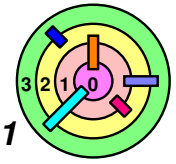
CS 350
PA3: Mutex & Condition Variable

Bill Cheng

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

Based on slides created by Kivilcim Cumbul



PA3

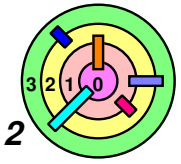
- Start with your PA2 code (no separate starter code will be provided)

```
cd
mkdir cs350/pa3
mkdir cs350/pa3/xv6-pa3-src
cd cs350/pa3/xv6-pa3-src
cp ../../pa2/xv6-pa2-src/* .
rm -f pa2-submit.tar.gz
make clean
```

  - PA3 only makes sense if you have multithreading
    - if your PA2 code is not working, you will not be able to pass any of the PA3 tests and you will end up with a very low score

- Part 1 - preparation
  - reading code and documentation

- Part 2 - implement mutex functions
  - `kthread_mutex_alloc()`, `kthread_mutex_dealloc()`, `kthread_mutex_lock()`, `kthread_mutex_unlock()`
  - no condition variables!



Submission

- Which files do you need to modify?
  - open a terminal and type the following:

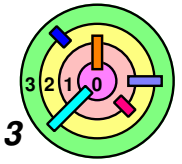
```
pwd
cd ~/cs350/pa3/xv6-pa3-src
make -n pa3-submit
```

    - you should see:

```
tar cvzf pa3-submit.tar.gz \
        Makefile \
        pa3-README.txt \
        proc.c \
        proc.h \
        syscall.c \
        sysproc.c \
        kthread.h \
        exec.c
```

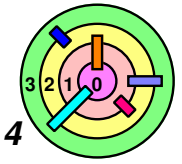
      - some files may be the same as in your PA2 submission

  - these are the only files are are supposed to submit
    - if you submit additional files, the grader will have to delete them before grading
    - if you submit binary files, points will be deducted



Part 1: Preparation

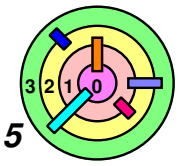
- Read POSIX Threads Programming tutorial by Blaise Barney from Lawrence Livermore National Laboratory
  - `pthread_mutex_lock()`, `pthread_mutex_unlock()`
- Read the `spinlock` code in XV6



Part 2: Implement Mutex Functions

- Implement mutex API for kernel
  - `kthread_mutex_alloc()`, `kthread_mutex_dealloc()`, `kthread_mutex_lock()`, `kthread_mutex_unlock()`
  - unless otherwise specified, we use the term *lock* and *mutex* interchangeably (although in general, a lock may allow multiple threads to have concurrent access to a resource)
  - you will implement these functions in "`proc.c`" and add the following to "`kthread.h`"

```
#define MAX_MUTEXES 64
int kthread_mutex_alloc();
int kthread_mutex_dealloc(int mutex_id);
int kthread_mutex_lock(int mutex_id);
int kthread_mutex_unlock(int mutex_id);
```



Changes In "proc.h"

- Mutex data structures
  - what are the possible mutex states?

```
enum mutexstate { MUNUSED, MLOCKED, MUNLOCKED };
```

  - what should go into a mutex struct?
    - at a minimum:

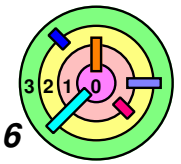
```
int mid; // unique mutex ID ≥ 1
enum mutexstate;
```

Changes In "proc.c"

- Similar to `ptable`, we need a `mtable`
- Need a global variable to know what mutex ID to return next

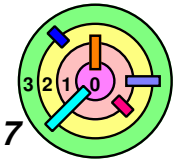
```
int nextmid = 1;
```

  - must never reuse a mutex ID



kthread_mutex_alloc()

- Allocates a mutex object and initializes it; the initial state should be unlocked
  - should return the ID of the initialized mutex, or -1 upon failure



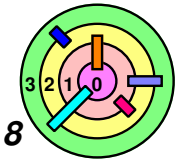
Looping Through Mutex Table

Something like the following:

```
for(m = mtable.mutex; && m < &mtable.mutex[MAX_MUTEXES]; Mp++) {
  ...
}
```

Or:

