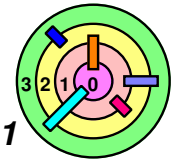
CS 350
PA2: Kernel Level Threads

Bill Cheng

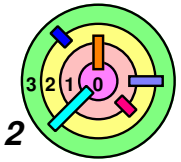
http://merlot.usc.edu/william/usc/

Based on slides created by Kivilcim Cumbul



PA2

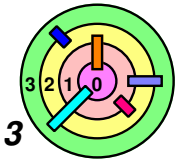
- Set up a standard 32-bit Ubuntu 16.04 system
  - download xv6 for PA2
- Part 1 - add threads to the kernel
  - `fork()`, `exec()`, `exit()`
- Part 2 - thread system calls
  - `kthread_create()`, `kthread_id()`, `kthread_exit()`, `kthread_join()`



Download XV6 For PA2

- Follow the instructions on the PA2 spec
- Open a *terminal* and type the following

```
cd
cd cs350
mkdir pa2
cd pa2
wget --user=USERNAME --password=PASSWORD \
  http://merlot.usc.edu/cs350-m25/programming/pa2/xv6-pa2-src.tar.gz
tar xvf xv6-pa2-src.tar.gz
cd xv6-pa2-src
```



Submission

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

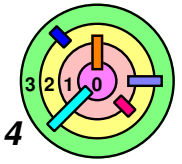
```
pwd
cd cs350/pa2/xv6-pa2-src
make -n pa2-submit
```

    - you should see:

```
tar cvzf pa2-submit.tar.gz \
        Makefile \
        pa2-README.txt \
        proc.c \
        proc.h \
        syscall.c \
        sysproc.c \
        kthread.h \
        exec.c
```

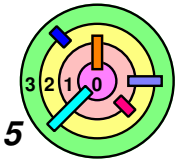
      - I don't think you need to modify Makefile at all

  - 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: Add Threads To The Kernel

- Change the implementation of some existing system calls
  - `fork()`, `exec()`, `exit()`
    - `fork()`, and `exit()` are in "`proc.c`"
    - `exec()` is in "`exec.c`"
  - note: some functions might not need changes (you need to pick which ones to change)

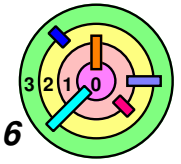


Background: `growproc()`

`growproc()` is responsible for retrieving more memory when the process asks for it

```
// Grow current process's memory
//     by n bytes.
// Return 0 on success,
//     -1 on failure.
int
growproc(int n)
{
  uint sz;
  acquire(&ptable.lock);
  sz = proc->sz;
  if (n > 0){
    if ((sz = allocuvm(
        proc->pgdir, sz,
        sz + n)) == 0){
      release(&ptable.lock);
      return -1;
    }
  } else if (n < 0){
    if ((sz = deallocuvm(
        proc->pgdir, sz,
        sz + n)) == 0){
      release(&ptable.lock);
      return -1;
    }
  }
  proc->sz = sz;
  switchuvm(proc);
  release(&ptable.lock);
  return 0;
}
```

- to ***access any PCB***, must acquire the `ptable.lock` spinlock
- need to synchronize accesses to `proc->sz`

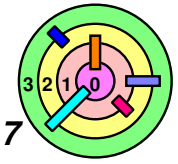


Background: `growproc()`

- `growproc()` is responsible for retrieving more memory when the process asks for it

```
// Grow current process's memory
//     by n bytes.
// Return 0 on success,
//     -1 on failure.
int
growproc(int n)
{
  uint sz;
  acquire(&ptable.lock);
  sz = proc->sz;
  if (n > 0){
    if ((sz = allocuvm(
        proc->pgdir, sz,
        sz + n)) == 0){
      release(&ptable.lock);
      return -1;
    }
  } else if (n < 0){
    if ((sz = deallocuvm(
        proc->pgdir, sz,
        sz + n)) == 0){
      release(&ptable.lock);
      return -1;
    }
  }
  proc->sz = sz;
  switchuvm(proc);
  release(&ptable.lock);
  return 0;
}
```

