Implementing Threads 3

Blocking Locks

```
void blocking_lock(mutex_t *mut) {
  if (mut->holder != 0) {
    enqueue(mut->wait_queue,
       CurrentThread);
    uthread_switch();
  } else
    mut->holder = CurrentThread;
}
```

```
void blocking_unlock(mutex_t *mut) {
  if (queue_empty(mut->wait_queue))
    mut->holder = 0;
  else {
    mut->holder =
       dequeue(mut->wait_queue);
    enqueue(RunQueue, mut->holder);
  }
}
```

Does it work?

Working Blocking Locks (?)

```
void blocking_lock(mutex_t *mut) {
  spin_lock(&mut->spinlock);
  if (mut->holder != 0) {
    enqueue(mut->wait_queue,
          CurrentThread);
    spin_unlock(&mut->spinlock);
    uthread_switch();
  } else {
    mut->holder = CurrentThread;
    spin_unlock(&mut->spinlock);
  }
}
```

```
void blocking_unlock(mutex_t *mut) {
  spin_lock(&mut->spinlock);
  if (queue_empty(
         mut->wait_queue)) {
    mut->holder = 0;
  } else {
    mut->holder =
        dequeue(mut->wait_queue);
    enqueue(RunQueue,
        mut->holder);
  }
  spin_unlock(&mut->spinlock);
}
```

Quiz 1

This

a) always works
b) sometimes doesn’t work
c) never works

Futexes

- **Safe, *efficient* kernel conditional queueing in Linux**
- **All operations performed atomically**
  - `futex_wait(futex_t *futex, int val)`
    - **if** `futex->val` **is equal to** `val`**, then sleep**
    - **otherwise return**
  - `futex_wake(futex_t *futex)`
    - **wake up one thread from** `futex`'s **wait queue, if there are any waiting threads**

Ancillary Functions

- **int** atomic_inc(**int** *val)
  - **add 1 to** *val**, return its original value**
- **int** atomic_dec(**int** *val)
  - **subtract 1 from** *val**, return its original value**

Attempt 1

```
void lock(futex_t *futex) {
  int c;
  while ((c = atomic_inc(&futex->val)) != 0)
    futex_wait(futex, c+1);
}

void unlock(futex_t *futex) {
  futex->val = 0;
  futex_wake(futex);
}
```

Attempt 2

```
void lock(futex_t *futex) {
  int c;
  if ((c = CAS(&futex->val, 0, 1) != 0)
    do {
      if (c == 2 || (CAS(&futex->val, 1, 2) != 0))
        futex_wait(futex, 2);
    while ((c = CAS(&futex->val, 0, 2)) != 0))
}

void unlock(futex_t *futex) {
  if (atomic_dec(&futex->val) != 1) {
    futex->val = 0;
    futex_wake(futex);
  }
}
```

Quiz 2
Does it work?

a) No
b) Yes

Blocking Locks in MThreads

- We could use futexes, but don’t
- *uthread_switch* gets an additional argument
  - a POSIX mutex (representing a spin lock)
  - unlock it after getting out of the context of the calling thread

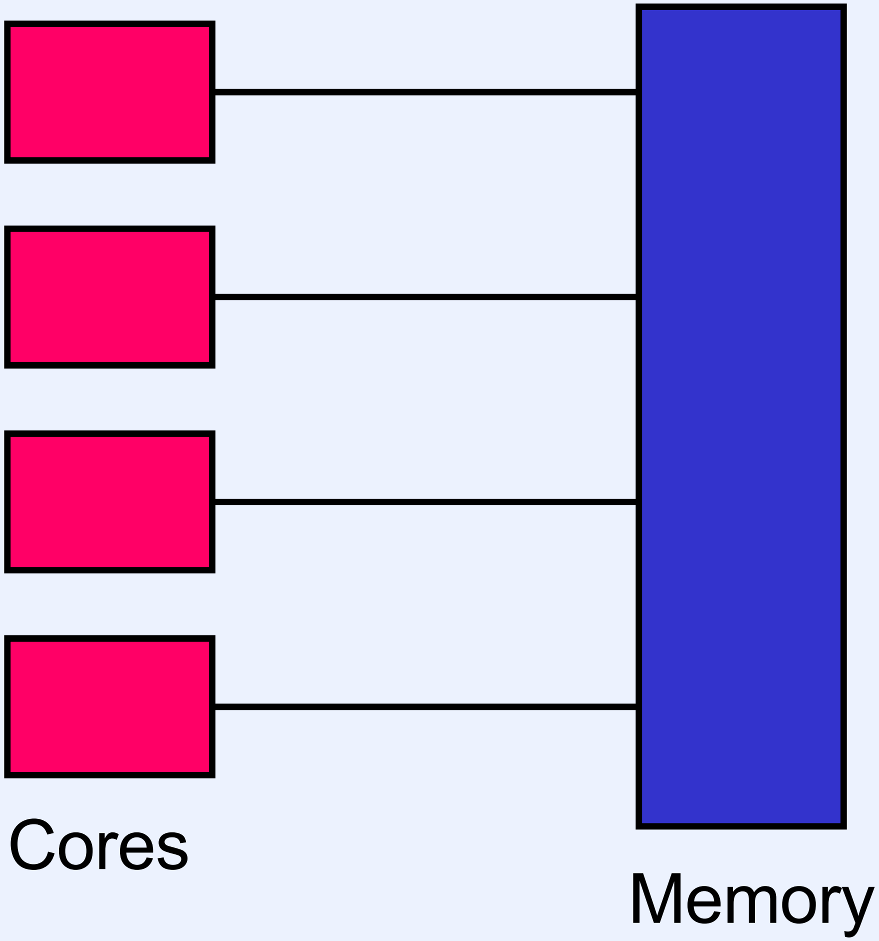
Actual Code

```
uthread_mtx_lock(uthread_mtx_t *mtx) {
    uthread_nopreempt_on();
    pthread_mutex_lock(&mtx->m_pmut);
    if (mtx->m_owner == NULL) {
        mtx->m_owner = ut_curthr;
        pthread_mutex_unlock(&mtx->m_pmut);
        uthread_nopreempt_off();
    } else {
        ut_curthr->ut_state = UT_WAIT;
        uthread_switch(&mtx->m_waiters, 0, &mtx->m_pmut);
        uthread_nopreempt_off();
    }
}
```

