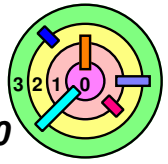
Ch 5: Processor Management

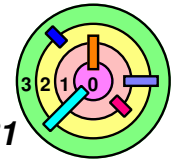
Bill Cheng

http://merlot.usc.edu/william/usc/



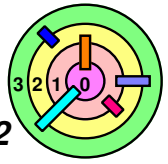
Processor Management

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



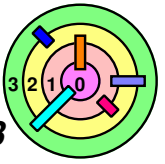
5.1 Threads Implementations

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

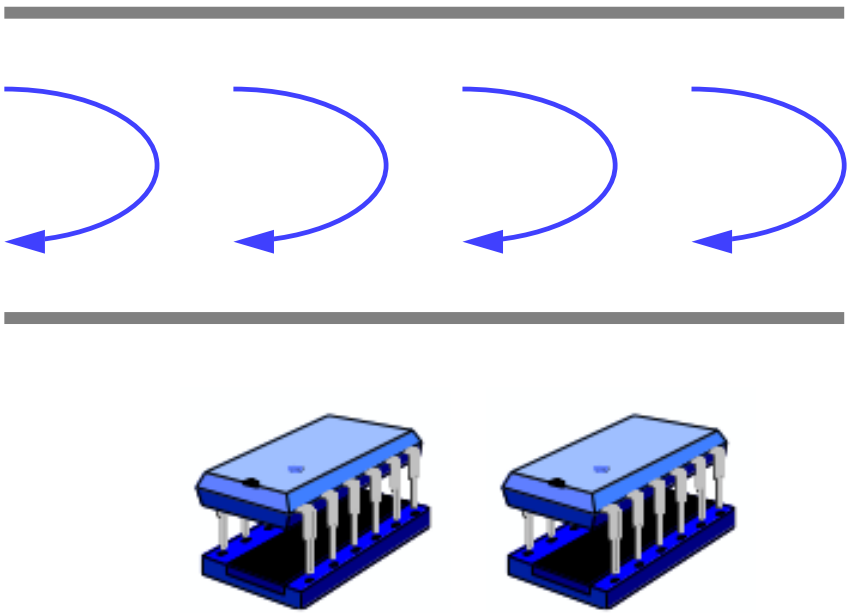


Threads Implementation

- 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 or 1 × 1 model (threads are implemented in the kernel)
    - variable-weight processes
  - two-level model (threads are implemented in user library)
    - N × 1
    - M × N
  - scheduler activations model

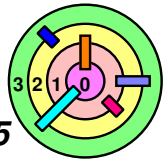


One-Level Model (1 × 1)



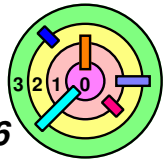
One-Level Model (1×1)

- 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



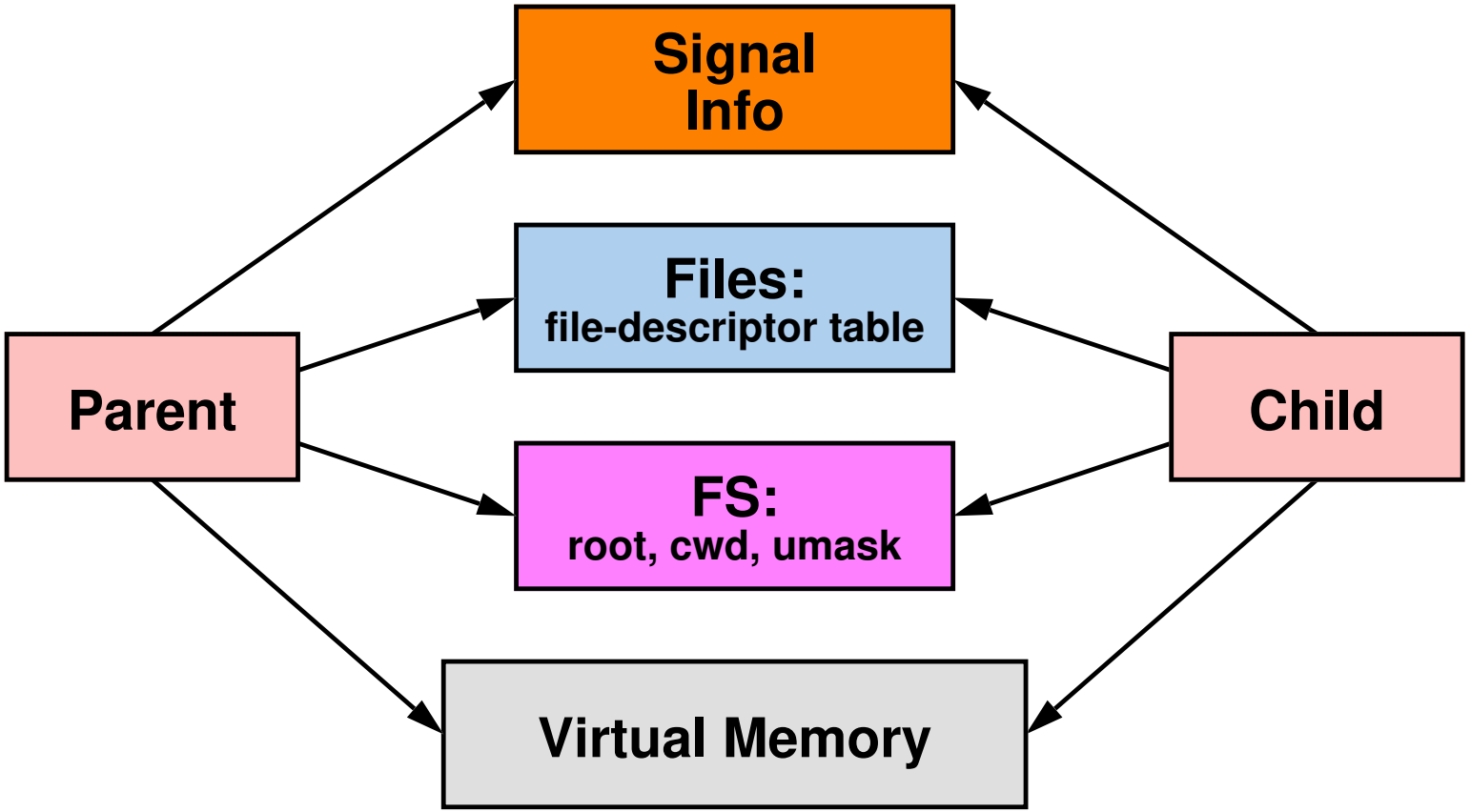
One-Level Model (1 × 1)

