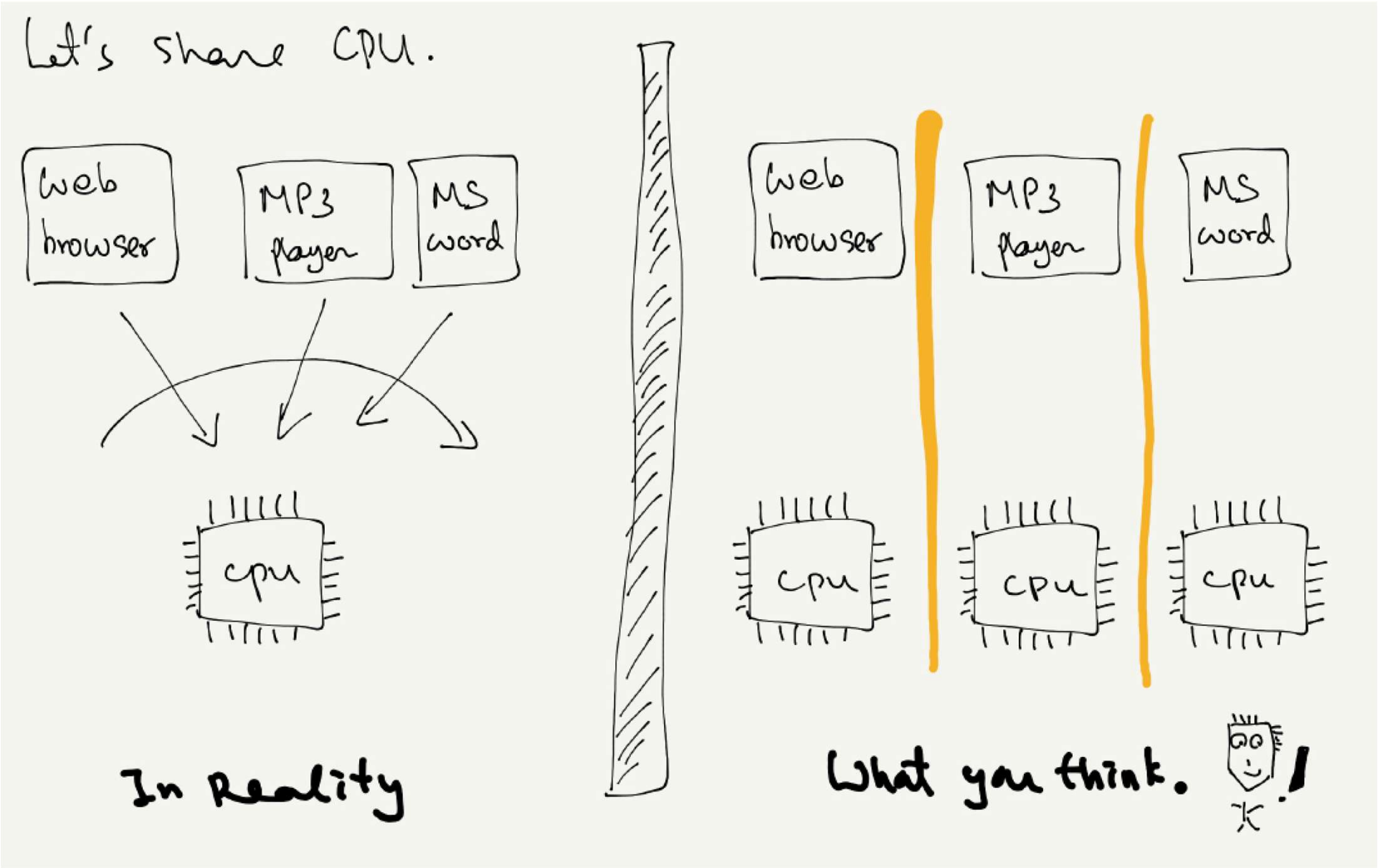
CPU Scheduling

Youjip Won

KAIST

Outline

- sharing CPU
- `swtch()`
- `sched()`
- `switch()`
- `sleep()` and `wakeup()`


Let's share CPU.
Web browser
MP3 player
MS word
cpu
In Reality
Web browser
MP3 player
MS word
cpu
cpu
cpu
What you think.