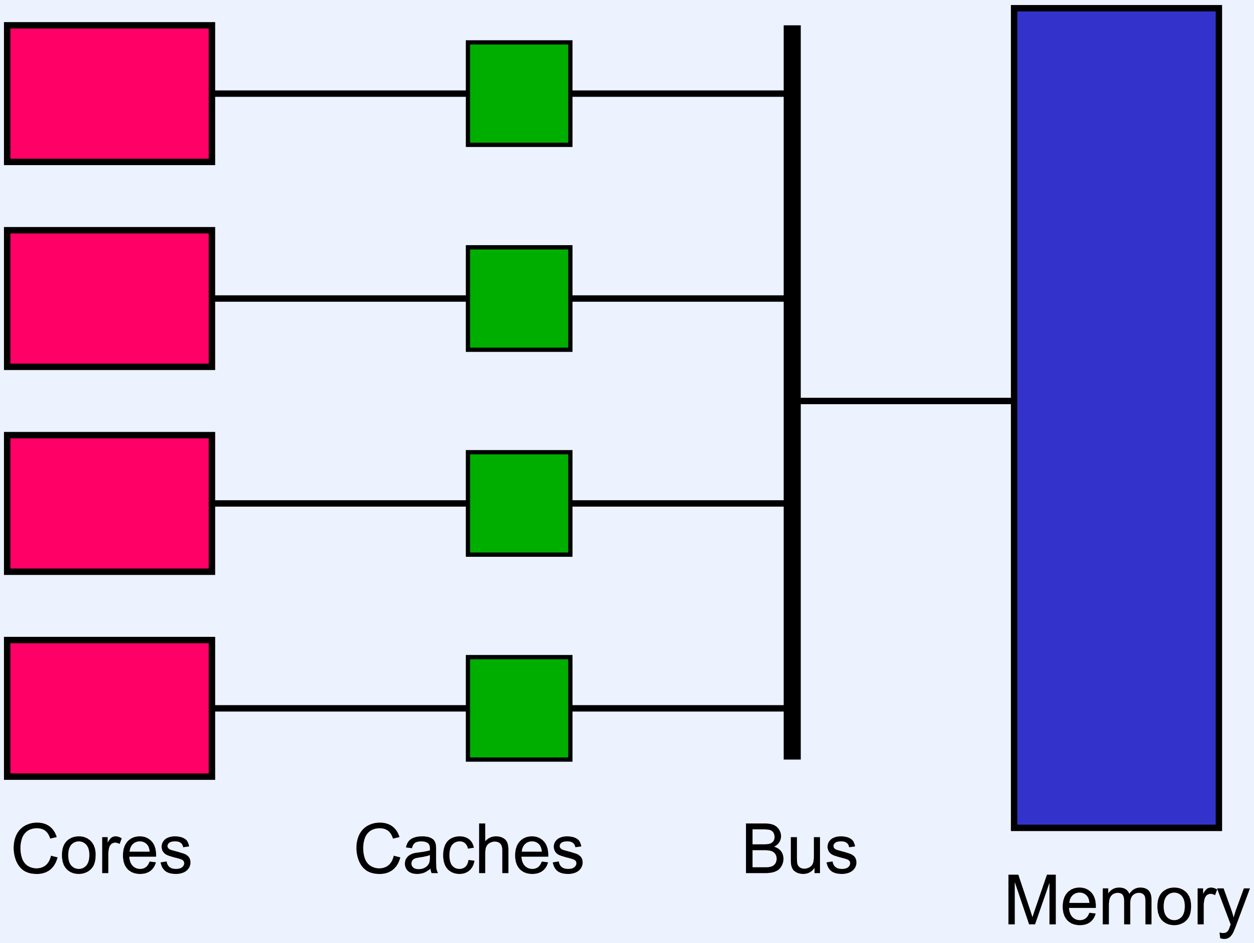
MP Memory Issues

- Naive view is that all processors in MP system see same memory contents at all times
  - they don’t