- 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
- Does it happen a lot that `pthread_mutex_lock` finds the mutex available?
  - think about your warmup2

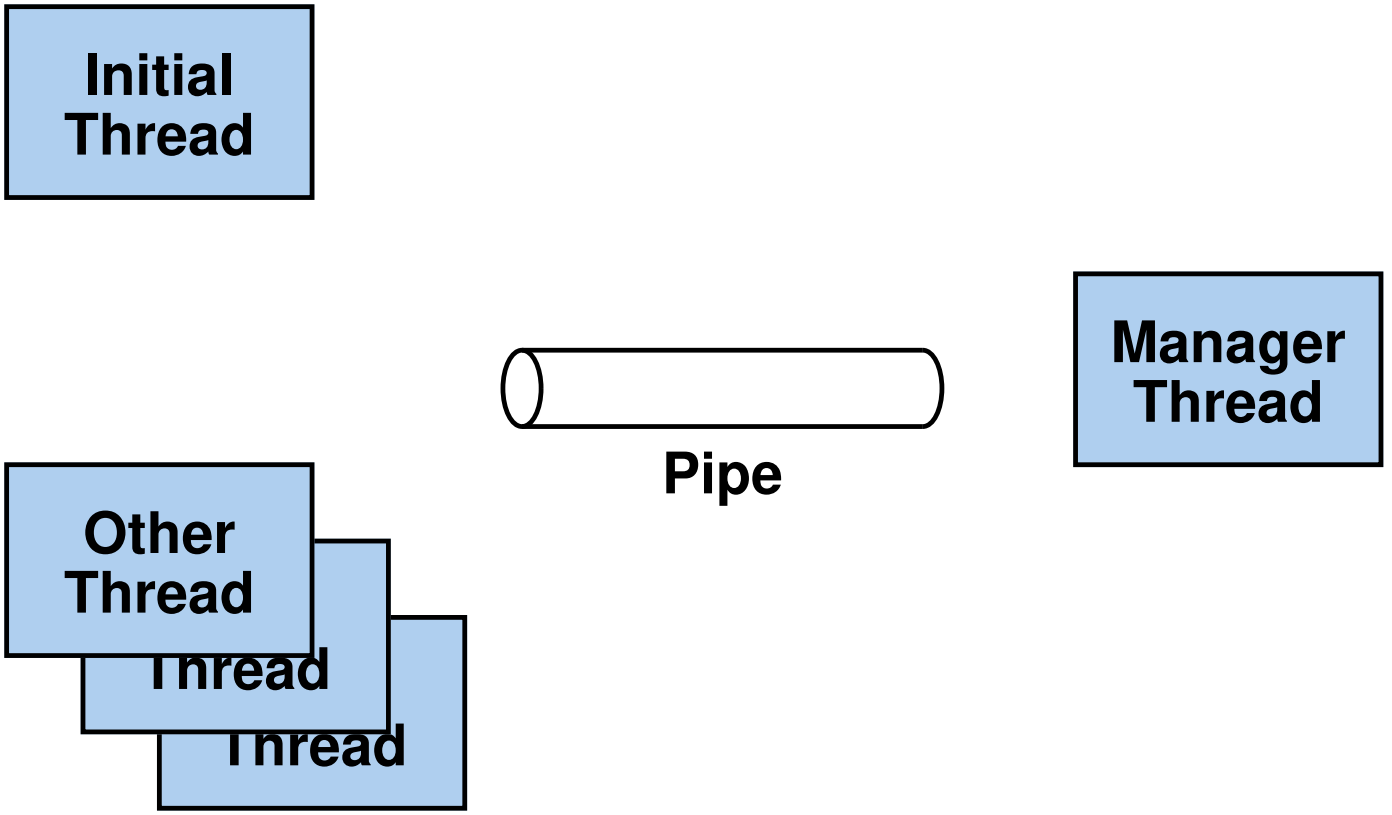


Variable-Weight Processes

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