Multiplexing

- number of processes >> number of processors
- multiplexing the processes onto the hardware processors

- Way to release CPU
  - voluntary context switch
    - Based on sleep & wakeup
    - waits for device or pipe IO to complete
    - waits for a child to exit
  - involuntary context switch
    - Based on timer interrupt
    - Multiplexing by scheduler

Issues in CPU multiplexing

- context switch mechanism: Which informations are saved and restored during the switch?
- How to do it transparently: timer interrupt.
- Avoid race condition: Many CPU's perform context switches concurrently. How to avoid race condition when multiple processors are switching processes?
- Release resources: How can exiting process release its resources?
- Maintain information on current processes.

Switching the processes in xv6

- Every process has its own kernel stack and register set.
- Each CPU has a its own scheduler thread.
- Switching from one thread to another
  - saving the old thread's CPU registers at the kernel stack.
  - restoring the previously-saved registers of the new thread from the associated kernel stack.
  - Kernel stack pointer is saved at the `struct context` pointed by `struct proc`.

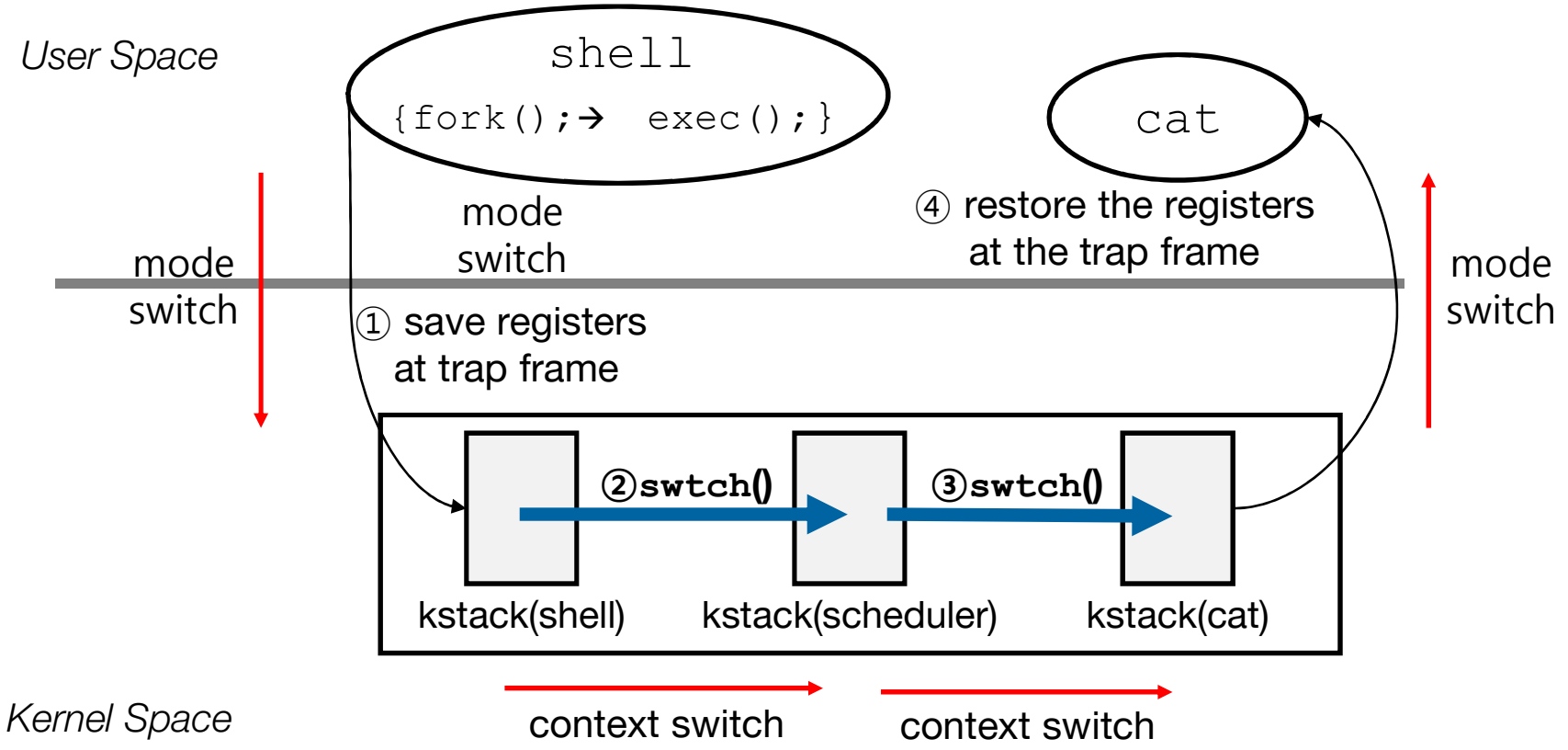
struct proc

- `struct proc` represents per-process state.
- `struct proc` is initialized whenever process is created. (`userinit()`, `fork()`)
- `struct proc` is used for including scheduling process.