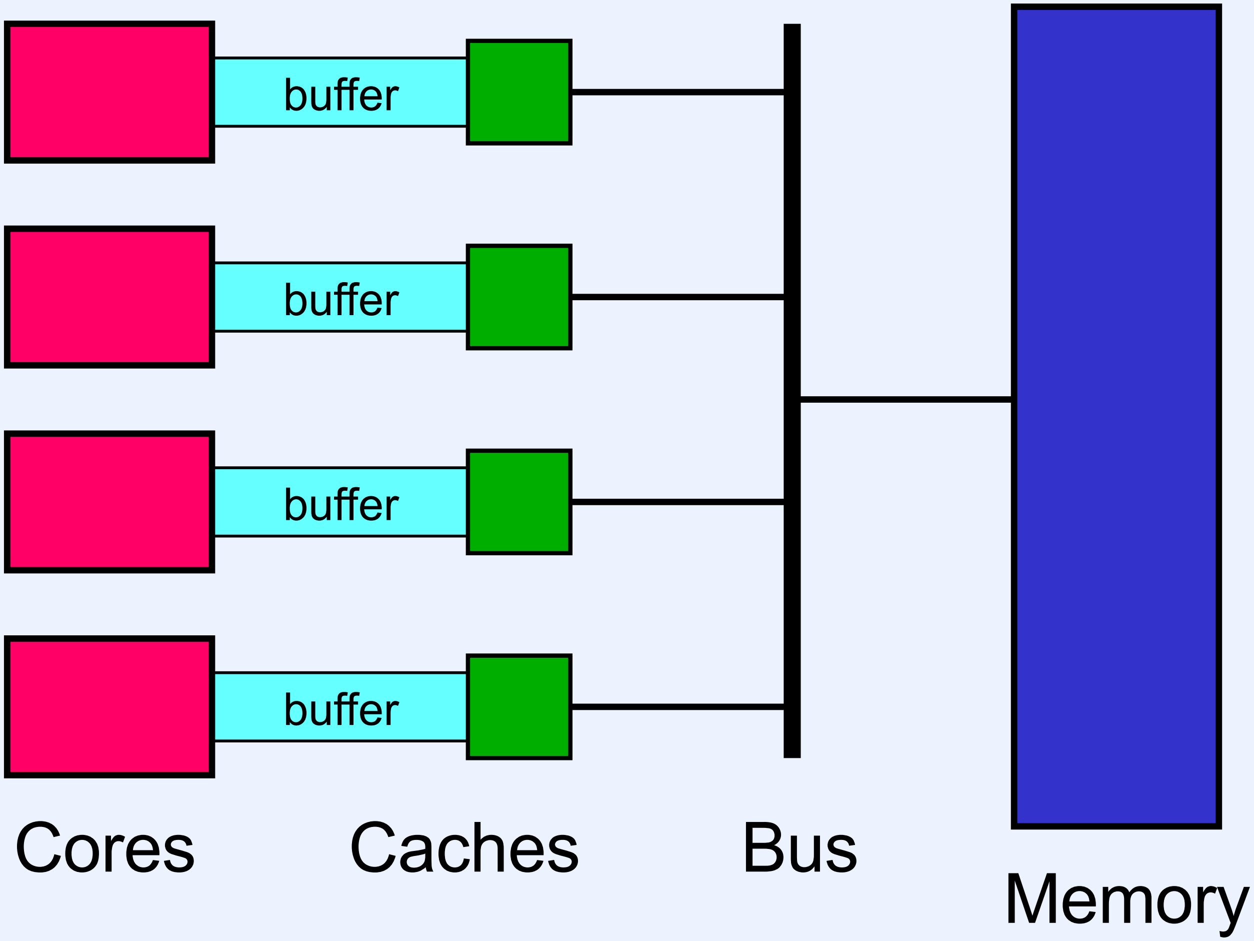
Multi-Core Processor: Simple View



Multi-Core Processor: More Realistic View



Multi-Core Processor: Even More Realistic



Concurrent Reading and Writing

Thread 1:

```
i = shared_counter;
```

Thread 2:

```
shared_counter++;
```

Mutual Exclusion w/o Mutexes

```
void peterson(long me) {
  static long loser;                       // shared
  static long active[2] = {0, 0};          // shared
  long other = 1 - me;                     // private
  active[me] = 1;
  loser = me;
  while (loser == me && active[other])
    ;
  // critical section
  active[me] = 0;
}
```

Quiz 3

With delayed stores

a) works
b) doesn’t work

Busy-Waiting Producer/Consumer

```
void producer(char item) {

  while(in - out == BSIZE)
    ;

  buf[in%BSIZE] = item;

  in++;
}
```

```
char consumer( ) {
  char item;
  while(in - out == 0)
    ;

  item = buf[out%BSIZE];

  out++;

  return(item);
}
```