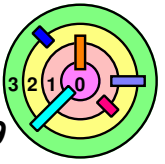


Linux Threads (pre 2.6)



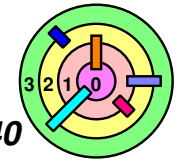
NPTL in Linux 2.6

- **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

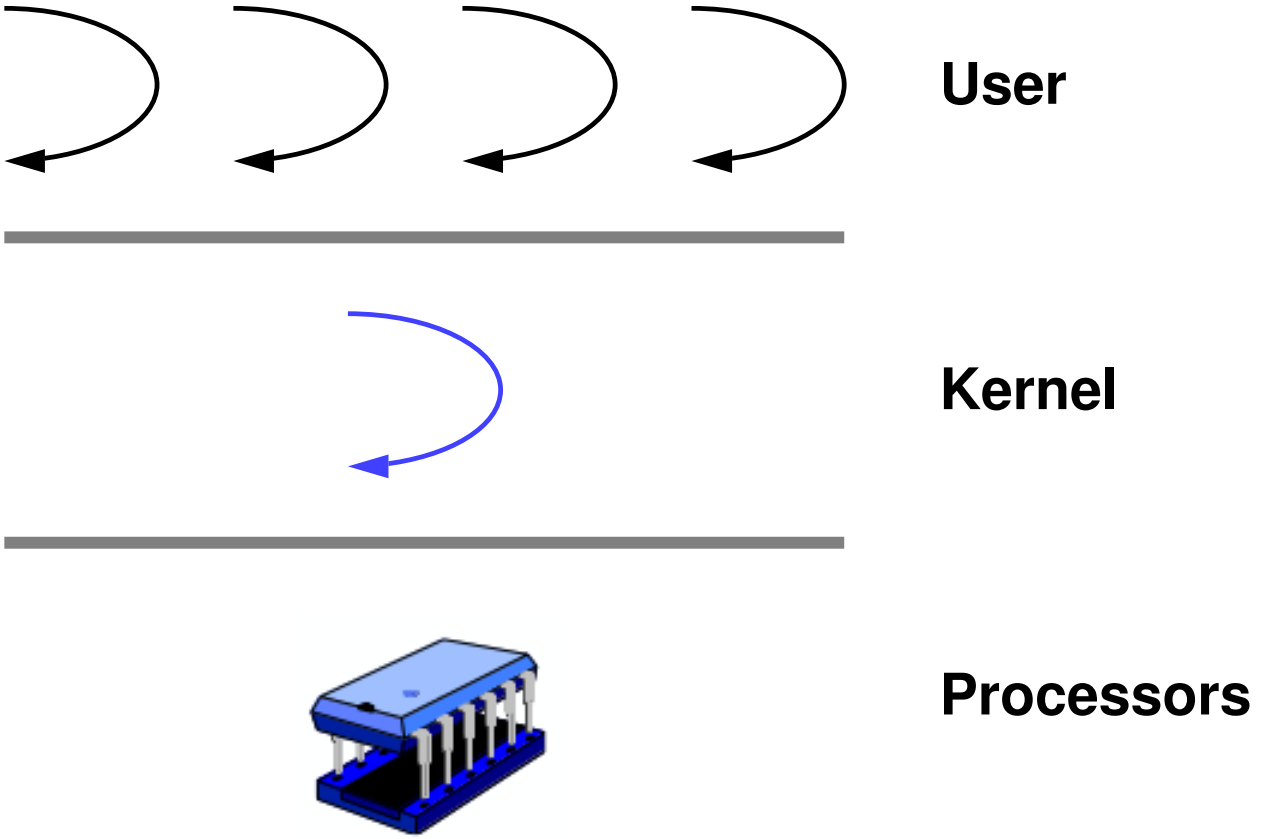


Two-Level Model

- 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)