- always release locks before return statement if it is not released previously

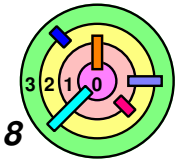


Background: `growproc()`

- `growproc()` is responsible for retrieving more memory when the process asks for it

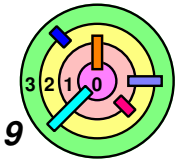
```
// Grow current process's memory
//     by n bytes.
// Return 0 on success,
//     -1 on failure.
int
growproc(int n)
{
  uint sz;
  acquire(&ptable.lock);
  sz = proc->sz;
  if (n > 0){
    if ((sz = allocuvm(
        proc->pgdir, sz,
        sz + n)) == 0){
      release(&ptable.lock);
      return -1;
    }
  } else if (n < 0){
    if ((sz = deallocuvm(
        proc->pgdir, sz,
        sz + n)) == 0){
      release(&ptable.lock);
      return -1;
    }
  }
  proc->sz = sz;
  switchuvm(proc);
  release(&ptable.lock);
  return 0;
}
```

- you might need to think more about synchronization and find where to put functions/methods, locks, etc.



`fork()`

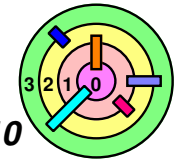
- `fork()` should duplicate only the calling thread, if other threads exist in the process they will notexist in the new process
- Questions to ask:
  - are there any conflicts between shared variables?
  - do we need to kill any threads after calling fork?
  - is the acquired the lock enough for synchronization or should we put more locks?



exit()

- `exit()` should kill the process and all of its threads, remember while a single threads executing `exit()`, others threads of the same process might still be running

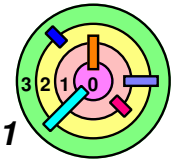
```
kill_all();

// jump into the scheduler, never to return
thread->state = TINVALID;
proc->state = ZOMBIE;
sched();
panic("zombie exit");
```



kill_all()

We have to create a `kill_all()` function to kill all the alive threads:

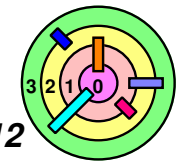
```
kill_all():
  create thread pointer *t
  for each thread t:
    if (thread t is not the current thread and
        not running and not unused) then
      make t a zombie
    end-if
  end-for
  make current thread zombie
  kill process
```



exec()

- The thread performing exec should "tell" other threads of the same process to destroy themselves and only then complete the `exec()` task
- modify `kill_all()` method and create `kill_others()`
  - `kill_others()` kills all alive threads but itself

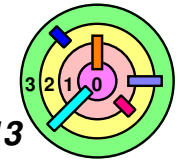
```
kill_others():
  create thread pointer *t
  for each thread t:
    if (thread t is not the current thread and
        not running and not unused) then
      make t a zombie
    end-if
  end-for
```



Part 2: Thread System Calls

- Implement thread API for kernel
  - `kthread_create()`, `kthread_id()`, `kthread_exit()`, `kthread_join()`
  - you will implement these functions in "`proc.c`" and add the following to "`kthread.h`"

```
int kthread_create(void*(*start_func)(),
                   void* stack, int stack_size);
int kthread_id();
void kthread_exit(void);
int kthread_join(int thread_id);
```



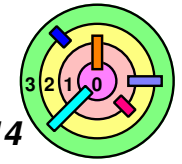
Changing Thread States

```
t->state = TZOMBIE;
```

- in "proc.h"

```
enum threadstate {
  TUNUSED,
  TEMBRYO,
  TSLEEPING,
  TRUNNABLE,
  TRUNNING,
  TZOMBIE,
  TINVALID
};
```

```
enum procstate {
  UNUSED,
  USED,
  ZOMBIE
};
```



