Ch 5: Processor Management

Bill Cheng

http://merlot.usc.edu/cs402-s16

Processor Management

- Threads *Implementation*
  - lock/mutex implementation on multiprocessors
- Interrupts
- Scheduling
- Linux/Windows Scheduler

5.1 Threads Implementations

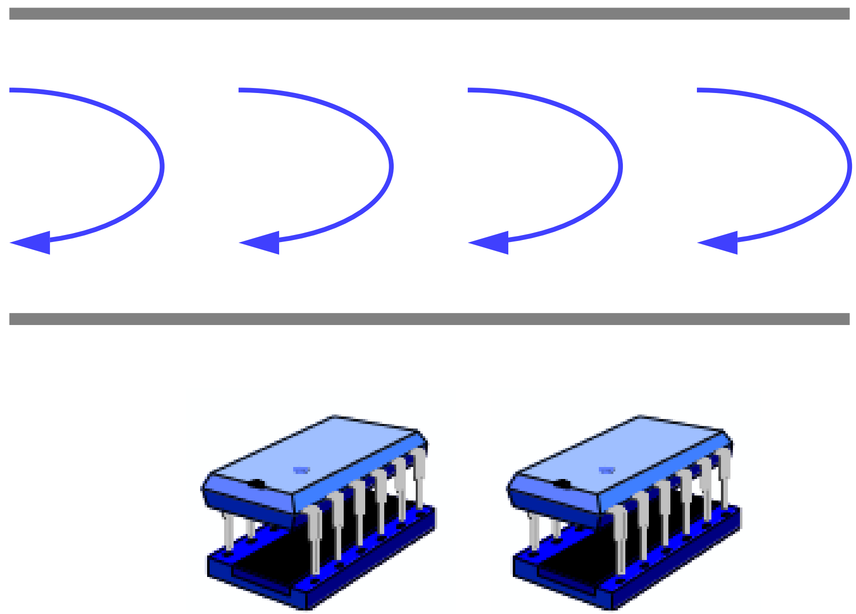
- *Strategies*
- A Simple Thread Implementation
- Multiple Processors

Threads Implementation

- The ultimate goal of the OS is to support user-level applications
  - we will discuss various strategies for supporting threads
- Where are operations on threads implemented?
  - in the kernel?
  - or in user-level library?
- Approaches
  - one-level model (threads are implemented in the kernel)
    - variable-weight processes
  - two-level model (threads are implemented in user library)
    - $N \times 1$
    - $M \times N$
    - scheduler activations model

One-Level Model

User



Kernel

Processors

One-Level Model

- The simplest and most direct approach is the *one-level model*
  - all aspects of the *thread implementation* are *in the kernel*
    - i.e., all thread routines (e.g., `pthread_mutex_lock`) called by user code are all system calls
  - each *user thread* is mapped one-to-one to a *kernel thread*

- If a thread calls `pthread_create()`
  - it's a system call, so it traps into the kernel
  - the kernel creates a thread control block
    - associate it with the process control block
  - the kernel creates a kernel and a user stack for this thread

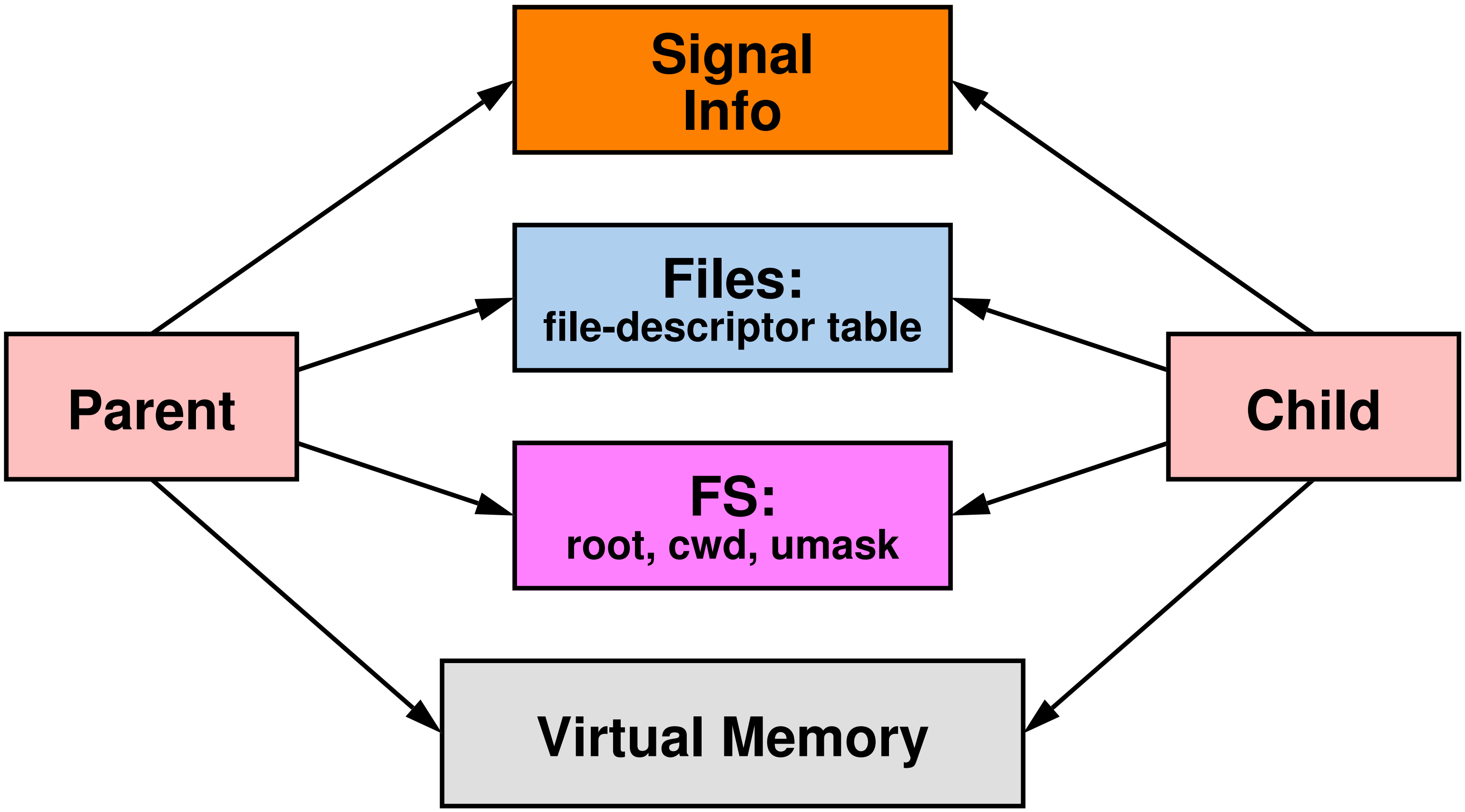
- What about `pthread_mutex_lock()`
  - why does it have to be done in the kernel?
  - it's not necessary to protect the threads from each other!
    - you definitely don't need the kernel to protect threads from each other

One-Level Model

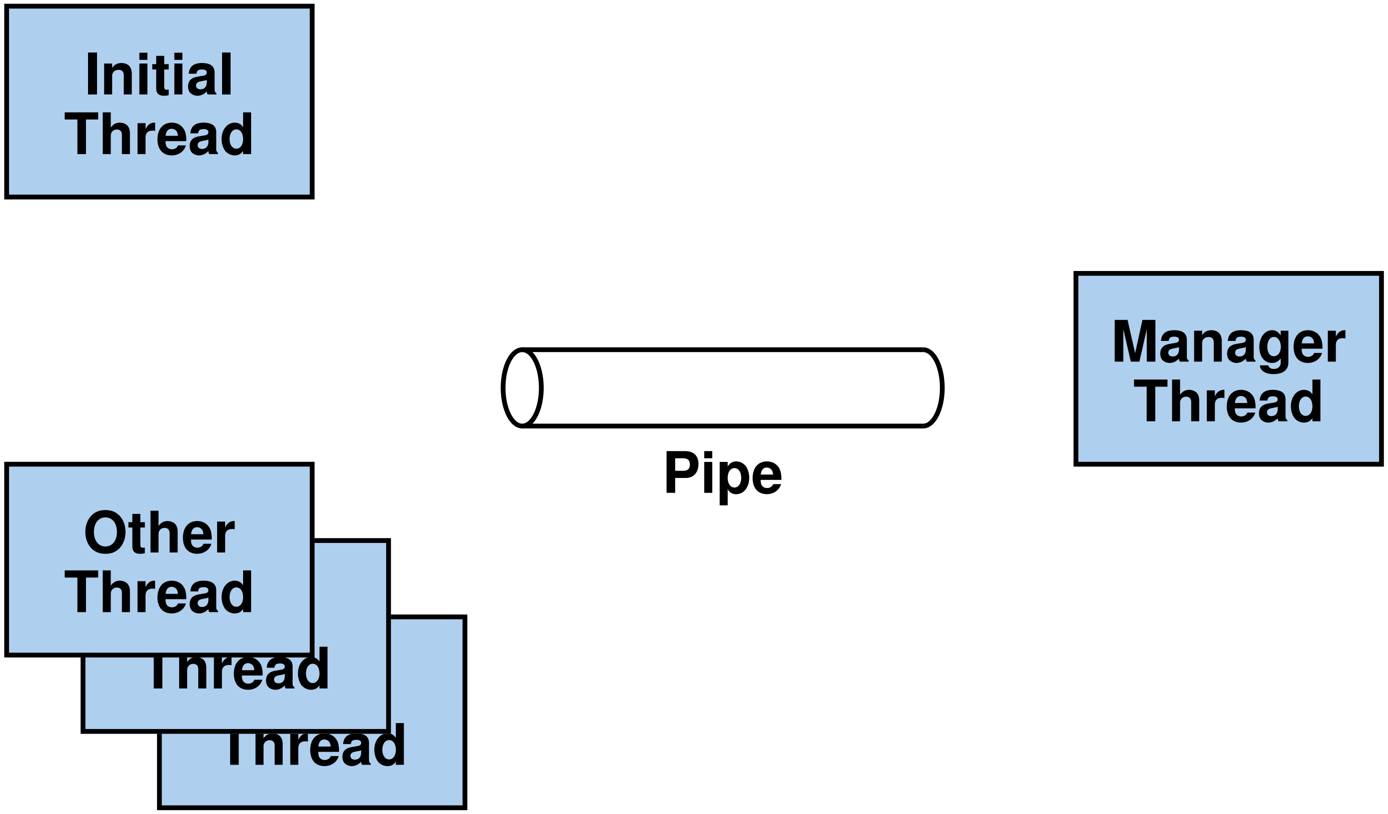
- Problem: system calls are expensive
  - if `pthread_mutex_lock` finds the mutex available, it should return quickly (and lock the mutex)
    - if this can be done in user code, it can be 20 times faster (for the case where the mutex is available)
    - in Win32 threads, an equivalent of a mutex is represented in a user-level data structure
      - if such an object is not locked, it returns quickly
      - if such an object is locked, it makes a system call and blocks in the kernel