Quiz 4

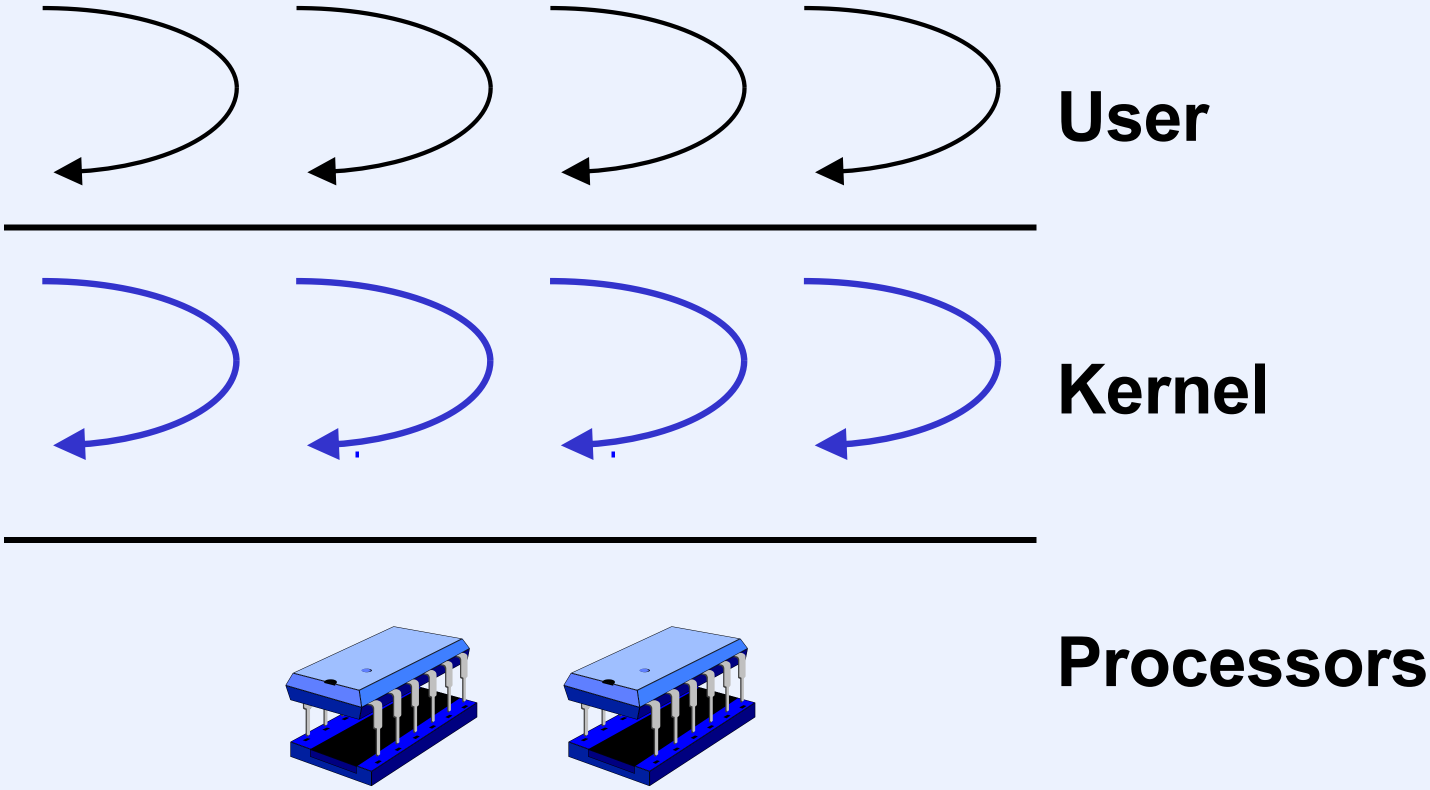
With re-ordered stores

a) works
b) doesn't work

Coping

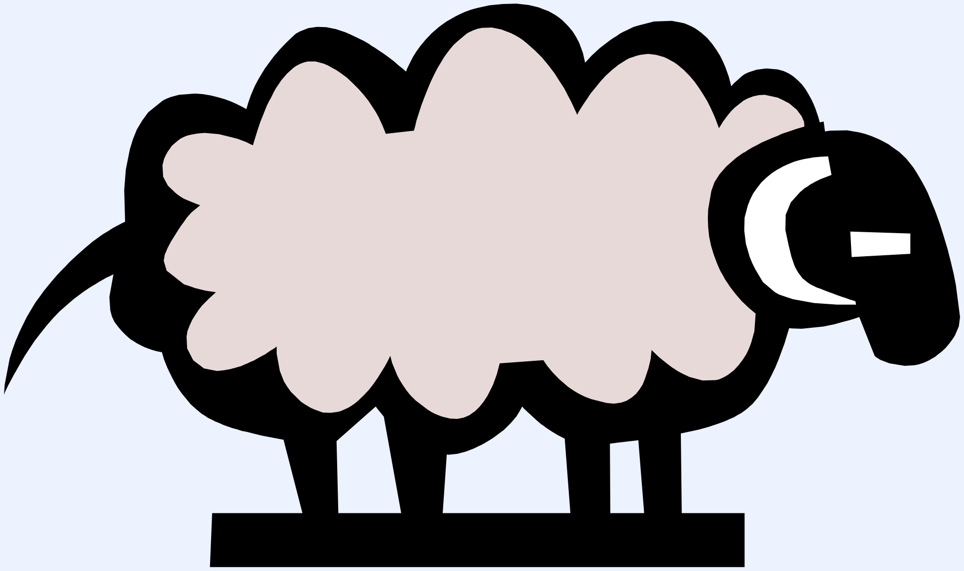
- Use what’s available in the architecture to make sure all cores have the same view of memory (when necessary)
  - – lock prefix on x86
  - – mfence x86 instruction
- Use the synchronization primitives
  - – presumably the implementers knew what they were doing

One-Level Model

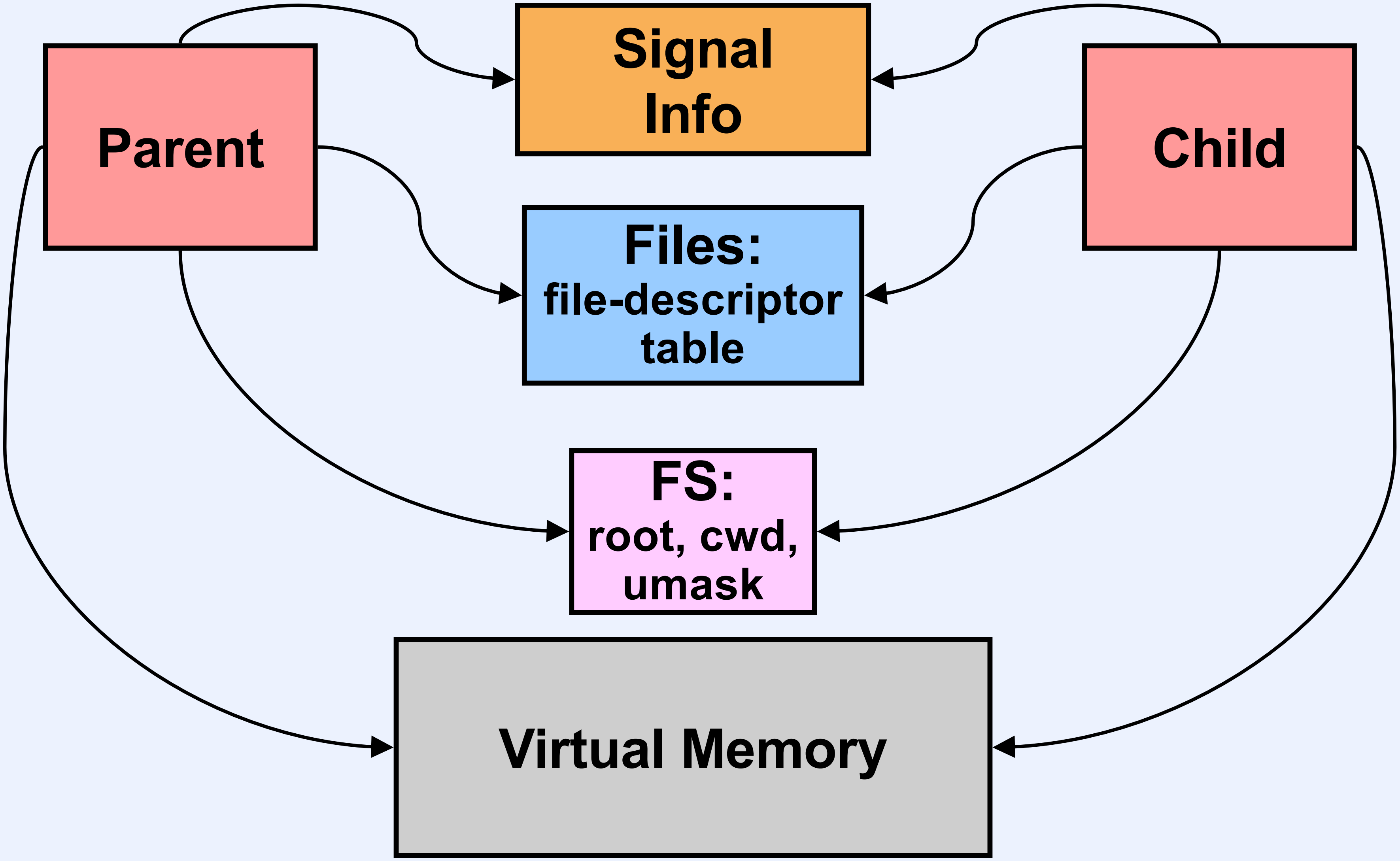


Variable-Weight Processes

- Variant of one-level model
- Portions of parent process selectively *copied* into or *shared* with child process
- Children created using *clone* system call