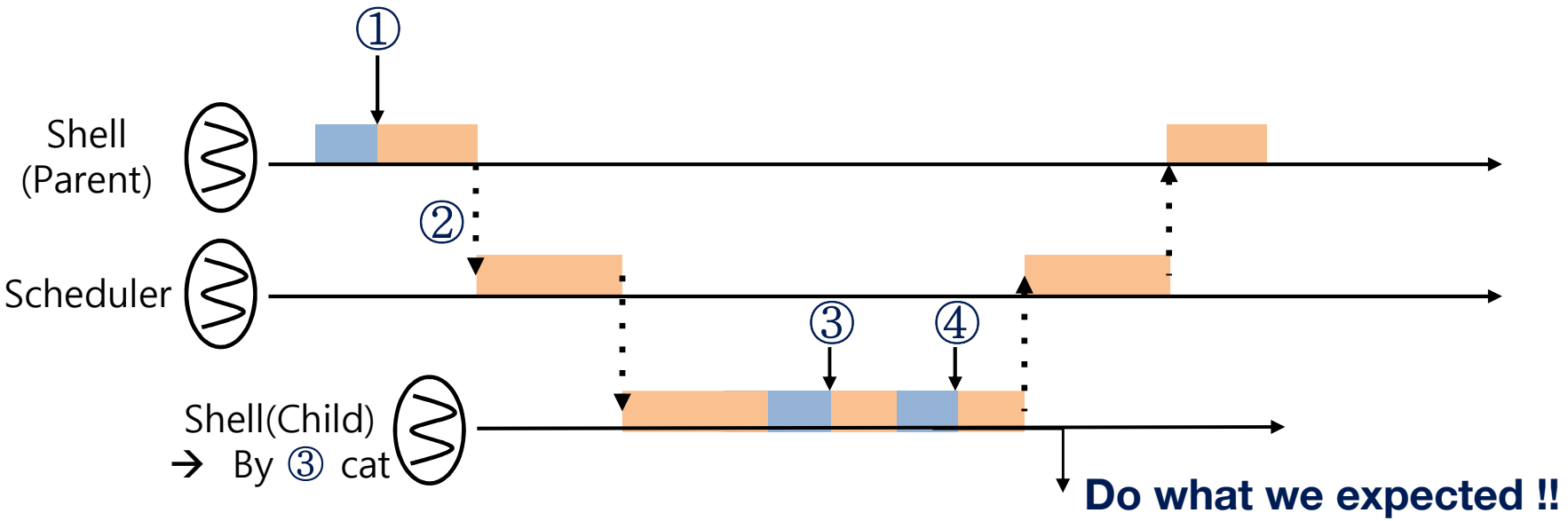
```
// Per-process state
struct proc {
  uint sz;                     // Size of process memory (bytes)
  pde_t* pgdir;                // Page table
  char *kstack;                // Bottom of kernel stack for this process
  enum procstate state;        // Process state
  int pid;                     // Process ID
  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
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)
};
```