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



Linux Threads (pre 2.6)



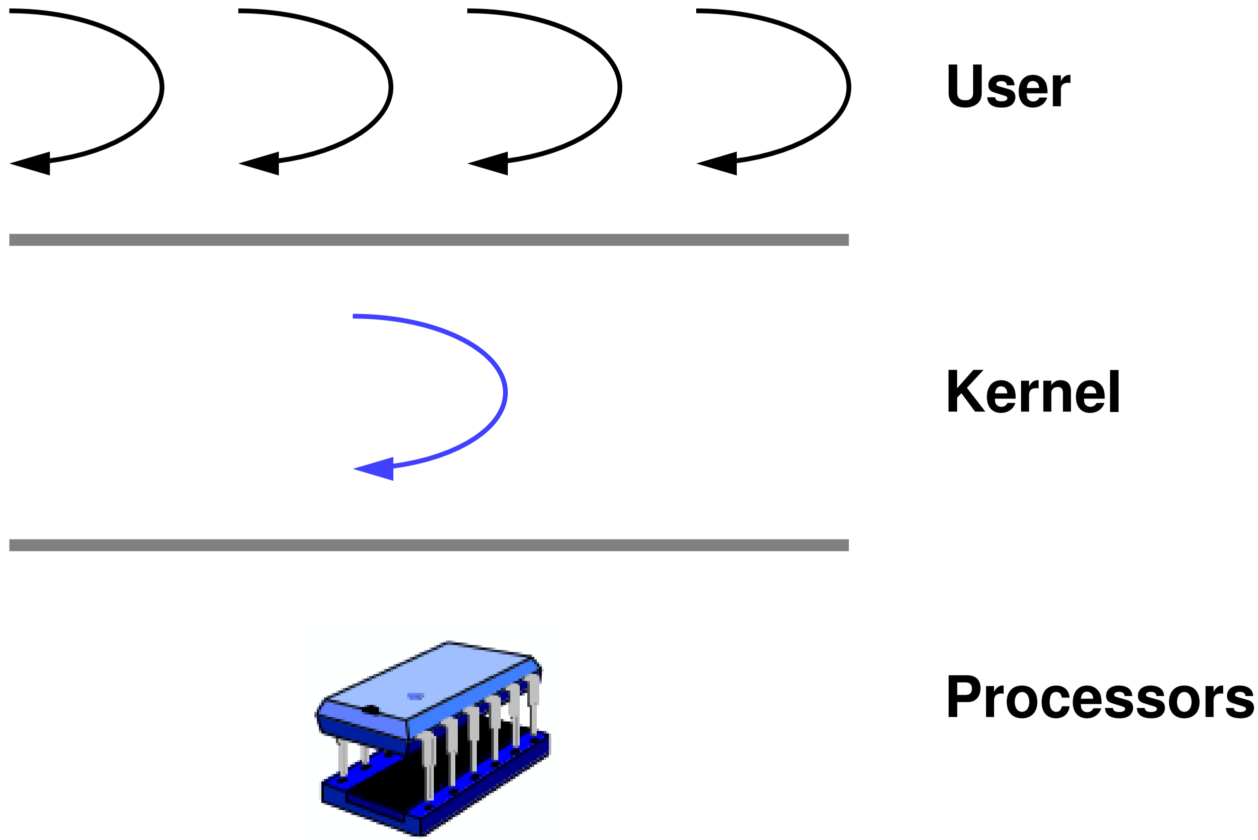
NPTL in Linux 2.6

- 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
  - new kernel-supported synchronization construct: *futex* (fast user-space mutex)
    - used to implement mutexes, semaphores, and condition variables

Two-Level Model

- In the two-level model, a user-level library plays a major role
  - what an user-level application perceives as a thread is implemented within user-level library code
- Two versions
  - single kernel thread (per user process)
  - multiple kernel threads (per user process)

Two-Level Model - One Kernel Thread



- This is one of the earliest ways of implementing threads
  - threads are implemented entirely in the user level
    - thread control block, mutex in user space
    - thread stack allocated by user library code
  - mostly done on uniprocessors

Two-Level Model - One Kernel Thread

- Within a process, user threads are multiplexed not on the processor, but on a kernel-supported thread
  - the OS multiplexes kernel threads (or equivalently, processes) on the processor
  - kernel does *not* know about *the existance of user threads*
- User thread creation
  - a stack and a thread control block is allocated
  - thread is put on a queue of runnable threads
    - wait for its turn to become the running thread
- Synchronization implementation
  - relative straightforward
  - e.g., mutex (one queue per mutex)
    - if a thread must block, it simply queues itself on a wait queue and calls context-switch routine to pass control to the first thread on the runnable queue

Two-Level Model - One Kernel Thread

- This is called the ***N-to-1 Model***
- **Major advantage**
  - **fast, because no system calls for thread-related APIs**
- **Major disadvantage**
  - **what if a thread makes a system call (for a non-thread-related API)?**
    - **it gets blocked in the kernel**
    - **no other user thread in the process can run**

Coping ...

```
ssize_t read(int fd, void *buf, size_t count)
{
  ssize_t ret;
  while (1) {
    if ((ret = real_read(fd, buf, count)) == -1) {
      if (errno == EWOULDBLOCK) {
        sem_wait(&FileSemaphore[fd]);
        continue;
      }
    }
    break;
  }
  return(ret);
}
```

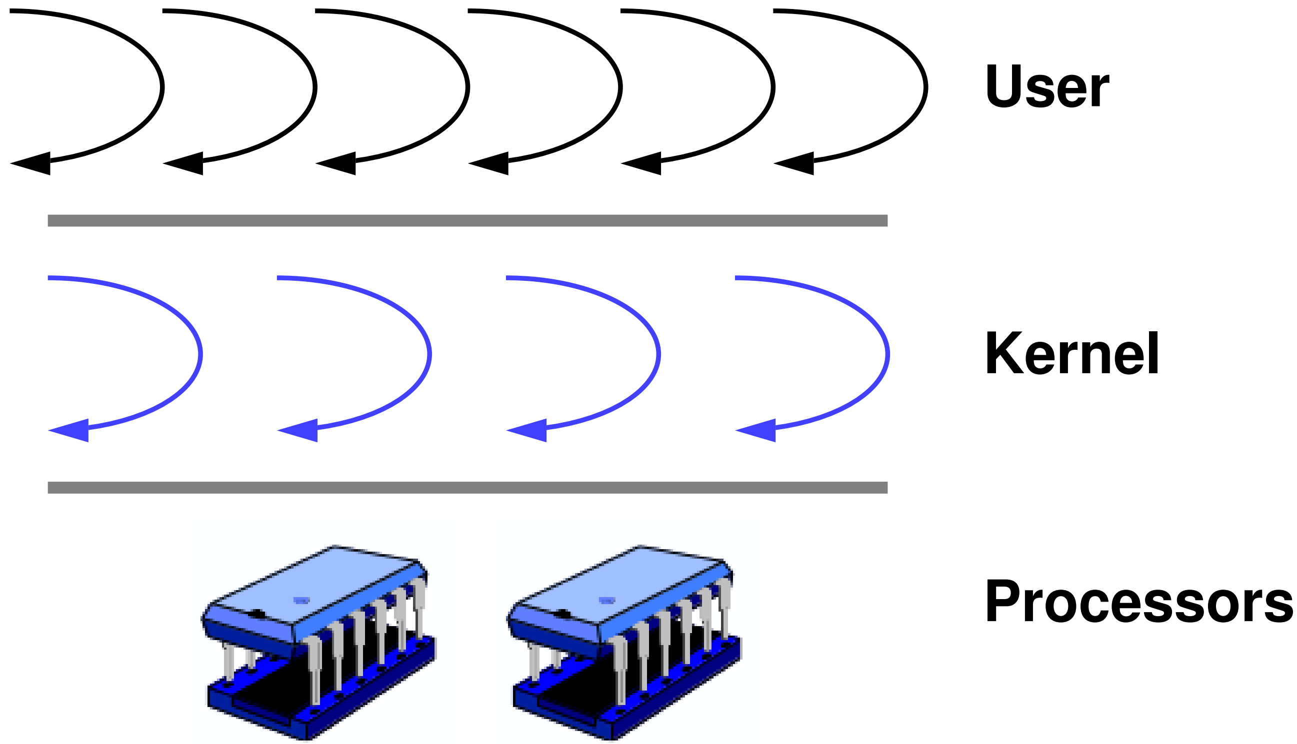
- Solution is to have a non-blocking `read()` called `real_read()`
  - `real_read()` either returns immediately with data in `buf`
  - or returns immediately with an error code in `errno`
    - `EWOULDBLOCK` means that a real `read()` would block, i.e., data is not ready to be read

Coping ...

```
ssize_t read(int fd, void *buf, size_t count)
{
  ssize_t ret;
  while (1) {
    if ((ret = real_read(fd, buf, count)) == -1) {
      if (errno == EWOULDBLOCK) {
        sem_wait(&FileSemaphore[fd]);
        continue;
      }
    }
    break;
  }
  return(ret);
}
```

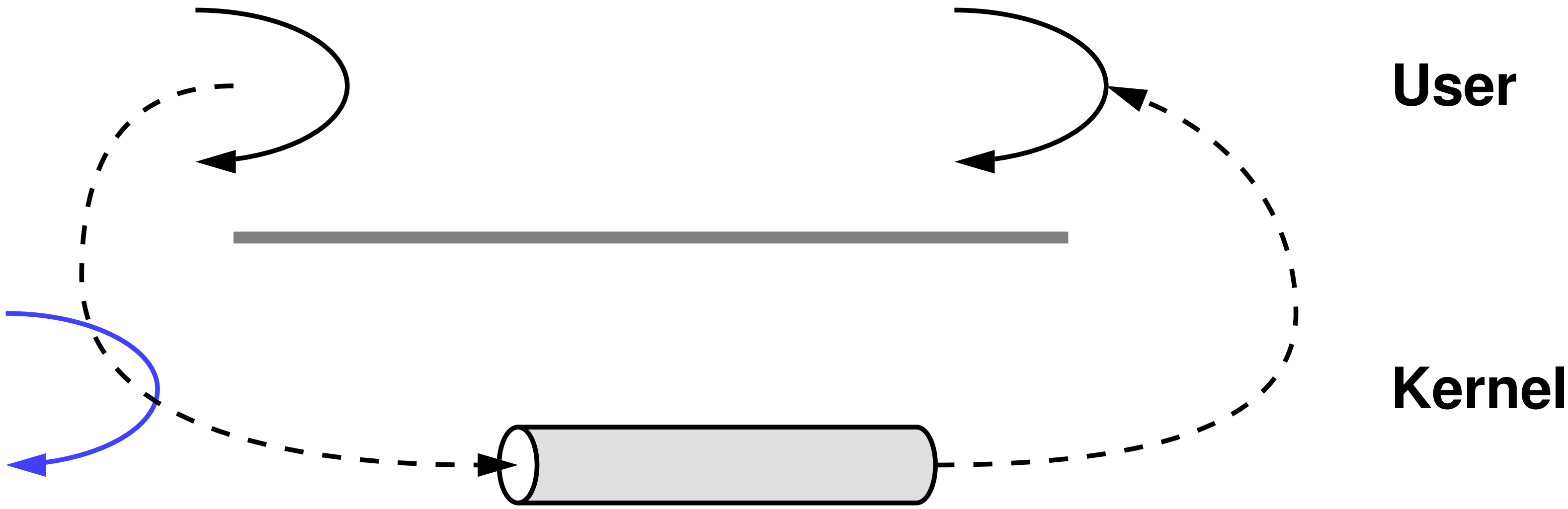
- **One semaphore for each open file**
  - **perhaps a signal handler will invoke `sem_post()` to when data is ready to be read**
- **Major drawback**
  - **only works for some I/O objects - not a general solution**