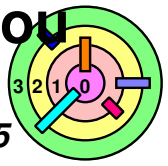
Changing Thread States

`t->tid != thread->tid;`

- `thread`, `proc`, and `cpu` are global variables that point to the current thread, the current process, and the current CPU
- in "`proc.h`"

```
struct thread {
  int tid;                          // Unique thread ID
  enum threadstate state;           // thread state
  char *kstack;                     // Bottom of kernel stack for this thread
  struct proc *parent;              // Parent process
  struct trapframe *tf;             // Trap frame for current syscall
  struct context *context;          // swtch() here to run process
  void *chan;                       // If non-zero, sleeping on chan
  int killed;                       // If non-zero, have been killed
};
```

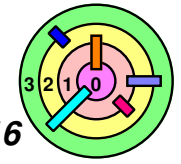
- Read the code in `allocthread()` to see how every field is initialized
  - e.g., since the trap frame is the bottom of the kernel stack, you don't need to allocate memory for the trap frame



How To Loop Through Threads?

Look at `allocthread()` in "`proc.c`"

```
struct thread*
allocthread(struct proc * p)
{
  struct thread *t;
  for(t = p->threads; found != 1 && t < &p->threads[NTHREAD]; t++) {
    ...
  }
  ...
}
```



How To Loop Through Processes?

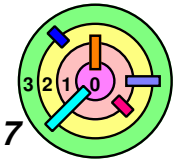
- Look at `allocproc()` in "`proc.c`"

```
struct thread*
allocproc()
{
  struct proc *p;
  struct thread *t;

  for(p = ptable.proc; && p < &ptable.proc[NPROC]; p++) {
    ...
  }
  ...
}
```

- Need a global variable to know what process ID and thread ID to return next

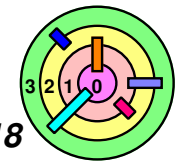
```
int nextpid = 1;
int nexttid = 1;
```

  - must never reuse a process ID or a thread ID



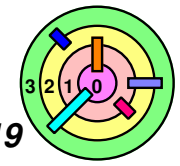
kill_all()

```
kill_all():
  create thread pointer *t
  for each thread t: // loop through threads
    if (t is not current thread and not running
        and not unused) then // check its state
      make t a zombie // change its state
    end-if
  end-for
  make current thread zombie // find current thread
                             // and change its state
  kill process // proc->killed = 1
```



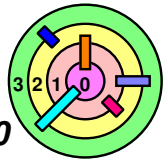
kthread_create()

- Calling `kthread_create()` will create a new thread within the context of the calling process
  - the newly created thread state will be TRUNNABLE
  - the caller of `kthread_create()` must allocate a user stack for the new thread to use (it should be enough to allocate a single page i.e., 4K for the thread stack)
  - this does not replace the kernel stack for the thread
- `start_func` is a pointer to the entry function, which the thread will start executing
  - upon success, the identifier of the newly created thread is returned
  - in case of an error, a non-positive value is returned



kthread_create()

- The kernel thread creation system call on real Linux does not receive a user stack pointer
  - in Linux the kernel allocates the memory for the new thread stack
  - you will need to create the stack in user mode and send its pointer to the system call in order to be consistent with current memory allocator of xv6

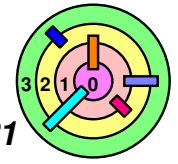


kthread_create()

```
kthread_create(void* (*start_func)(), void* stack, int stack_size):

  create a thread pointer
  allocate the thread using allocthread() function
  check if t is 0 // allocated correctly?
    if not, return -1
  else
    copy current thread's trap frame
    find stack address of the thread using stack pointer given parameter
    make stack pointer inside trap frame stack address + stack size
    update base pointer inside trap frame as stack pointer
    find address of the start function which is given in parameter
    make instruction pointer inside trap frame start address
    return tid
```

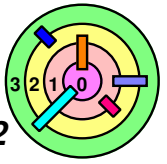
- stack pointer: `t->tf->esp`
- base pointer: `t->tf->ebp`
- instruction pointer: `t->tf->eip`