Cloning



Parent

Child

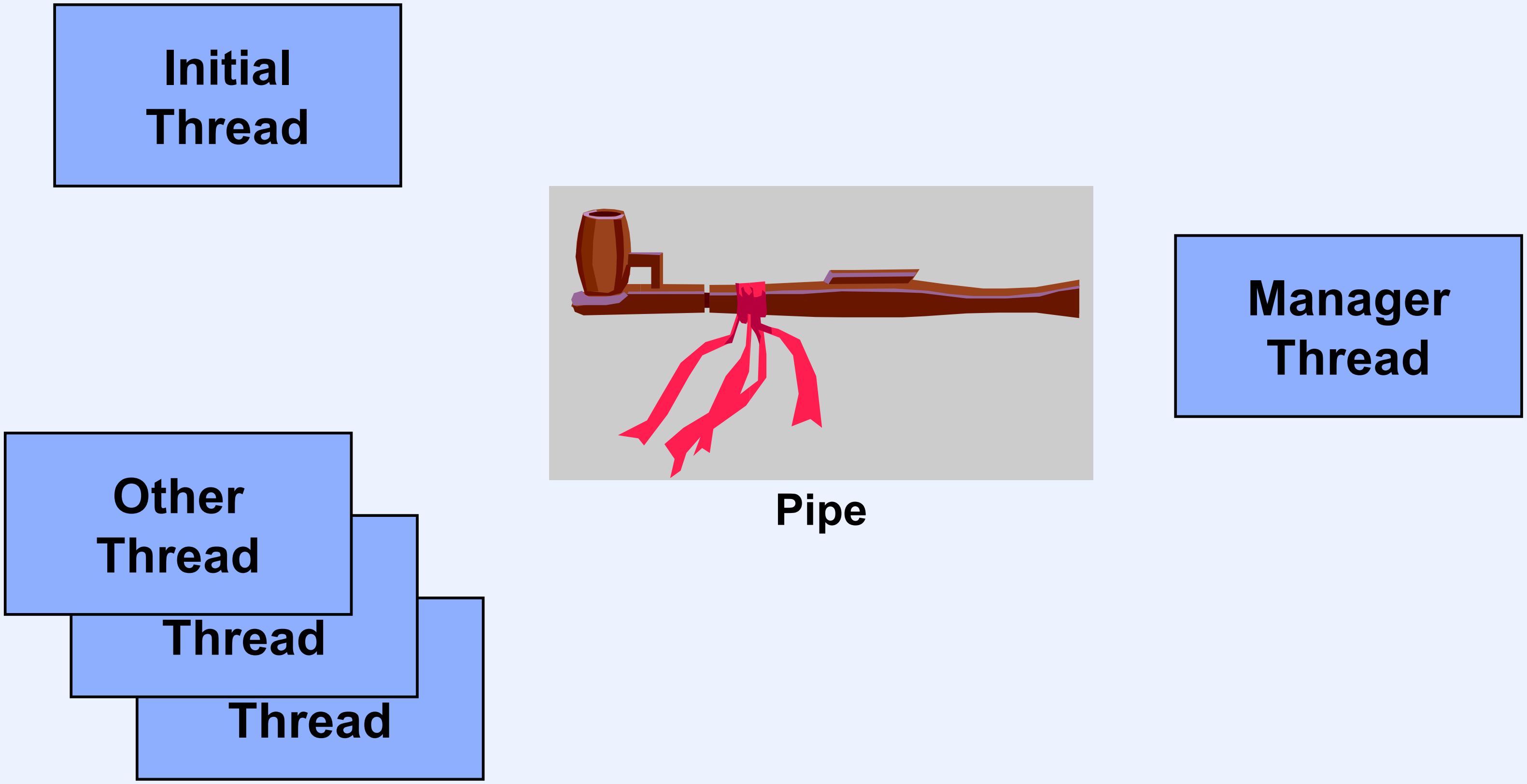
Signal Info

Files: file-descriptor table

FS: root, cwd, umask

Virtual Memory

Linux Threads
(pre 2.6)