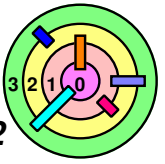
Two-Level Model - One Kernel Thread (N × 1)



- 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

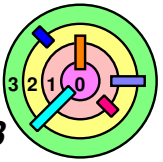
Two-Level Model - One Kernel Thread (N × 1)

- 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*
    - there are really no "kernel threads" in these systems
- 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 (N × 1)

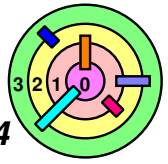
- Major advantage
  - fast, because no system calls for thread-related APIs
- Major disadvantage
  - what if a thread makes a system call (for a non-thread-related API)?
    - it gets blocked in the kernel
    - no other user thread in the process can run
  - also, there is no true parallelism within a process even when more CPUs are available



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);
}
```

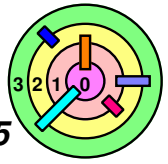
- Solution is to have a non-blocking `read()` called `real_read()`
  - `real_read()` either returns immediately with data in `buf`
  - or returns immediately with an error code in `errno`
    - `EWOULDBLOCK` means that a real `read()` would block, i.e., data is not ready to be read



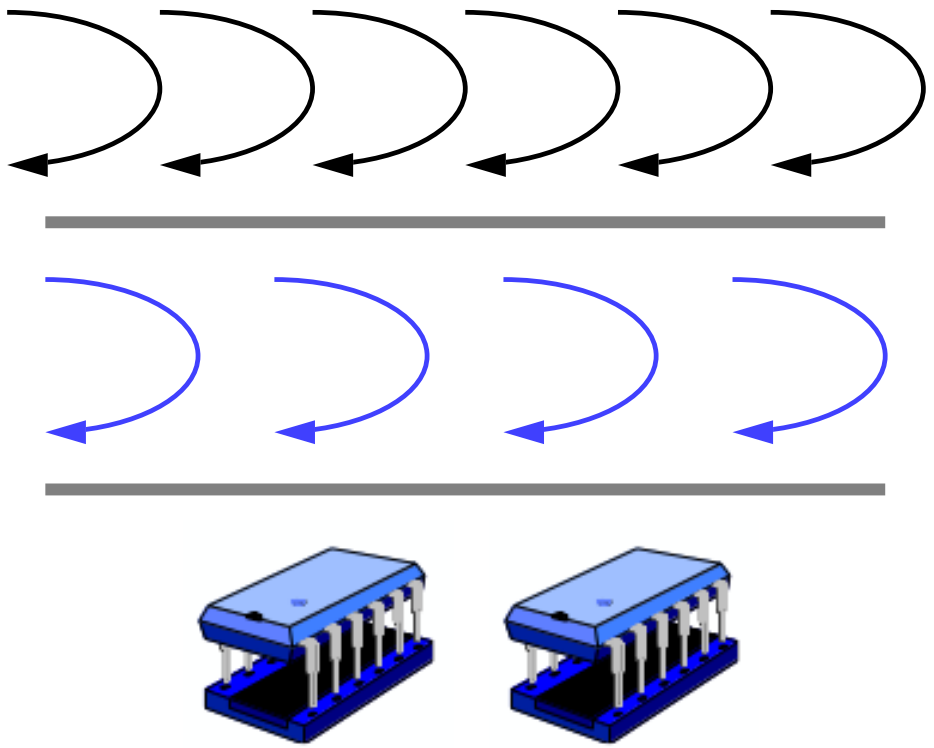
Coping ...