⇨ **Note: the above is not the only way to create a thread**



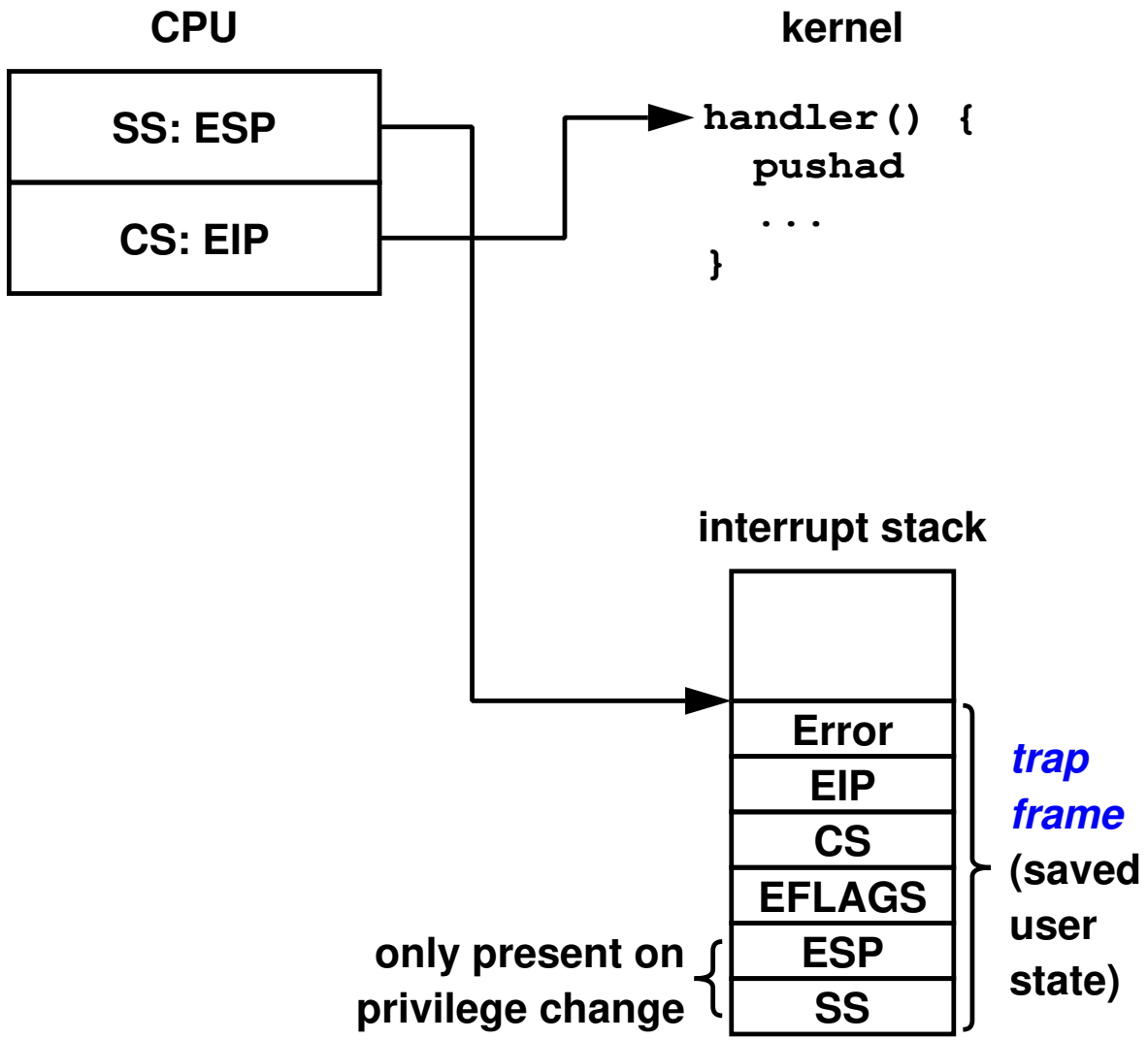
esp, eip, ebp

- `t->tf->esp`
  - we have to change the stack address of new thread
  - so we use given parameter to make stack address different than current thread
  - add given stack's address with stack size to find where to put stack pointer
- `t->tf->ebp`
  - initially base pointer and stack pointer point the same place
- `t->tf->eip`
  - instruction pointer points the instructions which will be implemented by thread
  - this pointer has to point stack function at the beginning