Two-Level Model: Multiple Kernel Threads



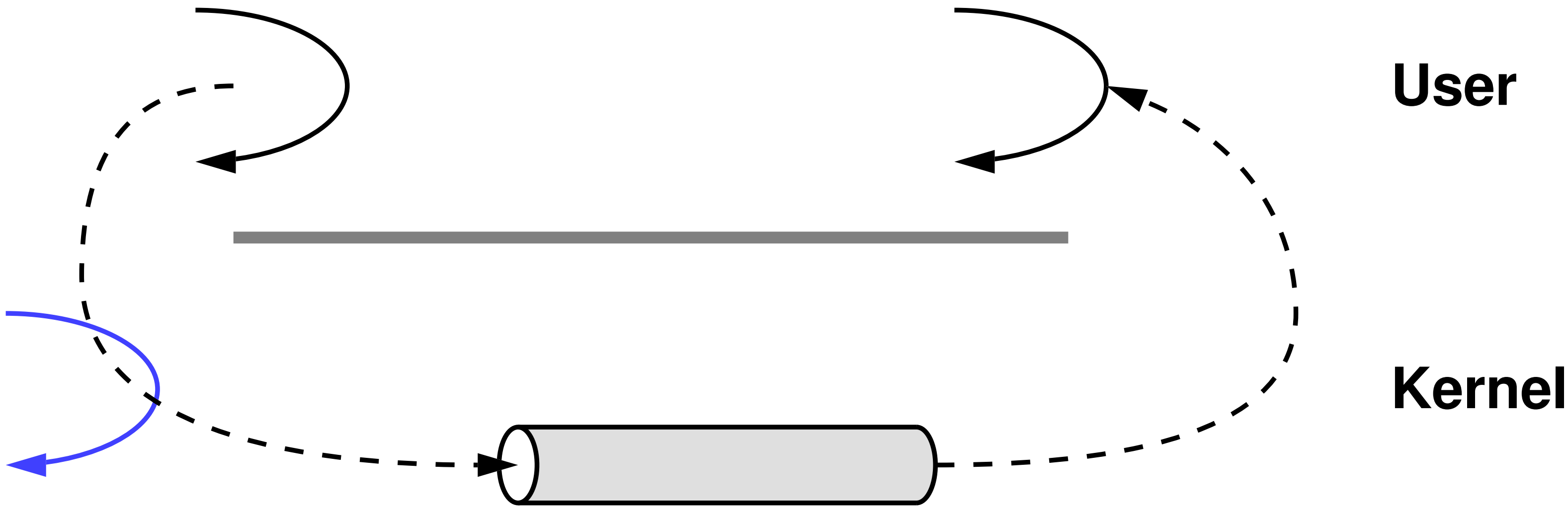
- This is called the ***M-to-N model***
- **Implementation is similar to the two-level model with a single kernel thread**
  - **no system calls (for thread-related APIs)**
  - **if we don't have enough kernel threads per user process, we end up having the same problem with the N-to-1 model**

Deadlock



- Ex: two threads are communicating using a pipe (this is essentially a kernel implementation of the producer-consmer problem)
  - first user thread writes to a full pipe and get blocked in the kernel
    - first thread just happened to use the last kernel thread
    - 2nd thread wants to read the pipe to unblock the first thread, but cannot because no kernel thread left

Deadlock



- Solaris solution: automatically create a new kernel thread
  - an obvious solution

Recap - Problems

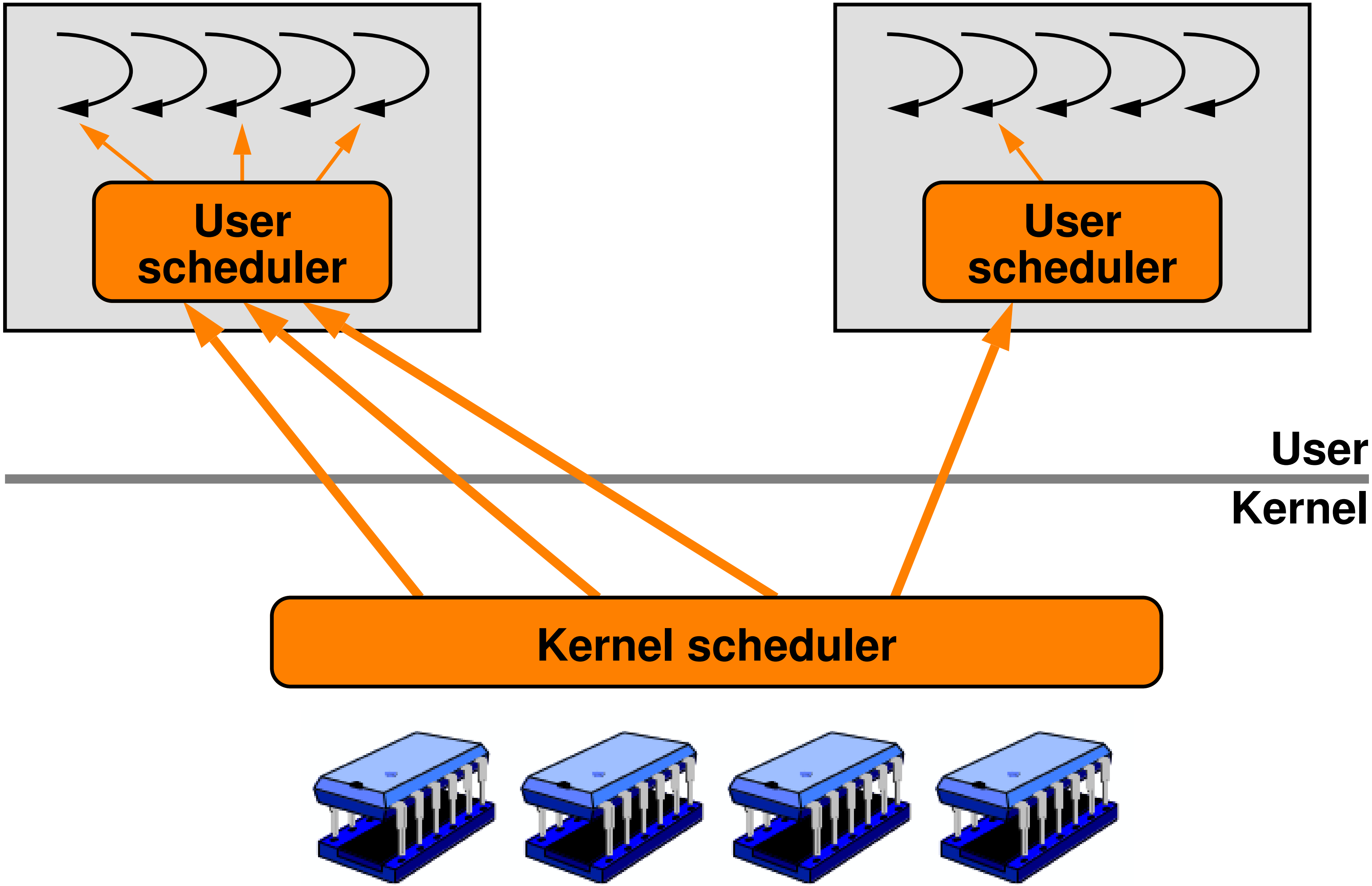
- Two-level model does not solve the I/O blocking problem
  - if there are N kernel threads and if N user threads are blocked in I/O
    - no other user threads can make progress

- Another problem: ***Priority Inversion***
  - user-level thread schedulers are not aware of the kernel-level thread scheduler
    - it may know the number of kernel threads
  - how can the user-level scheduler talk to the kernel-level scheduler?
    - people have tried this, but it's complicated
  - it's possible to have a higher priority user thread scheduled on a lower priority kernel thread and vice versa

Scheduler Activations Model

- The scheduler activations model is radically different from the other models
  - in other models, we think of the kernel as providing some kernel thread contexts
    - then multiplexing these contexts on processors using the kernel's scheduler
  - in scheduler activations model, we divvy up processors to processes, and processes determine which threads get to use these processors
    - the kernel should supply however many kernel contexts it finds necessary

