interrupt handler

Kernel Synchronization

- Different kernel threads access shared data
- Must be synchronized
- As in any multithreaded program
- But interrupts make things different

Interrupts

- Must be served quickly
- Interrupt handling must not block

Solution

- Add another set threads
  - Soft interrupt threads
- Interrupt
  - Does absolute minimum to service device
  - Never blocks!
  - Put request in queue for soft interrupt thread
  - Get soft interrupt thread ready
- Soft interrupt thread
  - Does bulk of work

Advantages

- Interrupts can be served quickly
- Narrow interface
  - Interrupt and rest of the kernel
- Soft interrupt threads ~ other kernel threads
  - With some exceptions, not going into it here

Summary

- Why shared data and multithreading?
- Application multithreading
  - Division of work
  - Synchronization of shared data
  - Fine-grain locking
  - Privatization
- Kernel multithreading
  - User threads vs. kernel threads
  - Interrupts
  - Soft interrupt threads