switching the processes

```
% cat
```



Switching the processes



① `fork();` **Creating Child Process**

② **Context switching to the Scheduler**

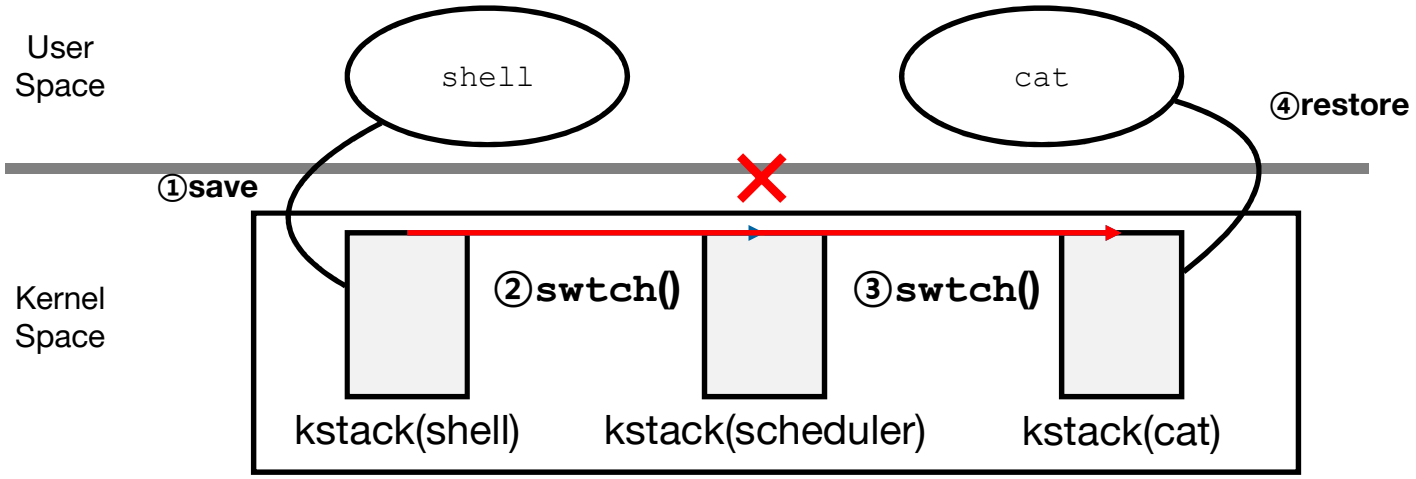
③ `exec();` **Load Cat Binary**

④ `exit();` **Terminate the process**

User Mode

Kernel Mode

switching the processes



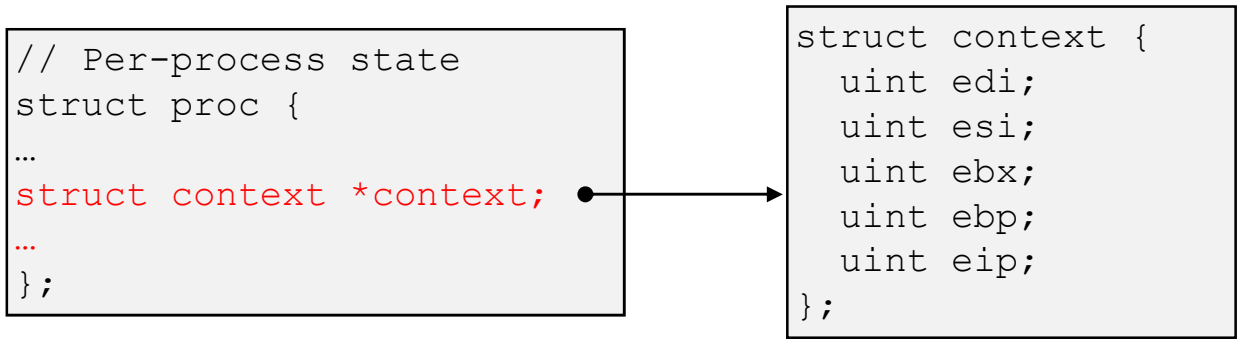
1. stack switch from user to kernel: old process
2. A context switch to the local CPU's scheduler thread.
3. A context switch to a new process's kernel thread.
4. stack switch from kernel to user: new process (A trap return)