```
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);
}
```

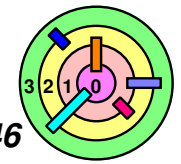
- One semaphore for each open file
  - perhaps a signal handler will invoke `sem_post()` when data is ready to be read
- Major drawback
  - only works for some I/O objects - not a general solution



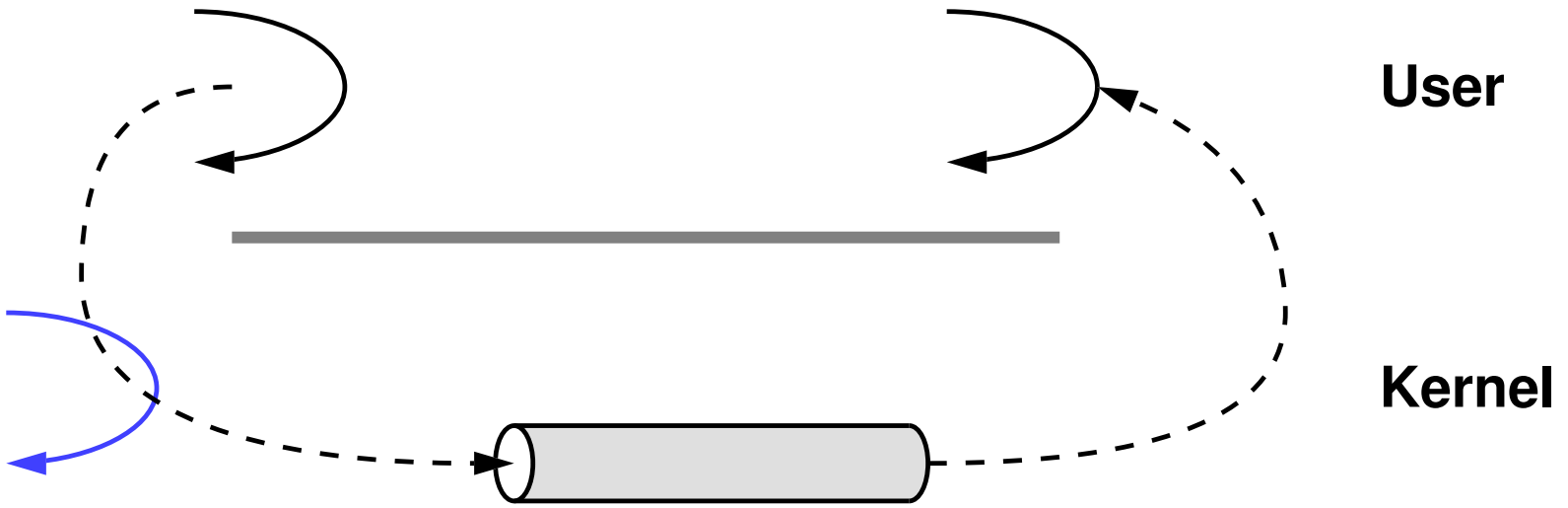
Two-Level Model: Multiple Kernel Threads (M × N)



- 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

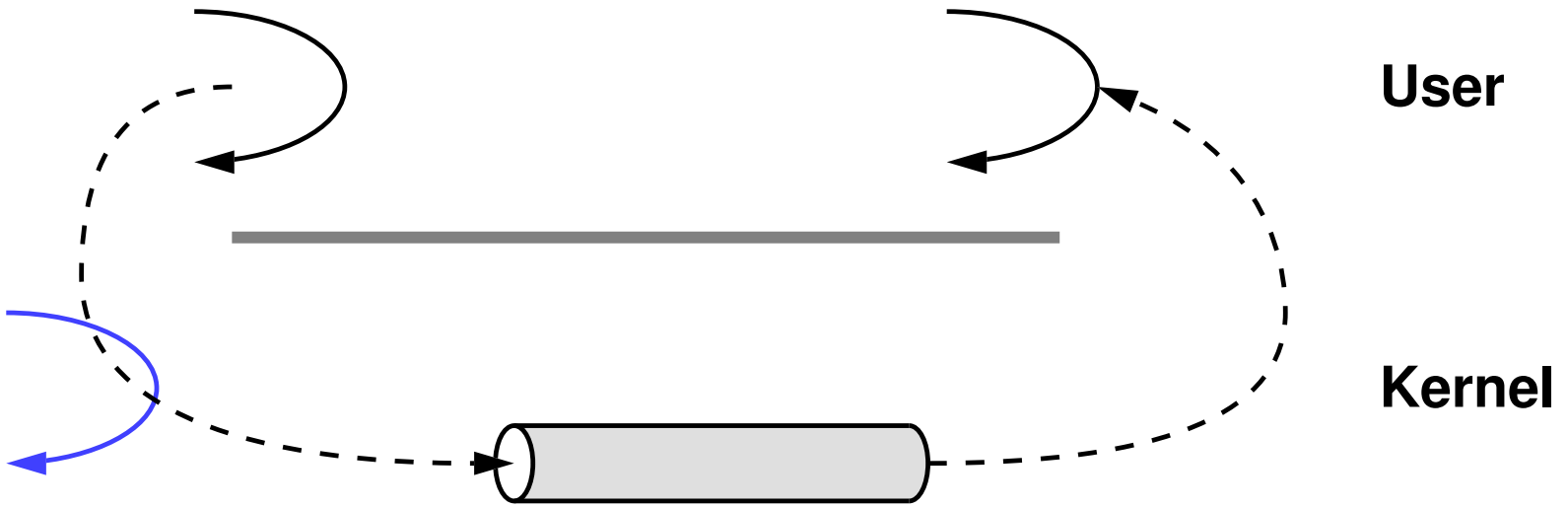


Deadlock

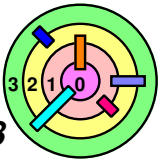


- Ex: two threads are communicating using a pipe (this is essentially a kernel implementation of the producer-consmer problem)
  - first user thread writes to a full pipe and get blocked in the kernel
    - first thread just happened to use the last kernel thread
    - 2nd thread wants to read the pipe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

Deadlock

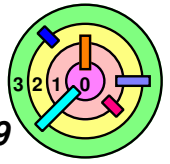


- Solaris solution: automatically create a new kernel thread
  - an obvious solution



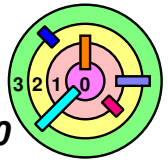