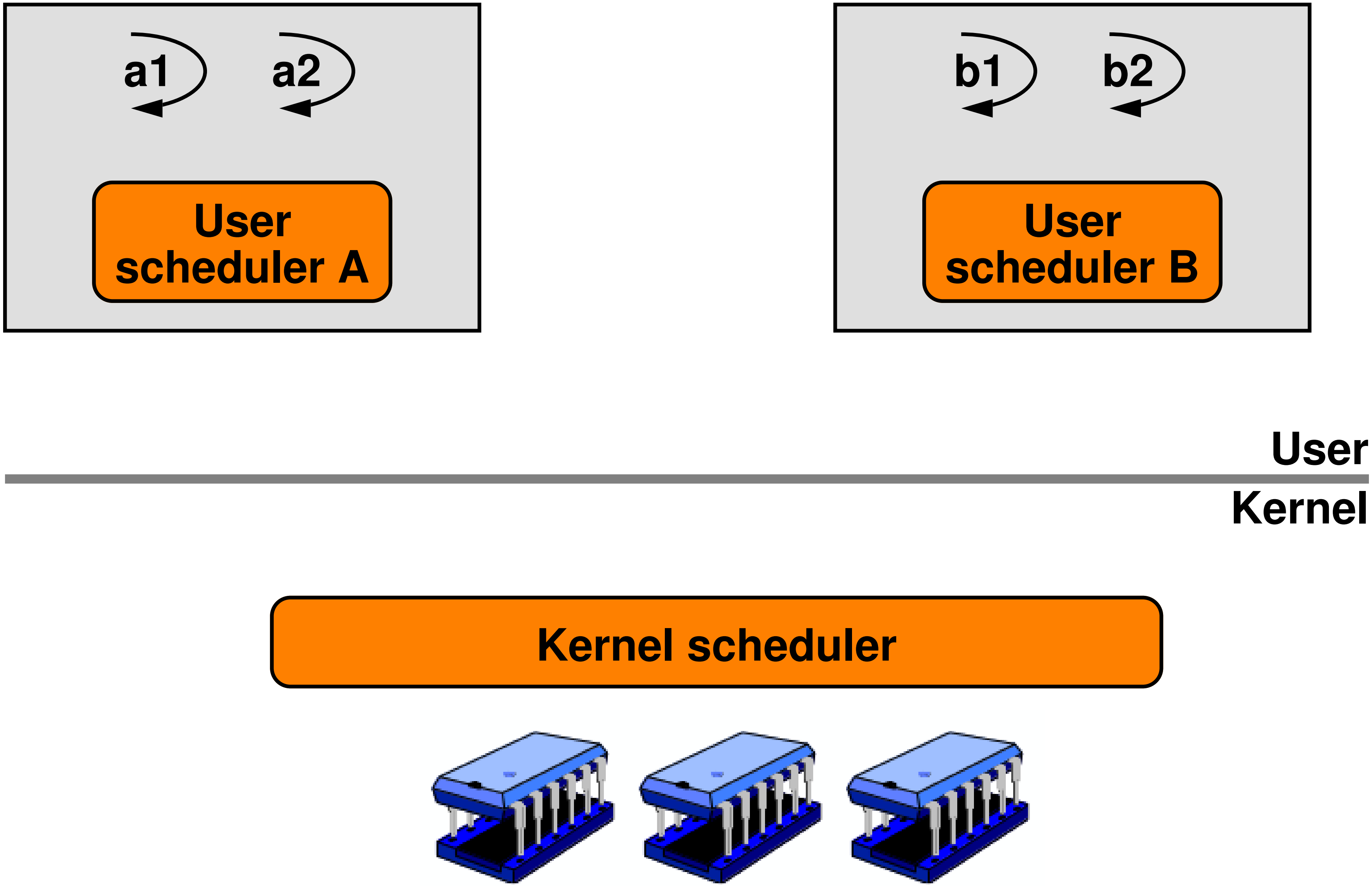
Scheduler Activations Model Example



Scheduler Activations Model Example

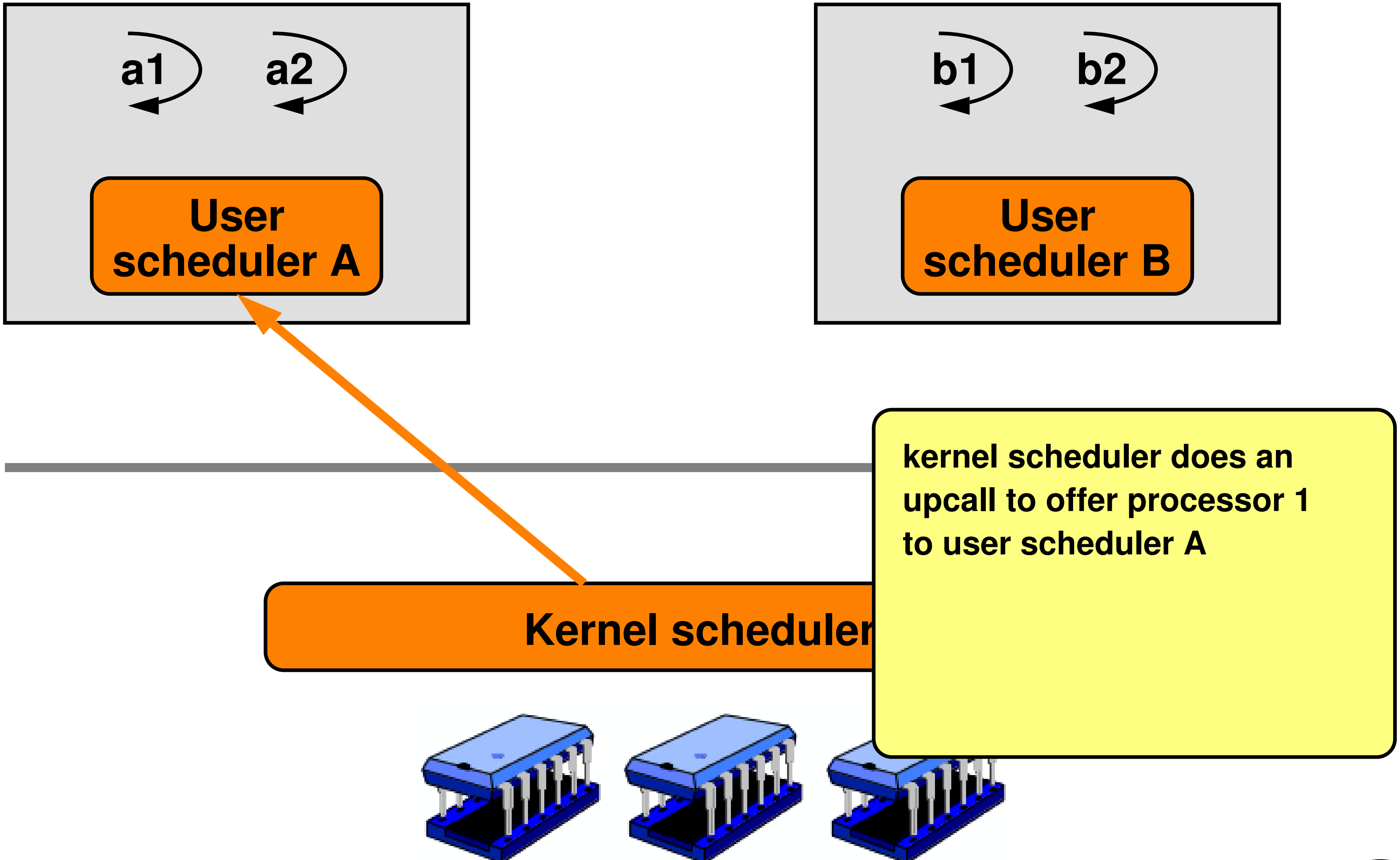
- Let's say a process starts up running a single thread
  - kernel scheduler assigns a processor to the process
  - if the thread blocks, the process gives up the processor to the kernel scheduler
- Suppose the user program creates a new thread and parallelism is desired
  - code in user-level library notifies the kernel that it needs two processors
  - when a processor becomes available, the kernel creates a new kernel context
    - the kernel places an upcall to the user-level library, effectively giving it the processor
    - the user-level library code assigns this processor to the new thread

Scheduler Activations Model Example



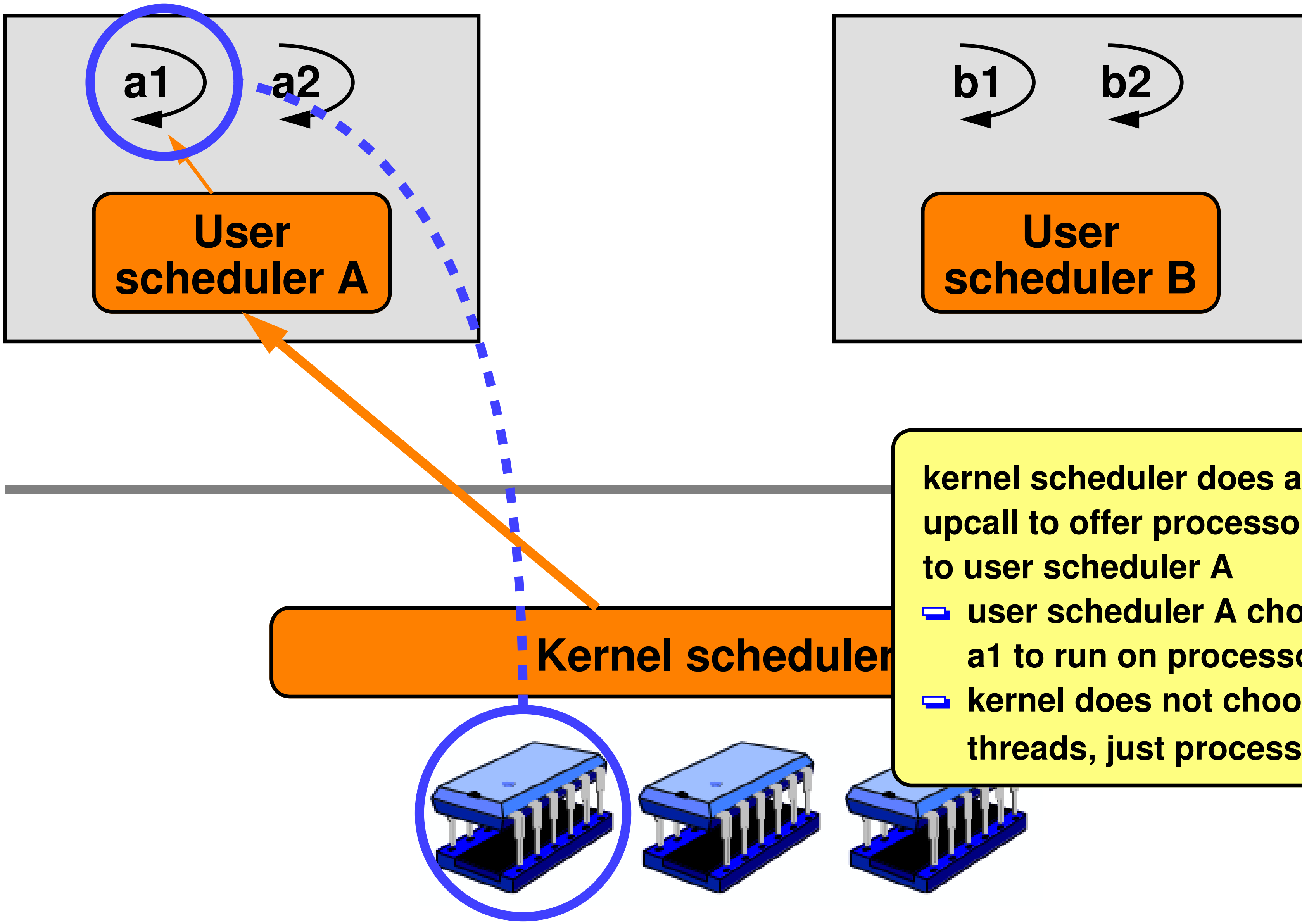
Kernel scheduler does not schedule threads