separate scheduler thread in xv6

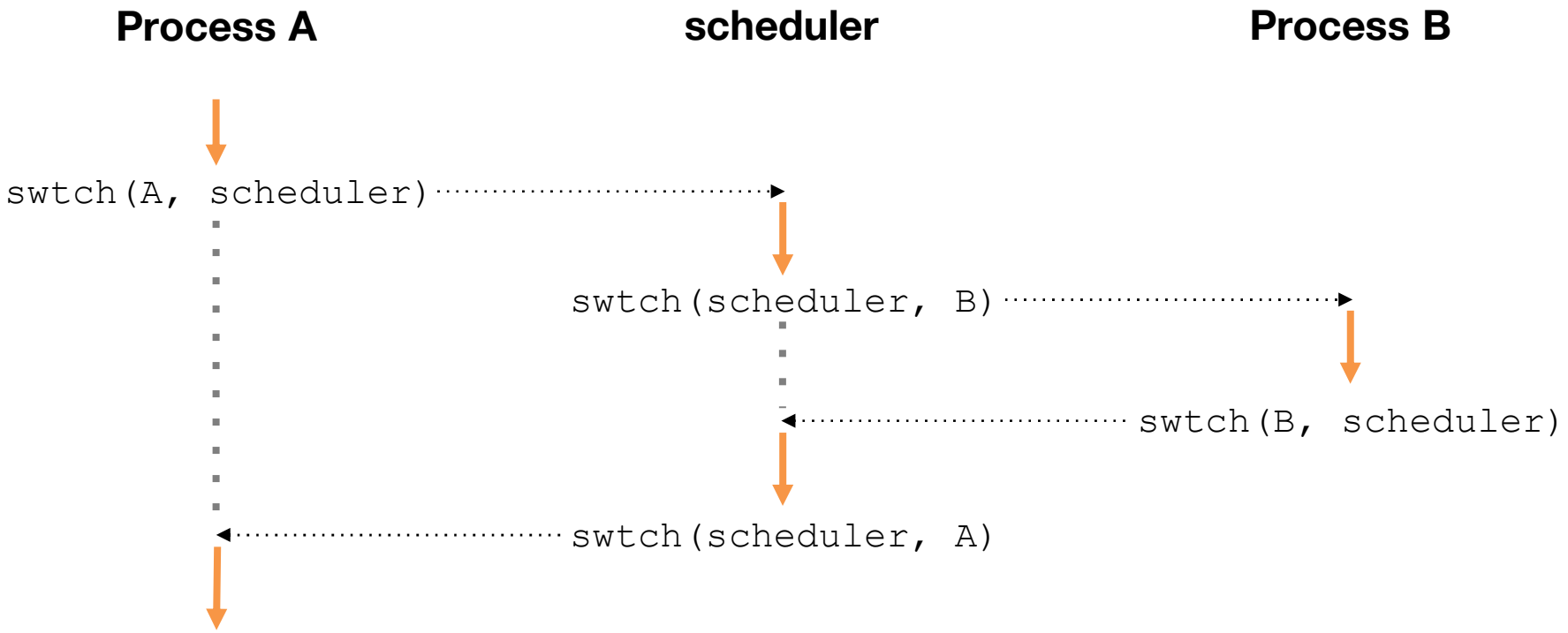
- xv6 scheduler runs on its own thread.
- The process switch accompanies two context switches.
- why?
  - simplify the procedure of cleaning up user processes.
  - `exit()`
  - `kill()`

swtch(void **old, void* new)

- `swtch()` saves and restores the register sets, called **contexts**.
- When it is time for a process to give up the CPU, the process's kernel thread calls `swtch()`.
- **`struct context*`**. It points to a structure stored on the **kernel stack** involved.