Initial Thread

Other Thread

Thread

Thread

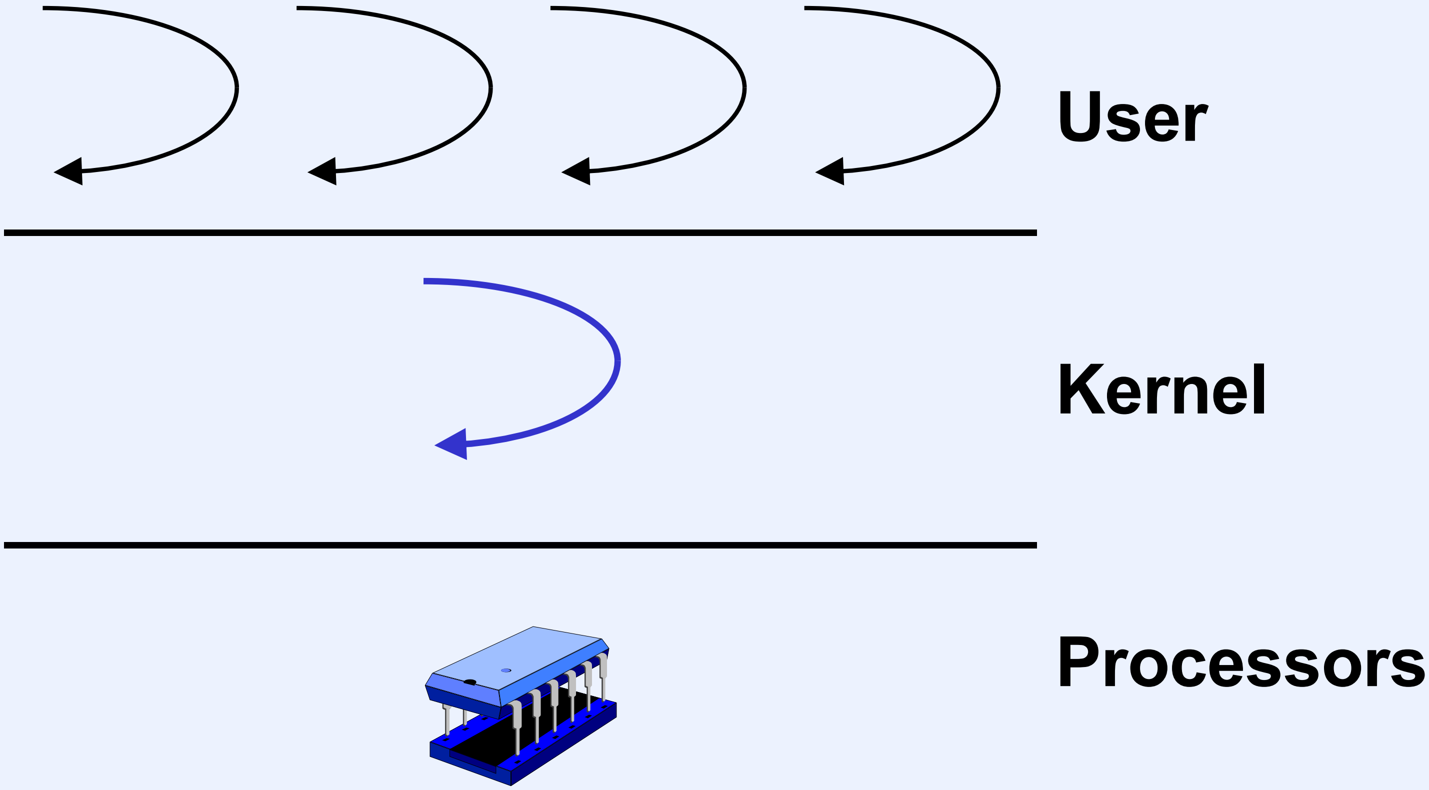
Pipe

Manager Thread

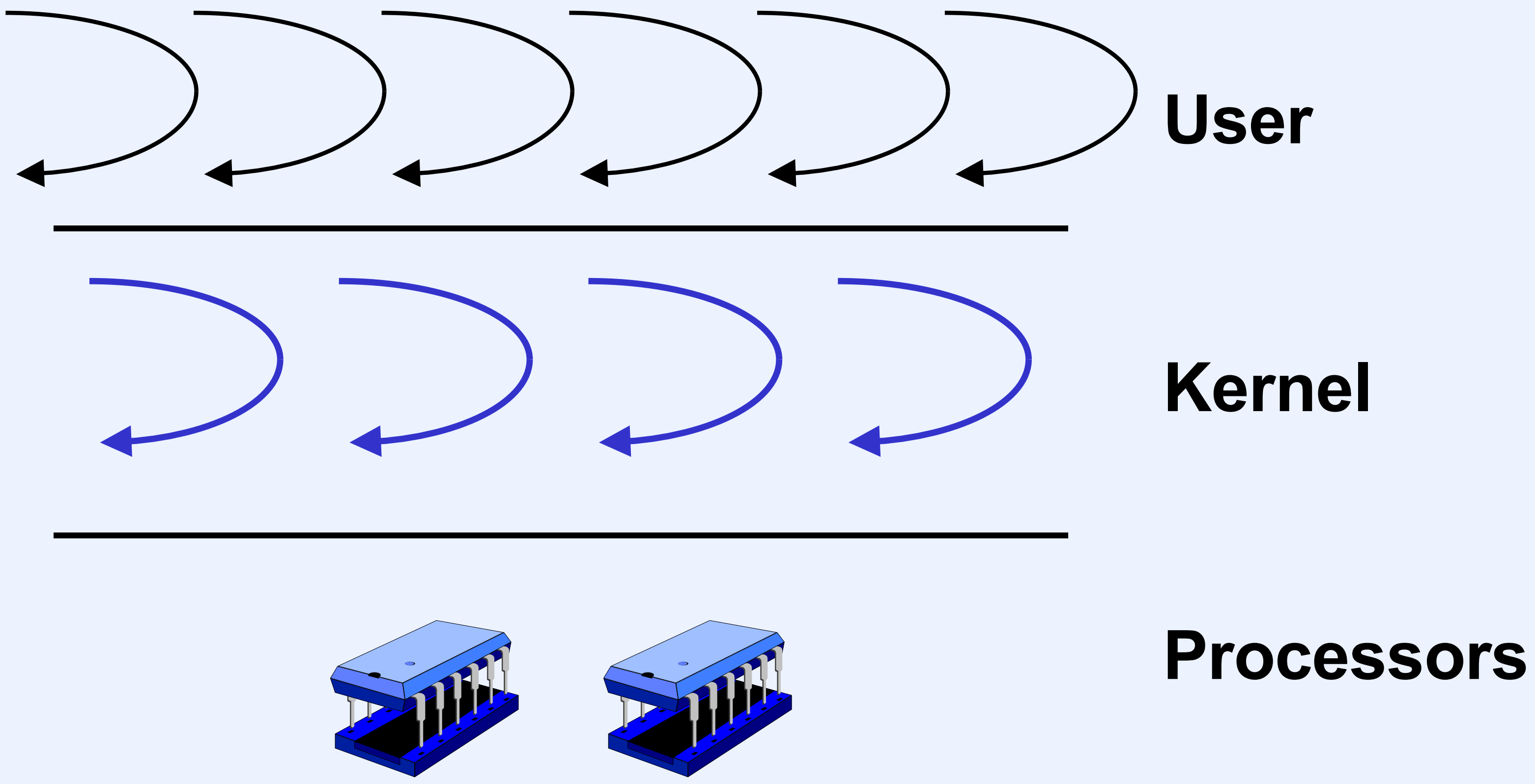
NPTL in Linux 2.6+

- Native POSIX-Threads Library
  - full POSIX-threads semantics on improved variable-weight processes
    - threads of a "process" form a *thread group*
      - *getpid()* returns process ID of first thread in group
      - any thread in group can wait for any other to terminate
      - signals to process delivered by kernel to any thread in group