Scheduler Activations Model Example



- Kernel scheduler does not schedule threads

Scheduler Activations Model Example

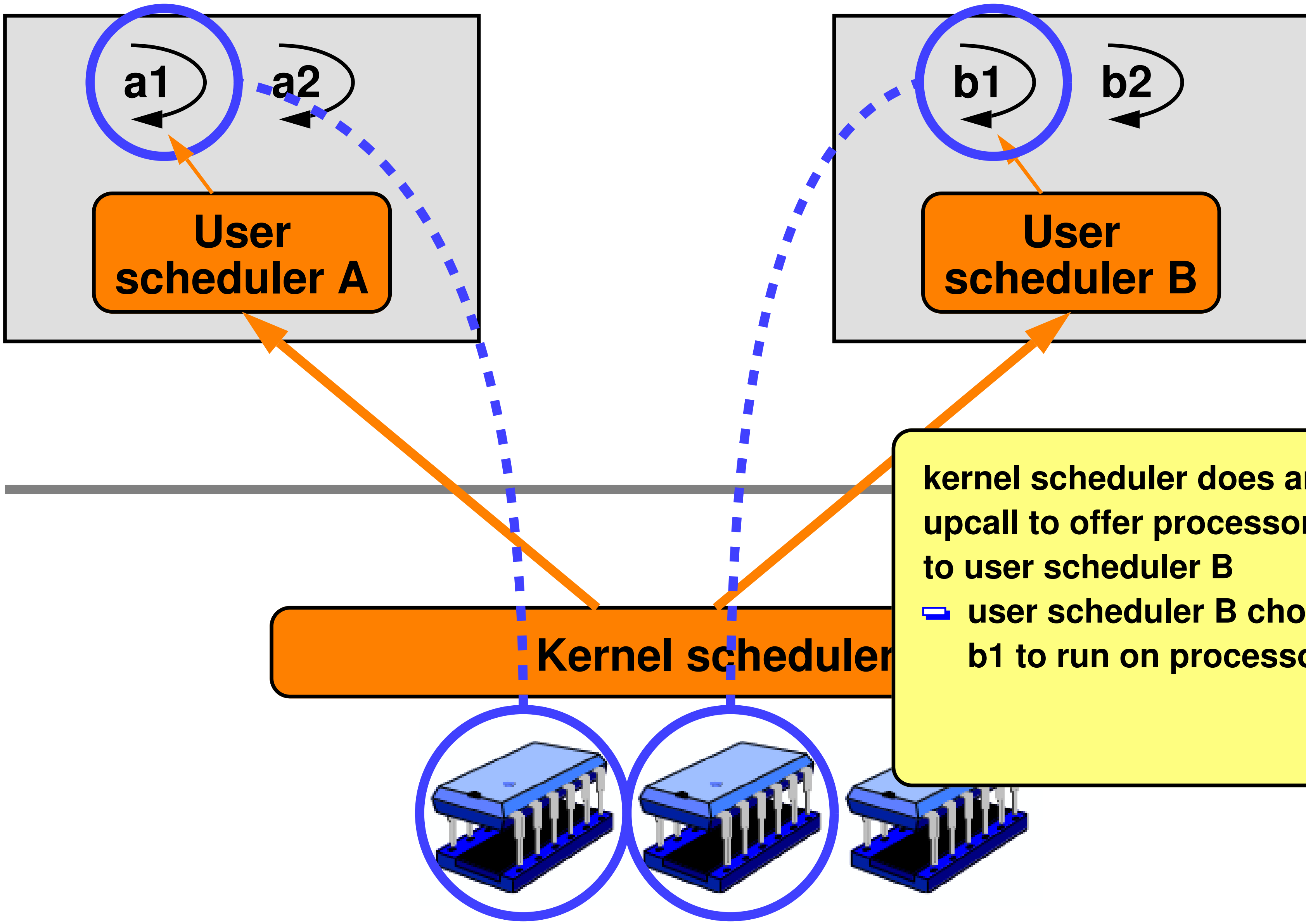


kernel scheduler does an upcall to offer processor 1 to user scheduler A

- user scheduler A chooses a1 to run on processor 1
- kernel does not choose threads, just processes