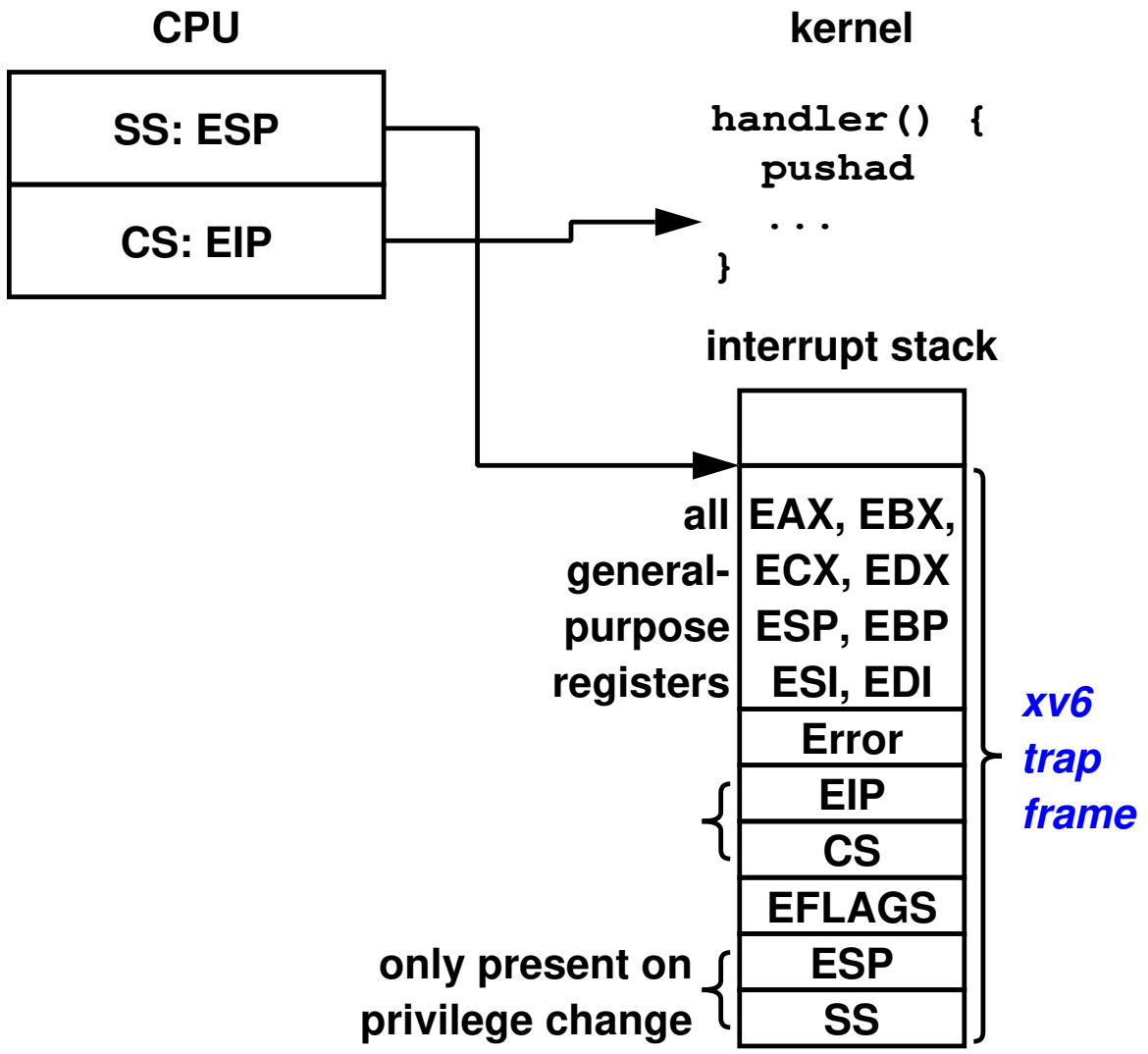
More Information In Ch 3 Of XV6 Book

- Figure 3.1 of xv6 book (with low address on top and high address on the bottom) shows the kernel stack after an `INT` instruction
  - also in Ch 2 slides for the x86 CPU