Problems With Two-level Model

- 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
    - Solaris solution basically goes back to one-level model
- 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
- Will address these problems a little later with Scheduler Activations Model



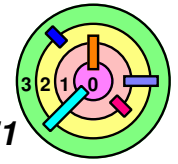
5.1 Threads Implementations

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



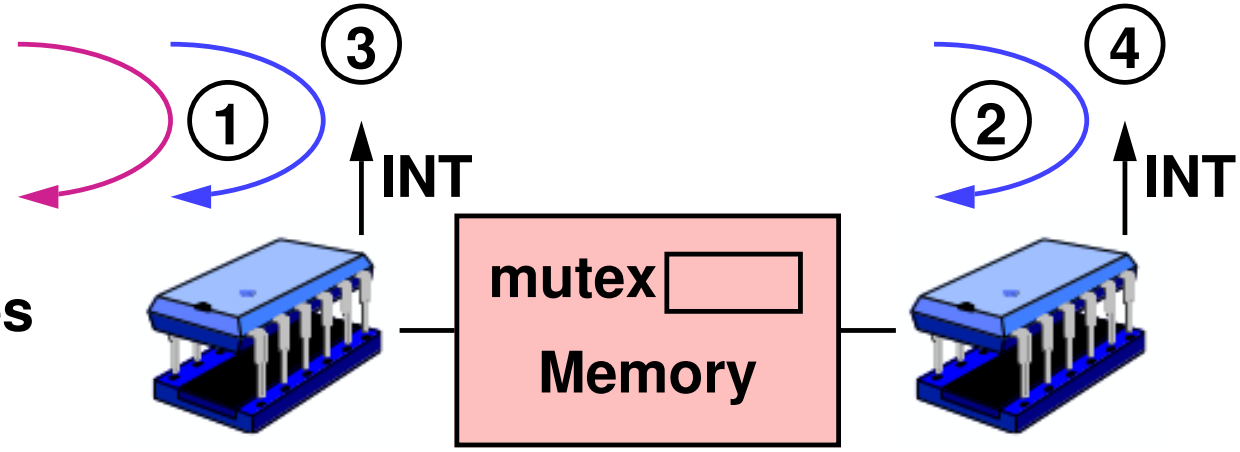
A Simple Threads Implementation

- 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/or* in the user space

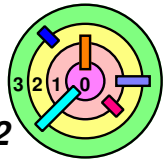


A Simple Threads Implementation

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

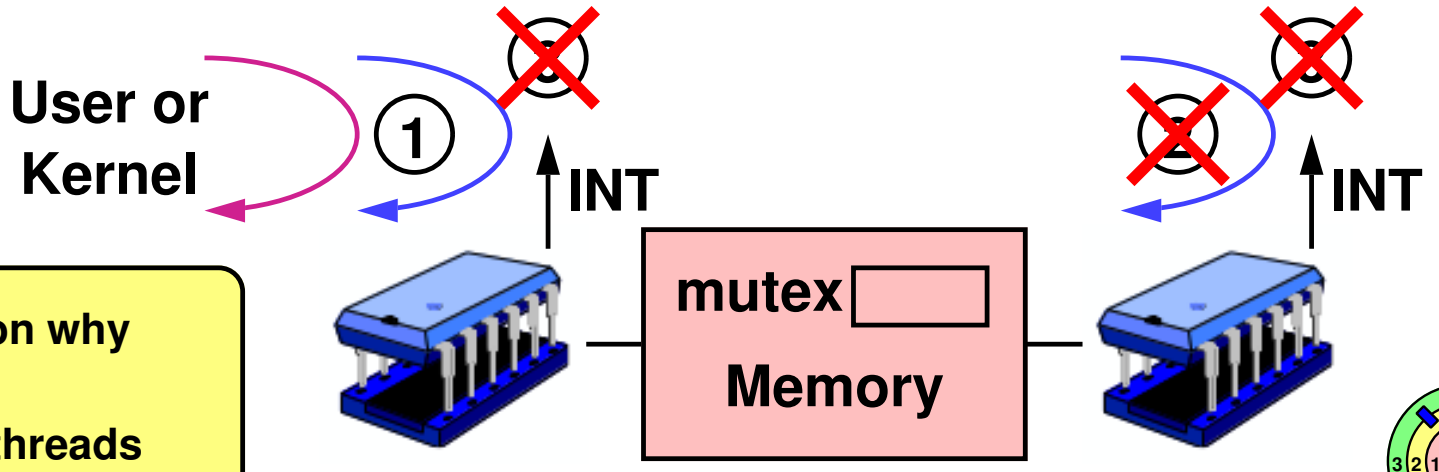


- Asynchronous activies that may require concurrency control
  1) *another thread* running on the *same processor* may *preempt* this thread and accesses the same data structure
  2) *another thread* running on *another processor* might access the same data structure
  3) an *interrupt handler* running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