⇨ Kernel scheduler does not schedule threads

Scheduler Activations Model Example

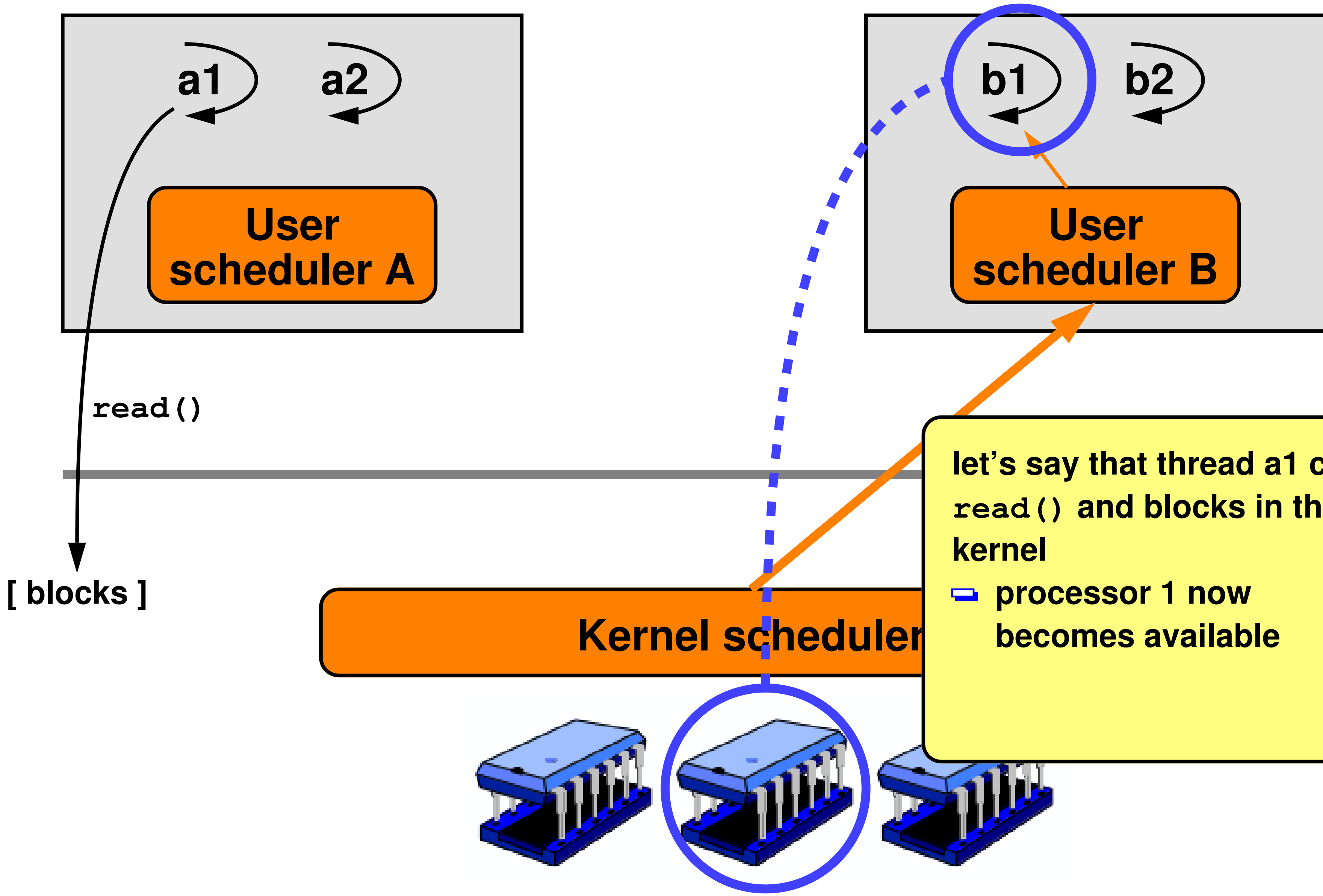


kernel scheduler does an upcall to offer processor 2 to user scheduler B

- user scheduler B chooses b1 to run on processor 2

- Kernel scheduler does not schedule threads

Scheduler Activations Model Example

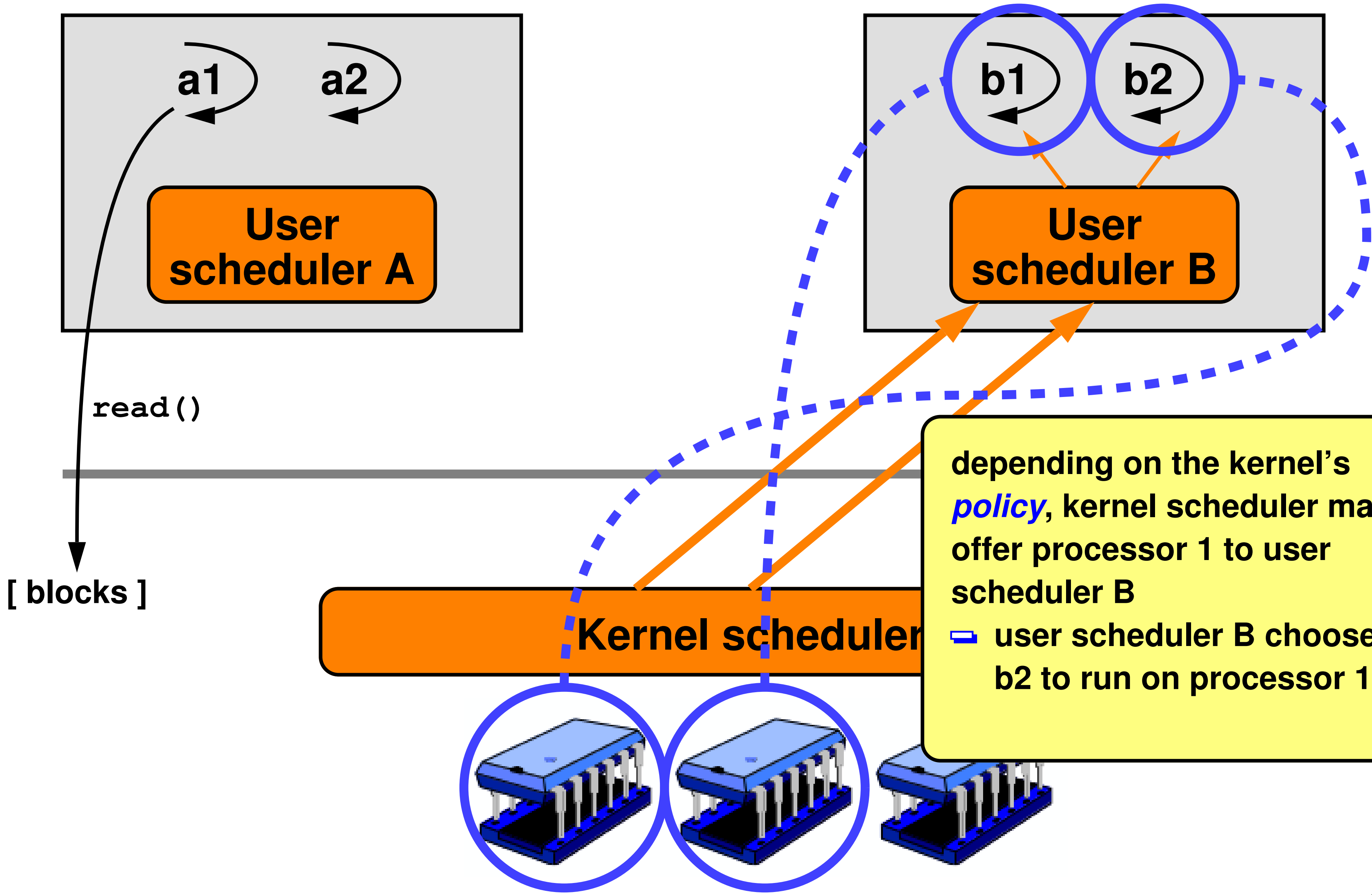


let's say that thread a1 calls `read()` and blocks in the kernel

- processor 1 now becomes available

- Kernel scheduler can have various scheduling policies

Scheduler Activations Model Example

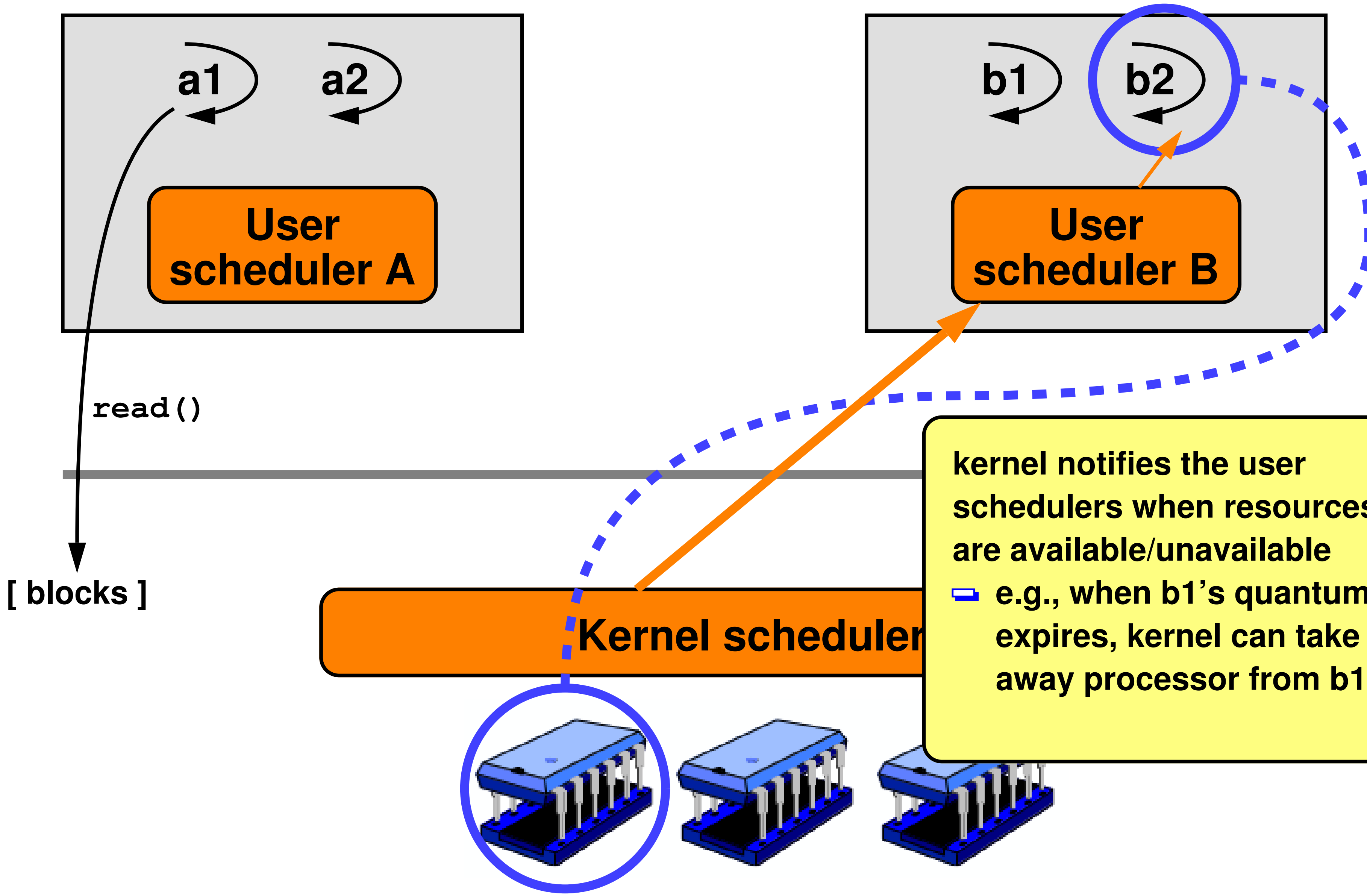


depending on the kernel's *policy*, kernel scheduler may offer processor 1 to user scheduler B

- user scheduler B chooses b2 to run on processor 1

⇨ **Kernel scheduler can have various scheduling policies**

Scheduler Activations Model Example



kernel notifies the user schedulers when resources are available/unavailable

- e.g., when b1's quantum expires, kernel can take away processor from b1

⇨ **Kernel scheduler can have various scheduling policies**

5.1 Threads Implementations

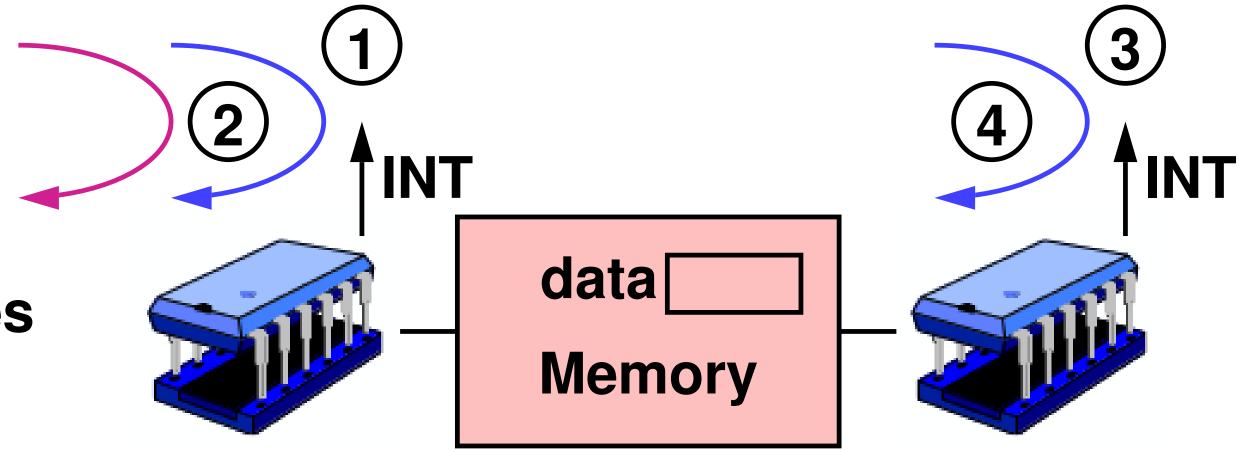
- Strategies
- *A Simple Thread Implementation*
- Multiple Processors

A Simple Threads Implementation

- Threads implementation considerations
  - data structures
  - thread switching
  - synchronization
    - how to implmement mutexes?
      - spin locks
      - sleep/blocking locks
      - futexes
    - please keep in mind that a mutex *can be* implemented in the kernel *and* in the user space

A Simple Threads Implementation

- **The challenge with implementing mutexes is that you have to ensure that they perform correctly under different kinds of concurrency**

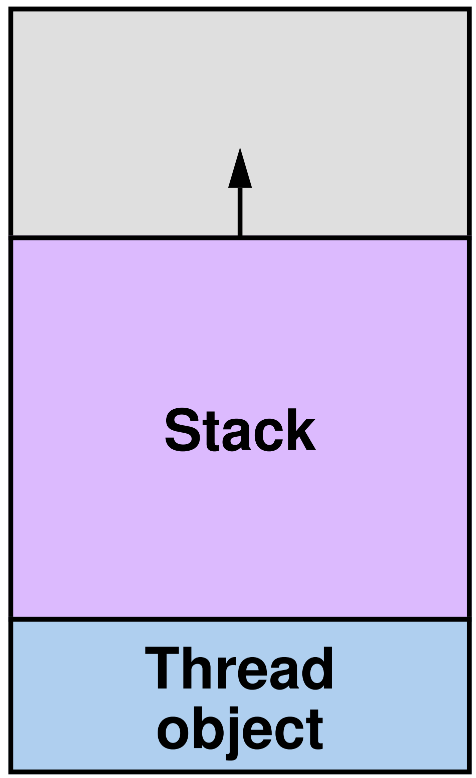


- **Asynchronous activies that may require concurrency control**
  1) **an *interrupt handler* running on the *same processor* that accesses the same data structure**
  2) ***another thread* running on the *same processor* may *preempt* this thread and accesses the same data structure**
  3) **an *interrupt handler* running on *another processor* might access the same data structure**
  4) ***another thread* running on *another processor* might access the same data structure**

A Simple Threads Implementation

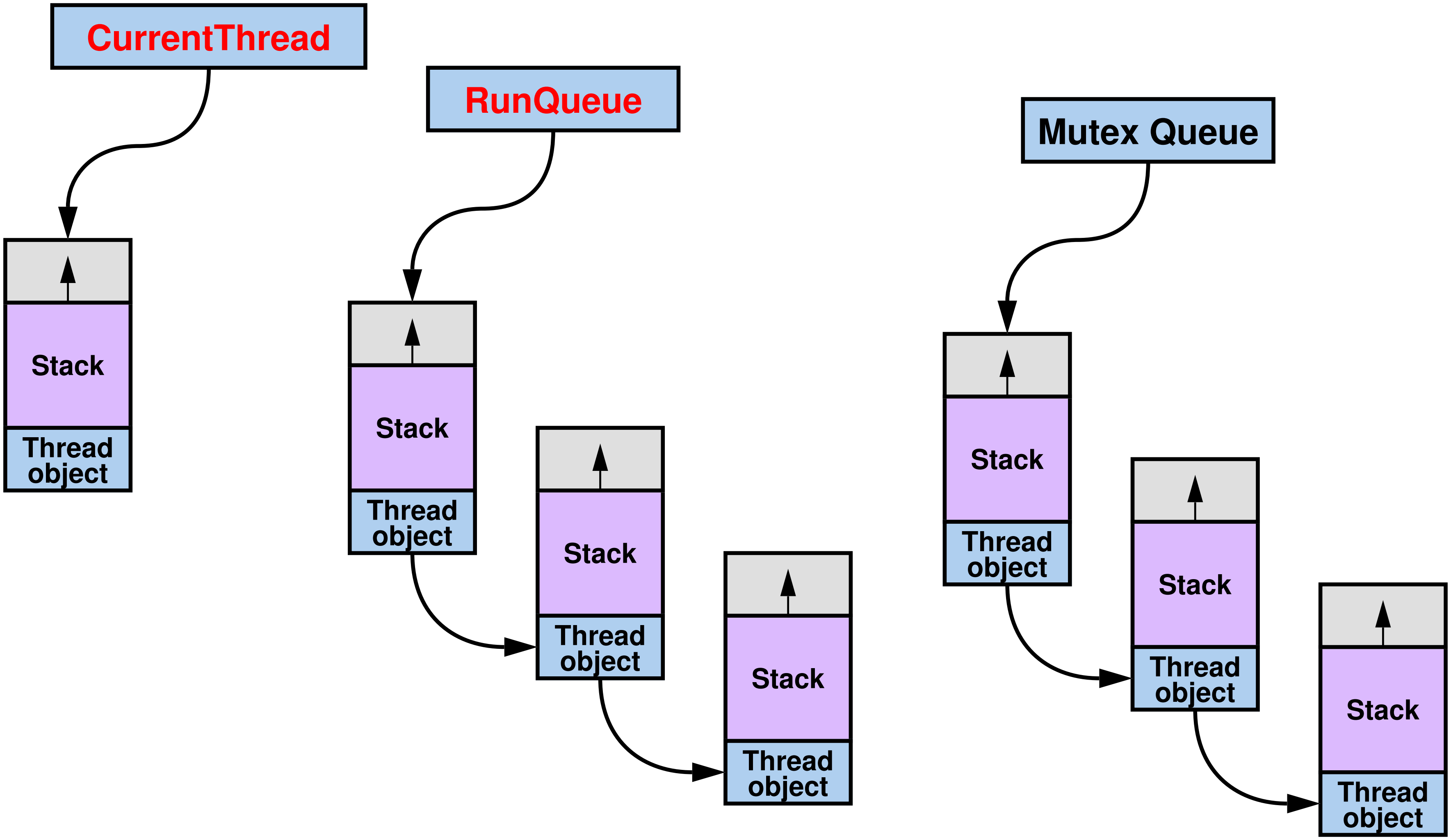
- This implementation is the basis for user-level threads package
  - "thread" can mean kernel thread or user thread
  - mutex does not need to be a kernel data structure
- *Straight-threads* implementation
  - everything happens in thread contexts
    - *no interrupt*
    - therefore, *no preemption*
  - *one processor*
  - this is like your kernel 1 with `DRIVERS=0` in `Config.mk`