Two-Level Model
One Kernel Thread



Two-Level Model:
Multiple Kernel Threads



Quiz 5

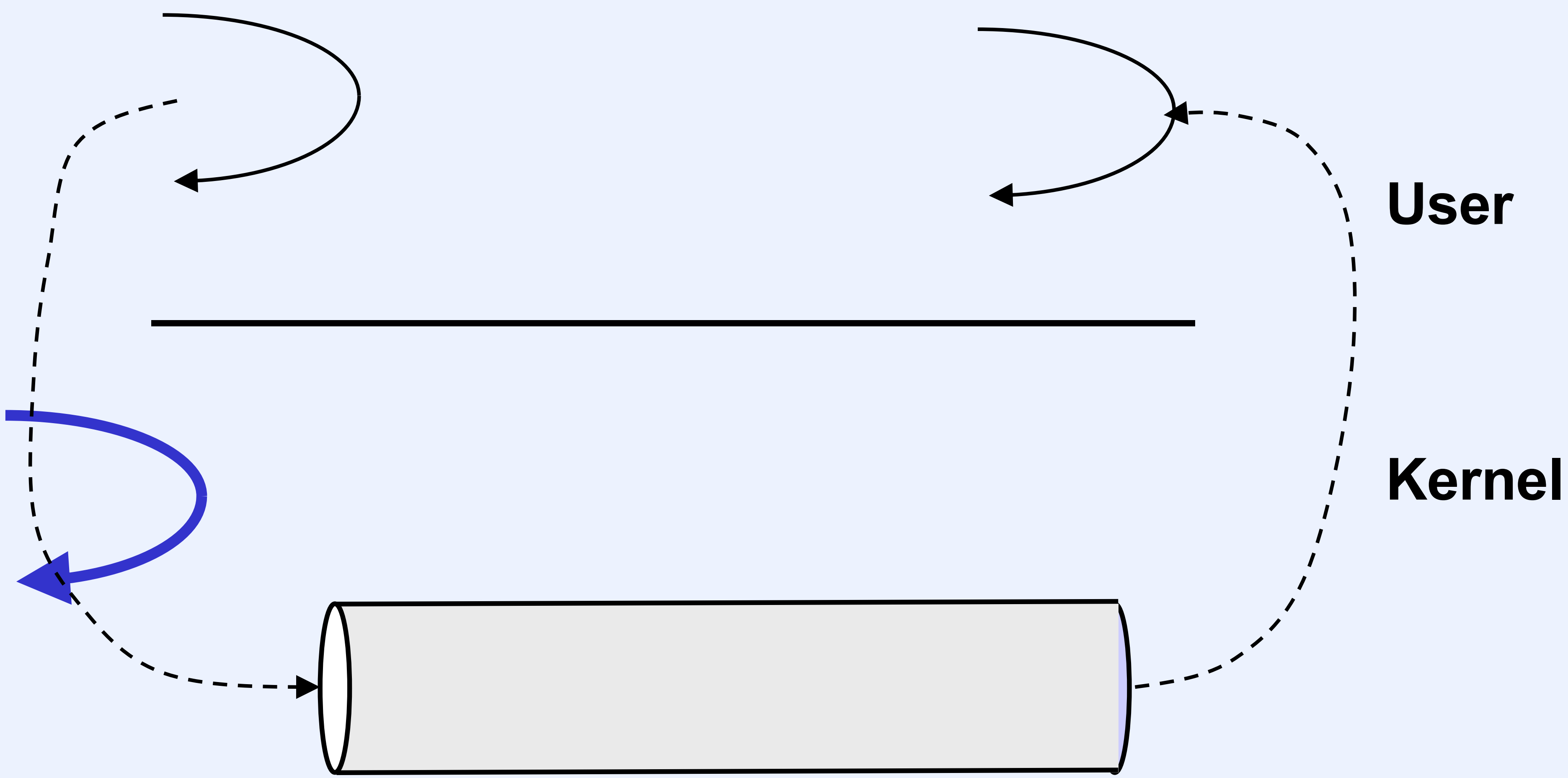
One kernel thread for each user thread is clearly a sufficient number of kernel threads in the two-level model. Is it necessary for maximum concurrency?

a) there are no situations in which that number of threads is necessary, as long as there are at least as many kernel threads as processors.

b) there must always be that number of kernel threads for the two-level model to work well.

c) there are situations in which that number is necessary, but they occur rarely.