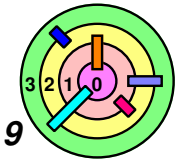
```
for(i = 0; i < MAX_MUTEXES; i++) {
  m = &mtable.mutex[i];
  ...
}
```



kthread_mutex_alloc()

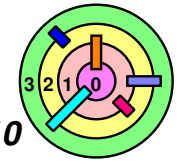
```
kthread_mutex_alloc():

  create a mutex pointer m (using struct in "proc.h")
  Loop through mutex table
    if m is unused
      m->mutex_id = nextmid++;
      m->state = MUNLOCKED;
      initialize all other values if needed
  if m == &mtable.mutex[MAX_MUTEXES]
    return -1
  else
    return m->mutex_id
```

- Note: the above is not the only way to implement mutex allocation
- also, this is not a complete pseudocode
  - you have to add locks if necessary



kthread_mutex_dealloc()

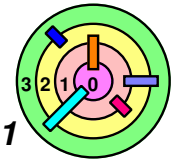
- De-allocates a mutex object which is no longer needed
  - the function should return 0 upon success and -1 upon failure
    - if the given mutex is currently locked, this function should return -1



kthread_mutex_dealloc()

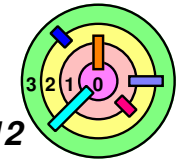
```
kthread_mutex_dealloc(int mutex_id):

  create a mutex pointer m (using struct in "proc.h")
  loop through mutex table to find given mutex_id
    if m is locked
      return -1

  if not found
    return -1
  else
    m->mid = 0
    m->state = MUNUSED
    zero out all the other values if needed
    return 0
```

- Note: the above is not the only way to implement mutex deallocation
- also, this is not a complete pseudocode
  - you have to add locks if necessary



kthread_mutex_lock()

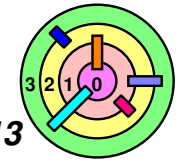
- This function is used by a thread to lock the mutex specified by the argument `mutex_id`
  - if the mutex is already locked by another thread, this call will block the calling thread (change the thread state to `TBLOCKED`) until the mutex is unlocked
    - you may add a `TBLOCKED` state in "`proc.h`" if you'd like



kthread_mutex_lock()

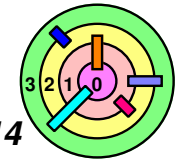
```
kthread_mutex_lock(int mutex_id):

  create a mutex pointer m (using struct in "proc.h")
  loop through mutex table to find the target mutex id (parameter)
    if m->mid == mutex_id
      break;
  if not found
    return -1
  while (m->state == MLOCKED)
    sleep // on m
  if (m->state != MUNLOCKED)
    return -1

  m->state = MLOCKED
  return 0
```

- Note: the above is not the only way to implement mutex lock
- also, this is not a complete pseudocode
  - you have to add locks if necessary



kthread_mutex_unlock()

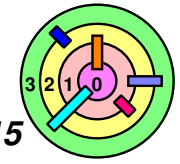
- This function unlocks the mutex specified by the argument `mutex_id` if called by the owning thread, and if there are any blocked threads, one of the threads will acquire the mutex
  - the mutex may be owned by one thread and unlocked by another
  - an error will be returned if the mutex was already unlocked



kthread_mutex_unlock()

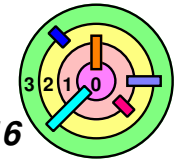
```
kthread_mutex_unlock(int mutex_id):

  create a mutex pointer m (using struct in "proc.h")
  loop through mutex table to find the target mutex id (parameter)
    if m->mid == mutex_id
      break;
  if not found
    return -1
  while (m->state == MUNLOCKED)
    return -1

  m->state = MUNLOCKED
  call wakeup on m to wake up all threads waiting for this mutex
  return 0
```

- Note: the above is not the only way to implement mutex lock
- also, this is not a complete pseudocode
  - you have to add locks if necessary



Common Errors

- You are supposed to read the code of the test programs
  - `mutextest1.c`
  - `mutextest2.c`
- You are supposed to be reading the XV6 book: xv6-rev11.pdf and the XV6 source code to understand how the spinlock (and locking in general) works in XV6



Common Errors

```
panic: sched locks
```

- this means process holding multiple locks
  - before calling `sched()` make sure to release all other locks

```
ac(ptable)
ac(mtable)

rel(mtable)
rel(ptable)
```