Basic Representation



- We will depict a thread like this (to be more compact)
  - although we know that a thread control block is separated from a thread's stack

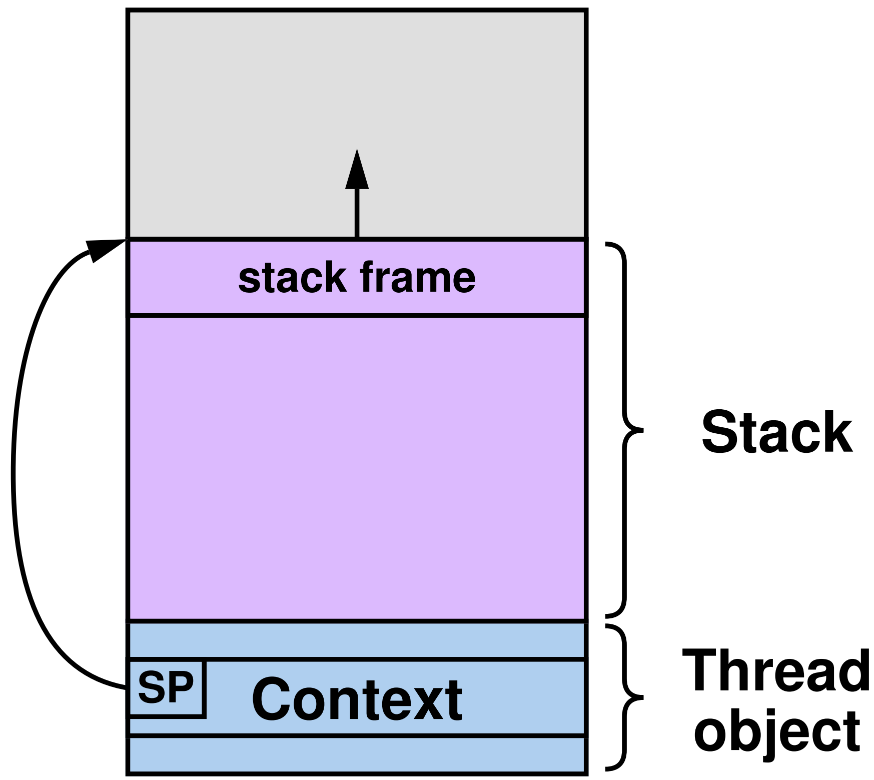
A Collection of Threads



- Each thread must be in one of these data structures
  - your kernel assignment looks like this
    - ***at any time***, you should know where your threads are

Context Pointer

- Recall from Ch 3



- if this thread is not currently running, "stack frame" corresponds to `switch()`

Straight-threads - Thread Switch

- Need a `thread_switch()` function to yield the processor

```
void thread_switch( ) {
  thread_t NextThread, OldCurrent;

  NextThread = dequeue(RunQueue);
  OldCurrent = CurrentThread;
  CurrentThread = NextThread;
  swapcontext(&OldCurrent->context,
              &NextThread->context);
  // We're now in the new thread's context
}
```

  - `switch()` in Ch 3 has a target thread argument
  - `swapcontext(old, new)` ***saves*** the caller's context into the `old` context and ***restores*** from the `new` context
  - note that the RunQueue may be empty, so this code is incomplete
  - before you get here, the current thread is queued onto somewhere else already (e.g., a mutex queue)

Straight-threads - Synchronization

- According to the textbook

```
void mutex_lock(mutex_t *m) {
  if (m->locked) {
    enqueue(m->queue, CurrentThread);
    thread_switch();
  } else
    m->locked = 1;
}

void mutex_unlock(mutex_t *m) {
  if (queue_empty(m->queue))
    m->locked = 0;
  else
    enqueue(runqueue, dequeue(m->queue));
}
```

  - `mutex_unlock()` does not seem to work becuase when it returns, the mutex can be locked and the new mutex holder is not holding the mutex
  - after further analysis, it actually does work!

Straight-threads - Synchronization

```
void mutex_lock(mutex_t *m) {
  if (m->locked) {
    enqueue(m->queue, CurrentThread);
    thread_switch();
  } else
    m->locked = 1;
}

void mutex_unlock(mutex_t *m) {
  if (queue_empty(m->queue))
    m->locked = 0;
  else
    enqueue(runqueue, dequeue(m->queue));
}
```

- Why is the code atomic?

Straight-threads - Synchronization

```
void mutex_lock(mutex_t *m) {
  if (m->locked) {
    enqueue(m->queue, CurrentThread);
    thread_switch();
  } else
    m->locked = 1;
}

void mutex_unlock(mutex_t *m) {
  if (queue_empty(m->queue))
    m->locked = 0;
  else
    enqueue(runqueue, dequeue(m->queue));
}
```

- Why is the code atomic?
  - single process and no interrupts
  - no way to preempt a thread's execution
    - a thread holds on to the processor as long as it wants, until it reliquishes processor all by itself

5.1 Threads Implementations

- Strategies
- A Simple Thread Implementation
- *Multiple Processors*

Straight-threads - Multiple Processors

- `thread_switch()` is no longer sufficient
  - it's meant for uniprocessor

```
void thread_switch( ) {
  thread_t NextThread, OldCurrent;
  NextThread = dequeue(RunQueue);
  OldCurrent = CurrentThread;
  CurrentThread = NextThread;
  swapcontext(&OldCurrent->context,
              &NextThread->context);
}
```

- Simple approach
  - run on each *processor* an *idle thread*

```
void idle_thread() {
  while(1) {
    enqueue(runqueue, CurrentThread)
    thread_switch()
  }
}
```

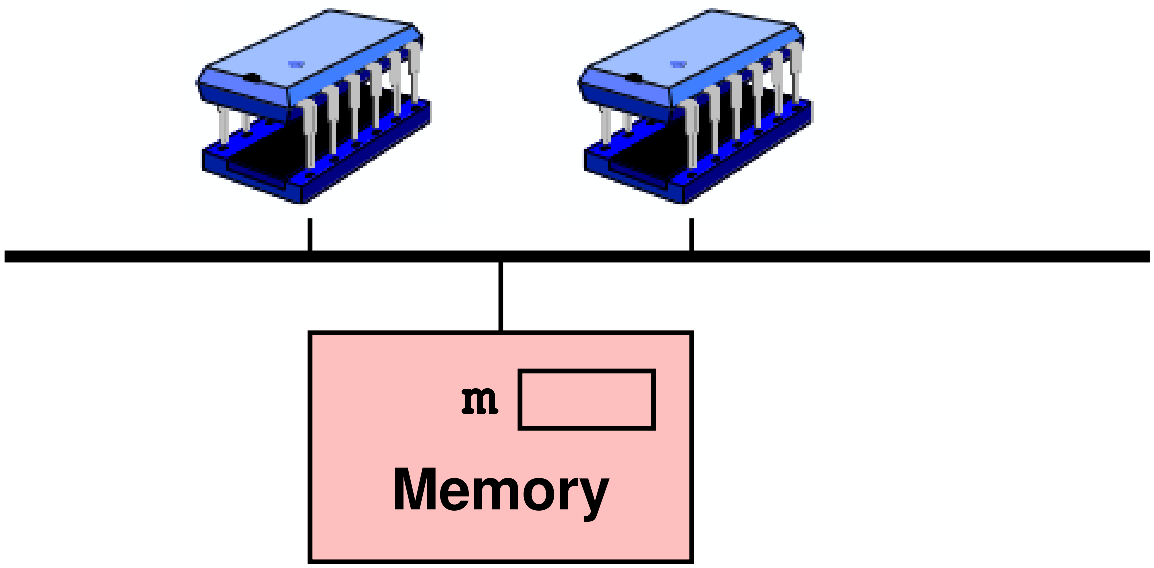
  - code is incomplete (because `thread_switch()` is incomplete, the way it was presented here)
  - this thread never blocks, so there is always something to run to avoid boundary condition
  - normal threads join the RunQueue when ready

Straight-threads - Multiple Processors

- When there are multiple processors, the difficulty lies in locking

```
if (!m->locked) {
  m->locked = 1;
}
```

  - if both threads execute the above code concurrently, in different processors, both threads think they got the lock



Hardware Support

- Compare and swap *machine instruction*

```
int CAS(int *ptr, int old, int new) {
   int tmp = *ptr;   // get the value of mutex
   if (tmp == old)   // if it equals to old
      *ptr = new;    //    set it to new
   return tmp;       // return old
}
```

- often implemented as a machine-level instruction
  - must execute atomically
    - how?