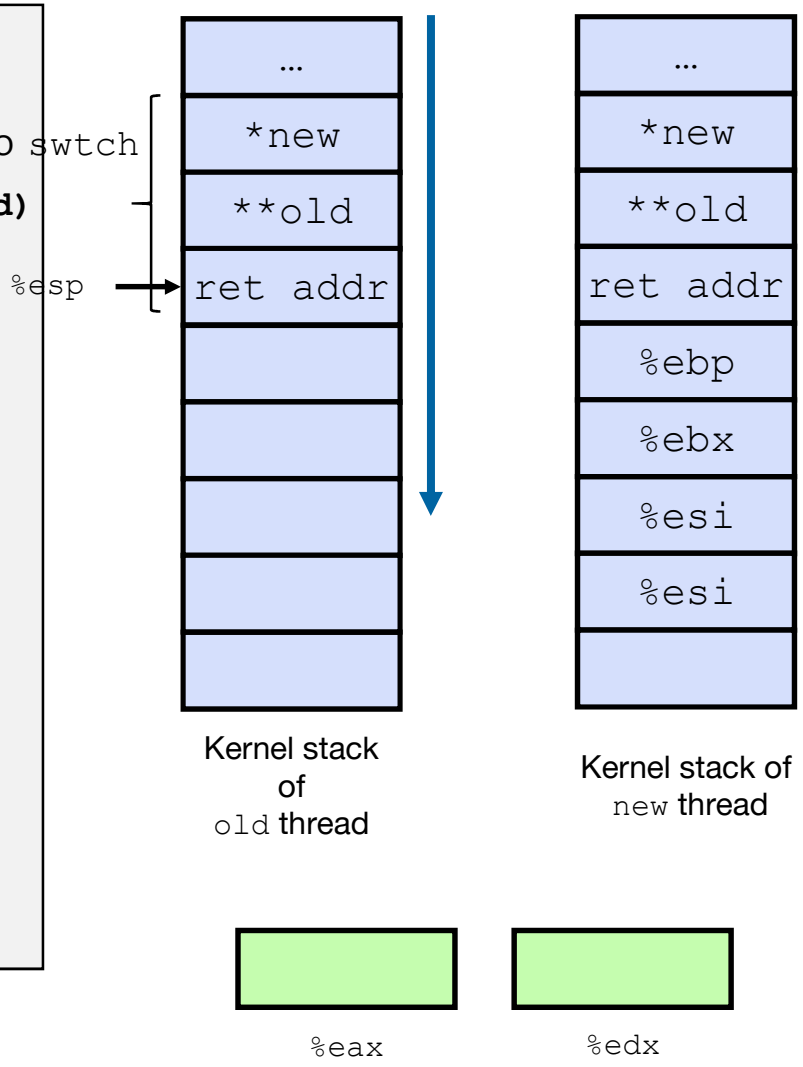


swtch(struct context **old, struct context *new)



swtch(struct context **old, struct context *new)

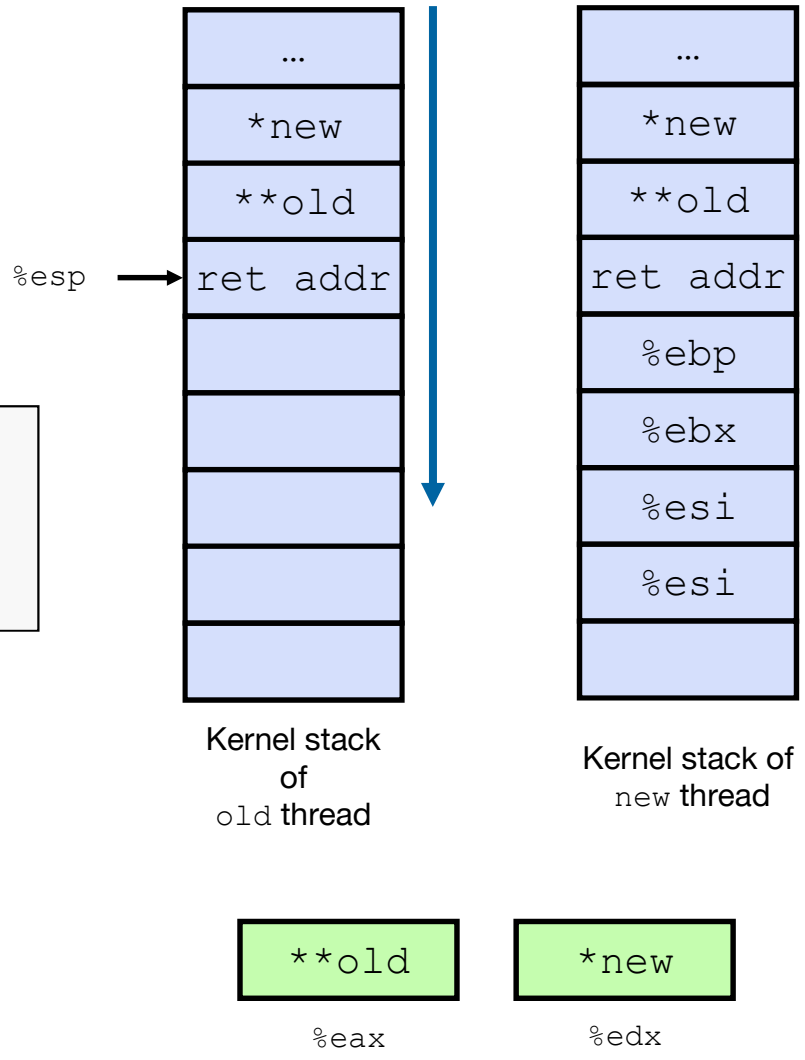
```
# load arguments to %eax and %edx
movl 4(%esp), %eax    //old
movl 8(%esp), %edx    //new

# Pushes the registers (Save context of old)
pushl %ebp
pushl %ebx
pushl %esi
pushl %edi

# Switch stack.
movl %esp, (%eax) //Saves %esp at *old
movl %edx, %esp   //Restore %esp from new

# Load registers (Restore context of new)
popl %edi
popl %esi
popl %ebx
popl %ebp

# make return address as %eip
ret
```



swtch(struct context **old, struct context *new)