More Information In Ch 3 Of XV6 Book

- Figure 3.1 of xv6 book (with low address on top and high address on the bottom) shows the kernel stack after an `INT` instruction
  - also in Ch 2 slides for the x86 CPU

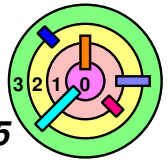


kthread_id()

- Easiest function to implement in PA2
  - upon success, this function returns the caller thread's id
  - in case of error, a non-positive error identifier is returned
  - remember, thread id and process id are not the same thing

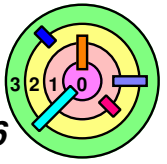
```
kthread_id():

  if process and thread exists
    return t->tid
  else
    return -1
```

- Note: this is not the only way to return a thread id



`kthread_exit()`

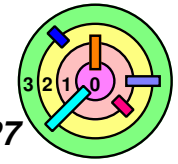
- This function terminates the execution of the calling thread
  - if called by a thread (even the main thread) while other threads exist within the same process, it shouldn't terminate the whole process
  - if it is the last running thread, process should terminate
  - each thread must explicitly call `kthread_exit()` in order to terminate normally



kthread_exit()

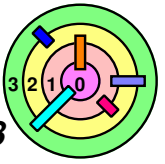
```
kthread_exit():

  create a thread pointer
  create a found flag
  loop through all threads to find another thread running
    if t is not current thread // because calling thread is current
      if t is not unused, not a zombie, and not invalid
        make found flag true
        break // only one running is enough

  if found // I am not the last thread in my process
    wakeup all waiting using wakeup1() // read wakeup1() code
  else // found flag is false, therefore, I'm the last thread
    exit()
  make this thread zombie
  call sched() to schedule another thread
```

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



kthread_join()

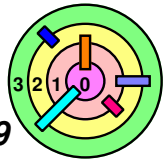
- This function suspends the execution of the calling thread until the target thread (of the same process), indicated by the argument thread_id, terminates
  - if the thread has already exited, execution should not be suspended
  - if successful, the function returns zero
  - otherwise, -1 should be returned to indicate an error



kthread_join()

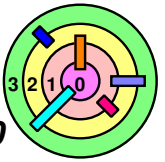
```
kthread_join(int thread_id):

  check if thread_id is valid
  create a thread pointer t
  loop through all threads to find target thread id (parameter)
    make t points to target thread with thread_id
  if not found
    return -1
  while (t->tid == thread_id and t is not in the TZOMBIE state)
    make t sleep using sleep() function with a lock // read sleep() code
  if state of t is zombie
    clearThread(t);
  return 0
```

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



Common Errors

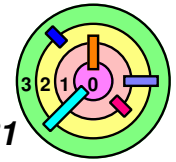
- You are supposed to read the code of the test programs
  - `threadtest1.c`
  - `threadtest2.c`
  - `threadtest3.c`
- You are supposed to be reading the XV6 book: xv6-rev11.pdf and the XV6 source code to understand how the scheduler works



Common Errors

```
$ threadtest1
3 threadtest1: unknown sys call 23
thread in main -1,process 3
3 threadtest1: unknown sys call 22
3 threadtest1: unknown sys call 22
3 threadtest1: unknown sys call 25
Got id : -1
3 threadtest1: unknown sys call 25
Got id : -1
Finished.
3 threadtest1: unknown sys call 24
...
```

Make sure to implement system calls for all kthread functions



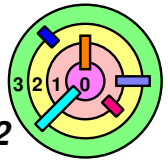
Common Errors

```
cpu with apicid 0: panic: acquire
 80104b85 80104334 80100226 80101a78 80101c4a ...
```

OR:

```
cpu with apicid 0: panic: release
 80104cc8 80103b51 80105dec 80105129 80106338 801060eb ...
```

- **panic: acquire and panic: release errors mean that program fails to acquire lock because it is already acquired earlier or it cannot released lock because it is already released**



Common Errors

```
panic: sched locks
```

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

```
ac(ptable)

rel(ptable)
```