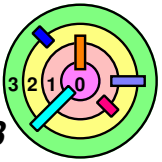


A Simple Threads Implementation

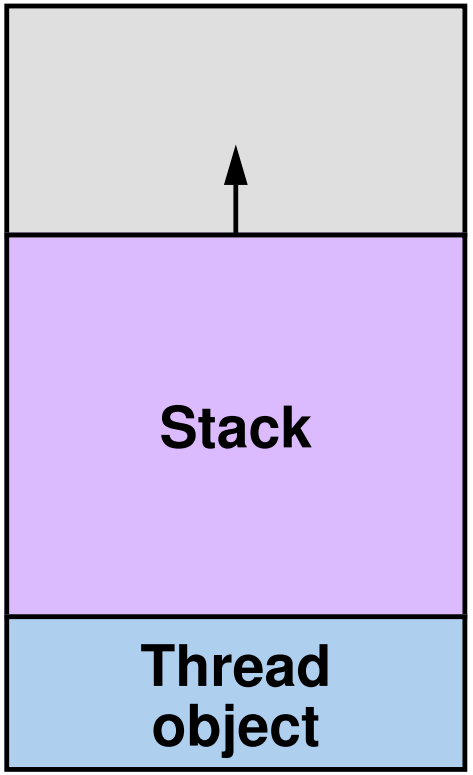
- This implementation is the basis for user-level threads package
  - "thread" can mean kernel thread or user thread
  - mutex does not need to be a kernel data structure
- *Straight-threads* implementation
  - everything happens in thread contexts
    - *no interrupt*
    - therefore, *no preemption*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



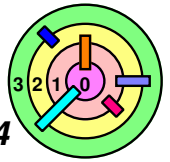
this is the real reason why kernel threads feels different from user threads



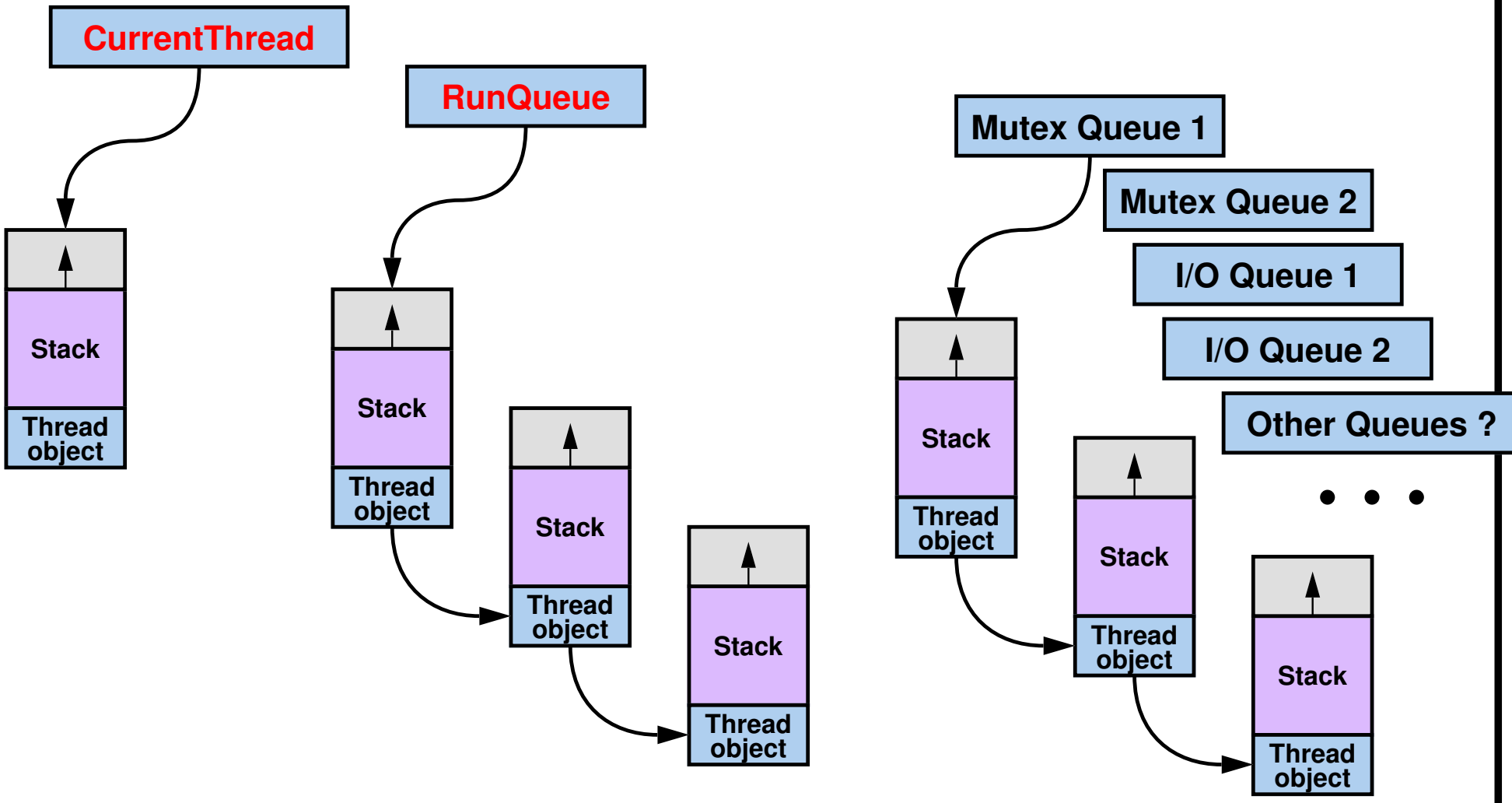
Basic Representation



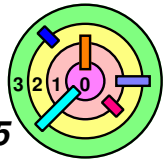
- We will depict a thread like this (to be more compact)
  - although we know that a thread control block is separated from a thread's stack



A Collection of Threads

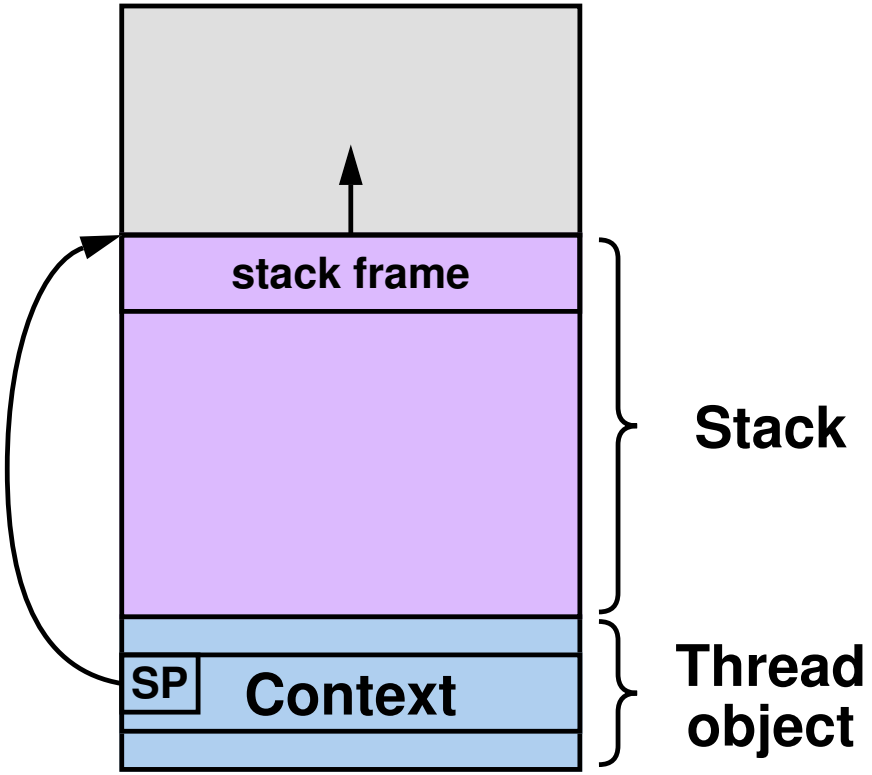


- Each thread must be in one of these data structures
  - your kernel assignment looks like this
    - *at any time*, you should know where your threads are



Context Pointer

- Recall from Ch 3



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

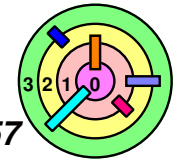
Straight-threads - Thread Switch

- 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
    - in `weenix`, this function is called `context_switch()`
  - since RunQueue may be empty, this code is incomplete
  - before you get here, the current thread is queued onto a queue somewhere else already (e.g., a mutex queue)



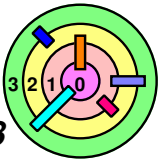
Straight-threads - Synchronization

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 still be locked and the new mutex holder does not seem to be holding the mutex
- after further analysis, it actually does work!

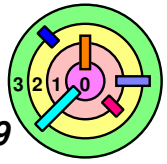


Straight-threads - Synchronization

```
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 this mutex implementation *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 this mutex implementation *atomic*?
  - *single processor* 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 relinquishes processor voluntarily