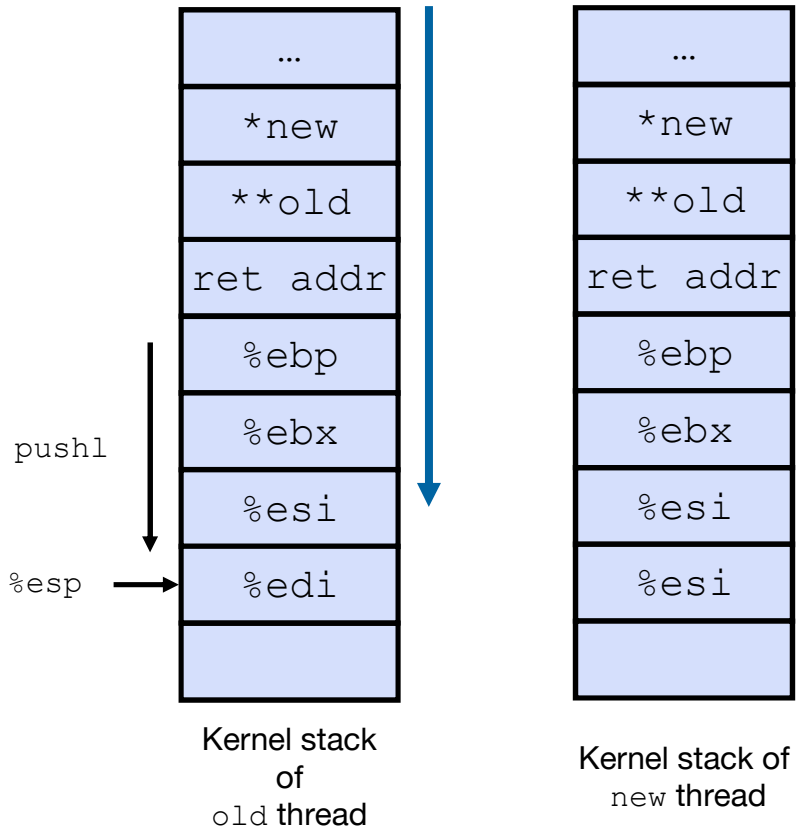
```
# load arguments to %eax and %edx
movl 4(%esp), %eax    //old
movl 8(%esp), %edx    //new
```

```
# For Your Information,
%eax = old = &(p->context);
// Pointer to the old thread's stack pointer
%edx = new = new->context;
// New thread's stack pointer
```



swtch(struct context **old, struct context *new)