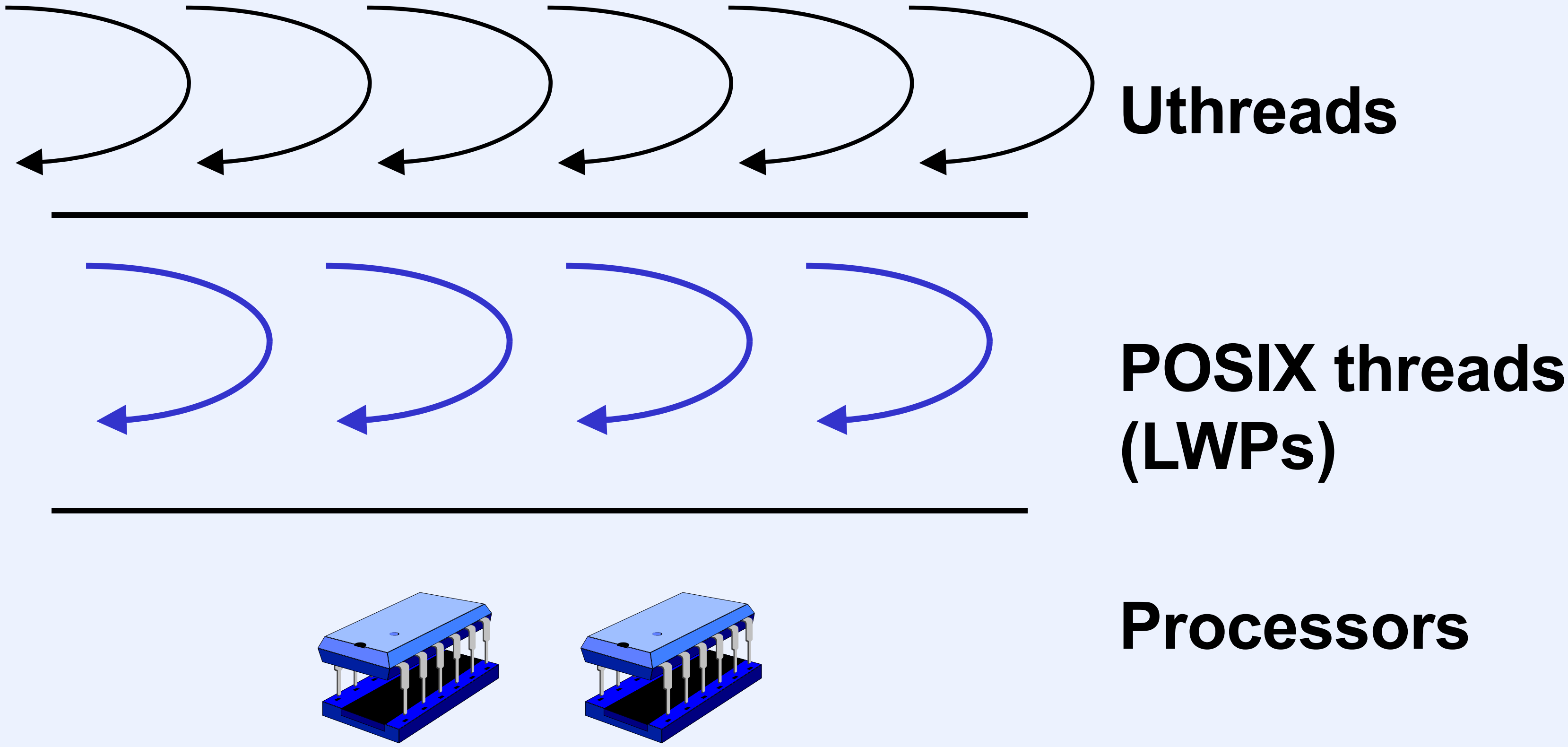
Deadlock



MThreads

- Two-level threads implementation of Uthreads
  - kernel-supported threads are POSIX threads
  - user threads based on your implementation of Uthreads
- Effectively a multiprocessor implementation
  - use POSIX mutexes rather than spin locks
  - use POSIX condition variables rather than the idle loop

Two-Level Model:
MThreads



Uthreads

POSIX threads (LWPs)

Processors

Synchronizing LWPs

```
uthread_switch(...) {
   uthread_mtx_lock(&runq_mtx)
   volatile int first = 1;
   getcontext(&ut_curthr->ut_ctx);
   if (!first) {
      ...
   }
   setcontext(&curlwp->lwp_ctx);
}
lwp_switch() {
   ...
   ut_curthr = top_priority_thread(&runq);
   uthread_mtx_unlock(&runq_mtx);
   setcontext(&ut_curthr->ut_ctx);
   ...
}
```

Synchronizing LWPs (2)

```
uthread_switch(...) {
   spin_lock(&runq_mtx)
   volatile int first = 1;
   getcontext(&ut_curthr->ut_ctx);
   if (!first) {
      ...
   }
   setcontext(&curlwp->lwp_ctx);
}
lwp_switch() {
   ...
   ut_curthr = top_priority_thread(&runq);
   spin_unlock(&runq_mtx);
   setcontext(&ut_curthr->ut_ctx);
   ...
}
```

Synchronizing LWPs (3)

```
uthread_switch(...) {
   pthread_mutex_lock(&runq_mtx)
   volatile int first = 1;
   getcontext(&ut_curthr->ut_ctx);
   if (!first) {
      ...
   }
   setcontext(&curlwp->lwp_ctx);
}
lwp_switch() {
   ...
   ut_curthr = top_priority_thread(&runq);
   pthread_mutex_unlock(&runq_mtx);
   setcontext(&ut_curthr->ut_ctx);
   ...
}
```

POSIX Mutexes and MThreads

- POSIX mutexes used to synchronize activity among LWPs
- Problem case
  - uthread (running on LWP) locks mutex
  - clock interrupt occurs, uthread yields LWP to another uthread
  - that uthread (running on same LWP) locks same mutex
  - deadlock: LWP attempting to lock mutex it currently has locked
- Solution
  - mask interrupts while thread has mutex locked

Example

```
void uthread_wake(uthread_t *uthr) {
    pthread_mutex_lock(&runq_mtx);
    ...
    // wake up thread, put it on runq
    ...
    pthread_mutex_unlock(&runq_mtx);
}
```

Example: Fixed

```
void uthread_wake(uthread_t *uthr) {
    uthread_noprempt_on();
    pthread_mutex_lock(&runq_mtx);

    ...

    // wake up thread, put it on runq

    ...

    pthread_mutex_unlock(&runq_mtx);
    uthread_nopreempt_off();
}
```

Thread-Local Storage in Mthreads

- `__thread thread_t *ut_curthr;`
  - **reference to the current uthread**
- `__thread lwp_t *curlwp`
  - **reference to the current LWP (POSIX thread)**

- **Thread-Local Storage accesses are not async-signal safe!**
- **Must turn off preemption while using TLS**
  - **otherwise thead could be preempted and later resumed on another LWP**
  - **TLS pointer refers to the wrong item!**