Hardware Support

- Compare and swap *machine instruction*

```
int CAS(int *ptr, int old, int new) {
    int tmp = *ptr;   // get the value of mutex
    if (tmp == old)   // if it equals to old
        *ptr = new;   //    set it to new
    return tmp;       // return old
}
```

  - e.g., assume mutex is *unlocked*, call `CAS(&lock, 0, 1)`
    - mutex is represented as a bit, 0 if unlocked, 1 if locked

A[0..31]

D[0..31]

RD

WR

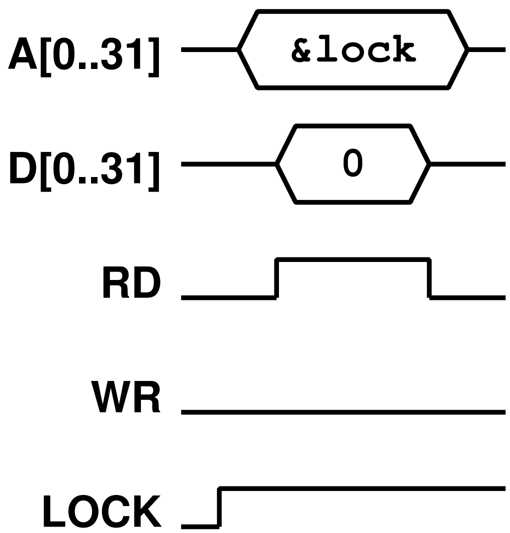
LOCK

Hardware Support

- Compare and swap *machine instruction*

```
int CAS(int *ptr, int old, int new) {
    int tmp = *ptr;   // get the value of mutex
    if (tmp == old)   // if it equals to old
       *ptr = new;    //    set it to new
    return tmp;       // return old
}
```

- e.g., assume mutex is *unlocked*, call `CAS(&lock, 0, 1)`
  - mutex is represented as a bit, 0 if unlocked, 1 if locked

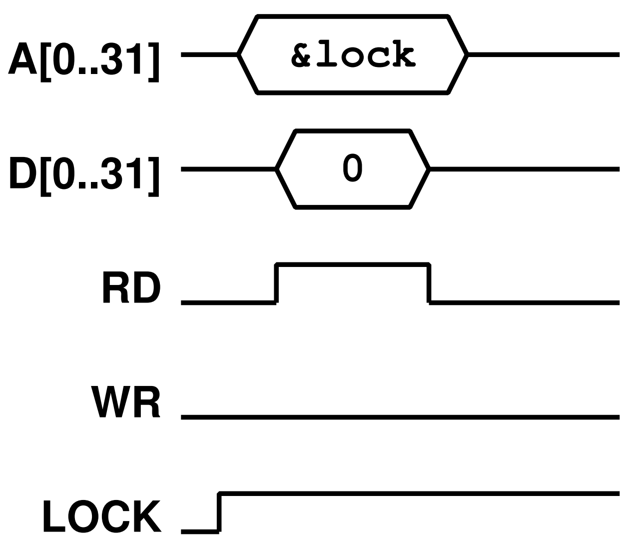


Hardware Support

- Compare and swap *machine instruction*

```
int CAS(int *ptr, int old, int new) {
    int tmp = *ptr;   // get the value of mutex
    if (tmp == old)   // if it equals to old
       *ptr = new;    //    set it to new
    return tmp;       // return old
}
```

- e.g., assume mutex is *unlocked*, call `CAS(&lock, 0, 1)`
  - mutex is represented as a bit, 0 if unlocked, 1 if locked

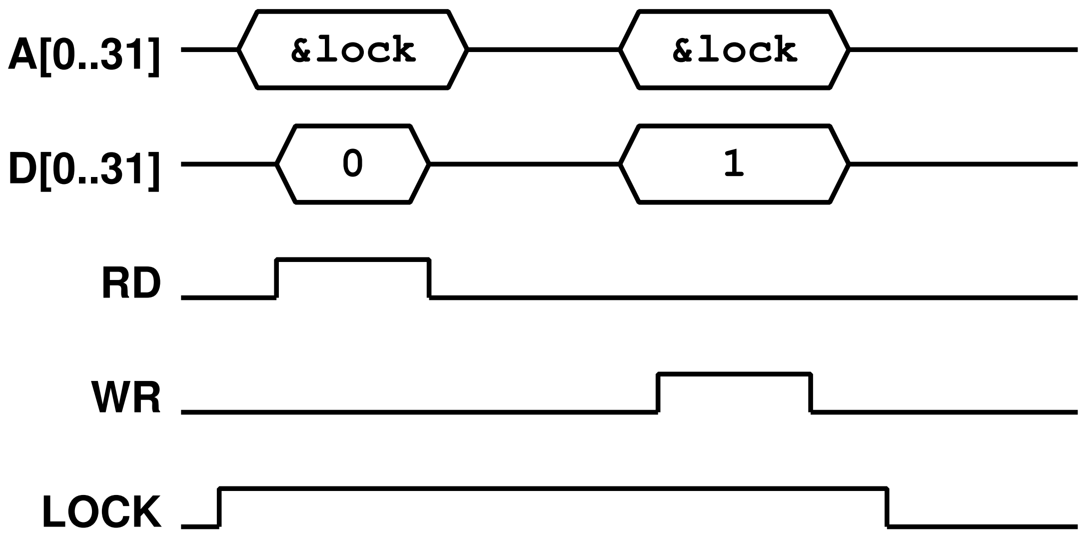


Hardware Support

- Compare and swap *machine instruction*

```
int CAS(int *ptr, int old, int new) {
    int tmp = *ptr;   // get the value of mutex
    if (tmp == old)   // if it equals to old
        *ptr = new;   //    set it to new
    return tmp;       // return old
}
```

  - e.g., assume mutex is *unlocked*, call `CAS(&lock, 0, 1)`
    - mutex is represented as a bit, 0 if unlocked, 1 if locked

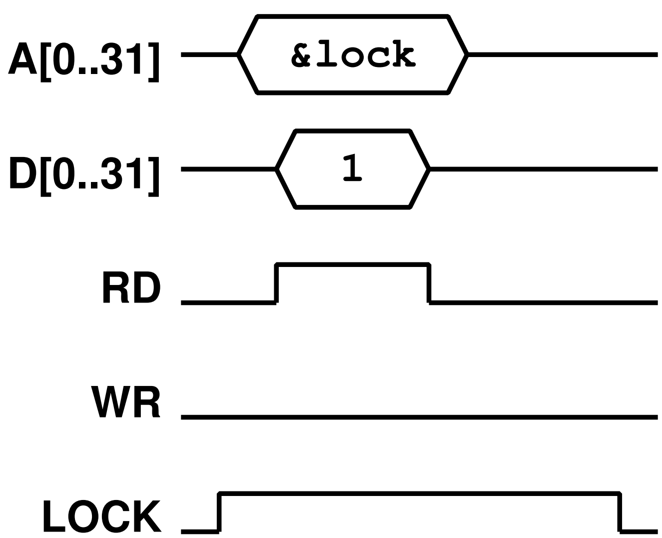


Hardware Support

- Compare and swap *machine instruction*

```
int CAS(int *ptr, int old, int new) {
   int tmp = *ptr;   // get the value of mutex
   if (tmp == old)   // if it equals to old
      *ptr = new;    //    set it to new
   return tmp;       // return old
}
```

- e.g., assume mutex is *locked*, call `CAS(&lock, 0, 1)`
  - mutex is represented as a bit, 0 if unlocked, 1 if locked



Hardware Support

- Compare and swap *machine instruction*

```
int CAS(int *ptr, int old, int new) {
   int tmp = *ptr;   // get the value of mutex
   if (tmp == old)   // if it equals to old
      *ptr = new;    //   set it to new
   return tmp;       // return old
}
```

  - often implemented as a machine-level instruction
    - must execute atomically

- Spin lock
  - mutex is represented as a bit, 0 if unlocked, 1 if locked

Spin Lock

Naive spin lock

```
void spin_lock(int *mutex) {
   while(CAS(mutex, 0, 1)) // textbook is wrong
      ;
}

void spin_unlock(int *mutex) {
   *mutex = 0;
}
```

Better spin lock

```
void spin_lock(int *mutex) {
  while (1) {
    if (*mutex == 0) {
      // the mutex was at least momentarily unlocked
      if (!CAS(mutex, 0, 1))
        break;  // we have locked the mutex
      // some other thread beat us to it, try again
    }
  }
}
```

Blocking Locks

- Spin locks are wasteful
  - processor time wasted waiting for the lock to be released
  - barely acceptable if locks are held only briefly
- A better approach is to have a blocking lock
  - threads wait by having their execution suspended
  - a thread much yield the processor and join a queue of waiting threads
    - later on, get resumed explicitly

Blocking Locks

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

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

- This code only works on a uniprocessor

Blocking Locks

1,2

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

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

- On a multiprocessor, it may not work
  - threads 1 and 2 can both think they've got the lock

Blocking Locks

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

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

- **On a multiprocessor, it may not work**
  - **thread 2 holds the mutex and wait queue is empty and thread 1 tries to lock the mutex at the same time thread 2 is releasing the mutex**
  - **thread 1 may wait forever**

Working Blocking Locks (?)

```
void blocking_lock(mutex_t *m) {
  spin_lock(m->spinlock); // okay to spin here
  if (m->holder != 0) {
    enqueue(m->wait_queue, CurrentThread);
    thread_switch();
  } else {
    m->holder = CurrentThread;
  }
  spin_unlock(m->spinlock);
}

void blocking_unlock(mutex_t *m) {
  spin_lock(m->spinlock); // okay to spin here
  if (queue_empty(m->wait_queue)) {
    m->holder = 0;
  } else {
    m->holder = dequeue(m->wait_queue);
    enqueue(RunQueue, m->holder);
  }
  spin_unlock(m->spinlock);
}
```

- **Will deadlock because of `thread_switch()`**

Working Blocking Locks (?)

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

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

- Has a different problem

Working Blocking Locks (?)

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

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

Thread 2 can move thread 1 to another processor! (Can it?)

Working Blocking Locks (?)

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

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

- Can you do `spin_unlock()` inside `thread_switch()`?

Futexes

- ***Futex:*** **fast user-space mutex**
  - **safe, efficient kernel conditional queueing in Linux**
    - **most of the time when you try to lock a mutex, it's unlocked; so just go ahead and lock it (no system call)**
    - **if it's locked (by another thread), then a system call is required for this thread to obtain the lock**
  - **contained in it is an unsigned integer state called `value` and a queue of waiting threads**

- **Two *system calls* are provided to support futexes**

```
futex_wait(futex_t *futex, int val) {
  if (futex->val == val)
    sleep();
}

futex_wake(futex_t *futex) {
  // wake up one thread from wait queue if
  //     there is any
  ...
}
```

Ancillary Functions

- Add 1 to `*val`, return its original value

```
unsigned int atomic_inc(unsigned int *val) {
  // performed atomically
  return((*val)++); // textbook is wrong
}
```

- Subtract 1 to `*val`, return its original value

```
unsigned int atomic_dec(unsigned int *val) {
  // performed atomically
  return((*val)--); // textbook is wrong
}
```

- Just like `CAS()`, both functions return the *previous* lock value

Attempt 1

- `futex->val`
  - 0 means unlocked; otherwise, locked

```
void lock(futex_t *futex) {
  unsigned 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);
}
```

- Problem with `unlock()`
  - slow because `futex_wake()` is a system call
- Problem with `lock()`
  - threads run in lock steps in a multiprocessor environment!
  - `futex->val` may wrap-around

Attempt 2

- `futex->val` can only take on values of 0, 1, and 2
  - 0 means unlocked
  - 1 means locked but no waiting thread
  - 2 means locked with the possibility of waiting threads

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

textbook is wrong

Thread Synchronization Summary

- Spin locks
  - used if the duration of waiting is expected to be small
    - as in the case at the beginning of `blocking_lock()`
- Sleep (or blocking) locks
  - used if the duration of waiting is expected to be long
- Futexes
  - optimized version of blocking locks
- In your kernel assignmen #1, you need to implement *kernel* threads
  - very different from *user* threads
    - keep in mind that the `weenix` kernel is *non-preemptive*
    - the kernel is very powerful (and therefore, must be bug free)
  - in kernel assignmen #3, you need to implement user threads/processes (well, still one thread per process)