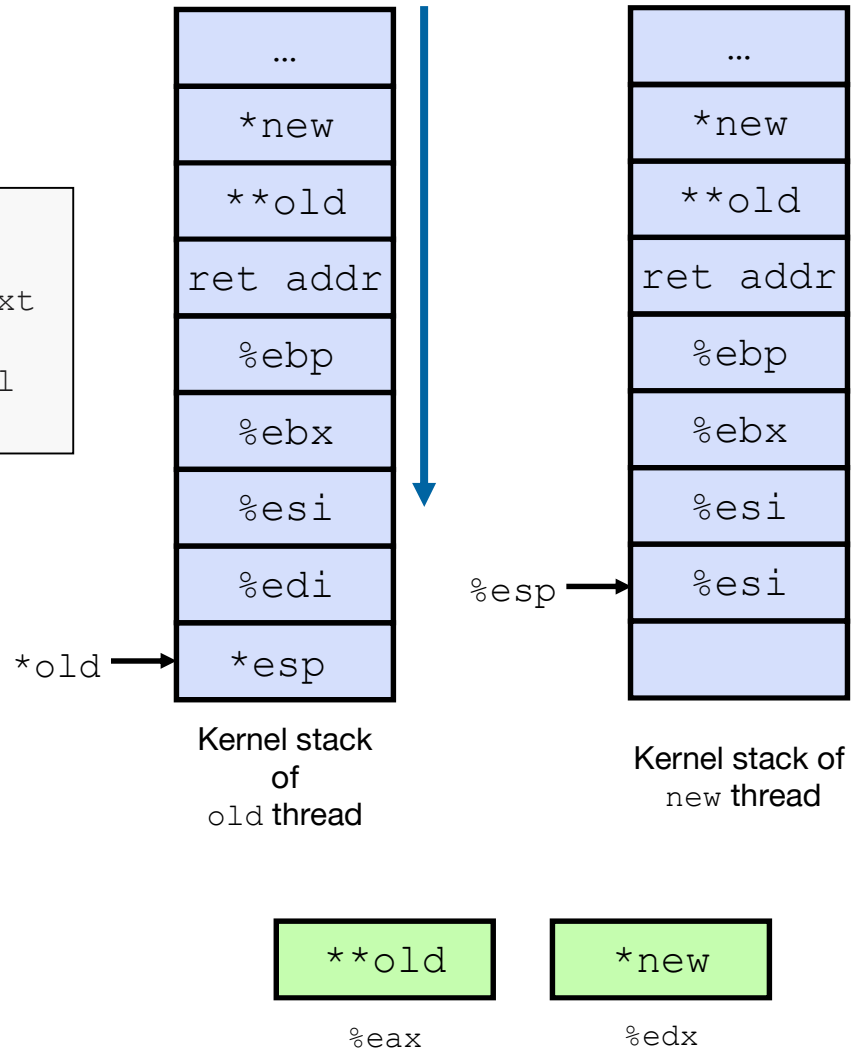
```
# Pushes the registers (Save context of old)
pushl %ebp
pushl %ebx
pushl %esi
pushl %edi
```



swtch(struct context **old, struct context *new)

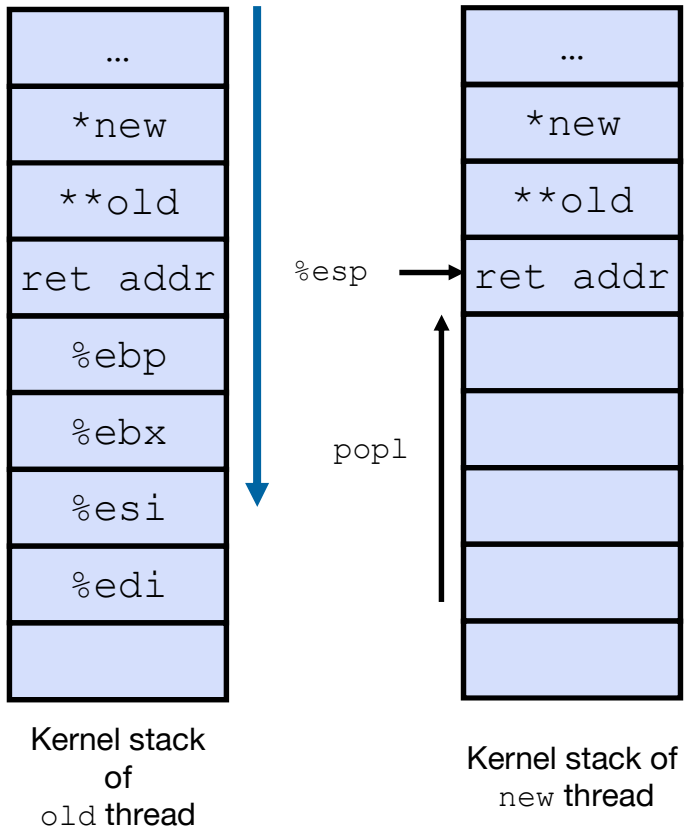
```
# For who may be confusing,
p->context = *old = (%eax) = %esp;
// Save current stack pointer at the p->context
%esp = %edx = new
// Set current stack pointer to the new kernel
stack
```

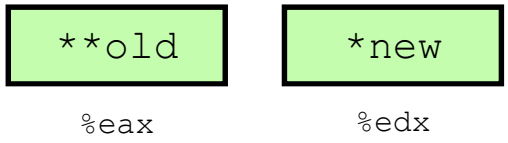
```
# Switch stack.
movl %esp, (%eax) //Saves %esp at *old
movl %edx, %esp   //Restore %esp from new
```



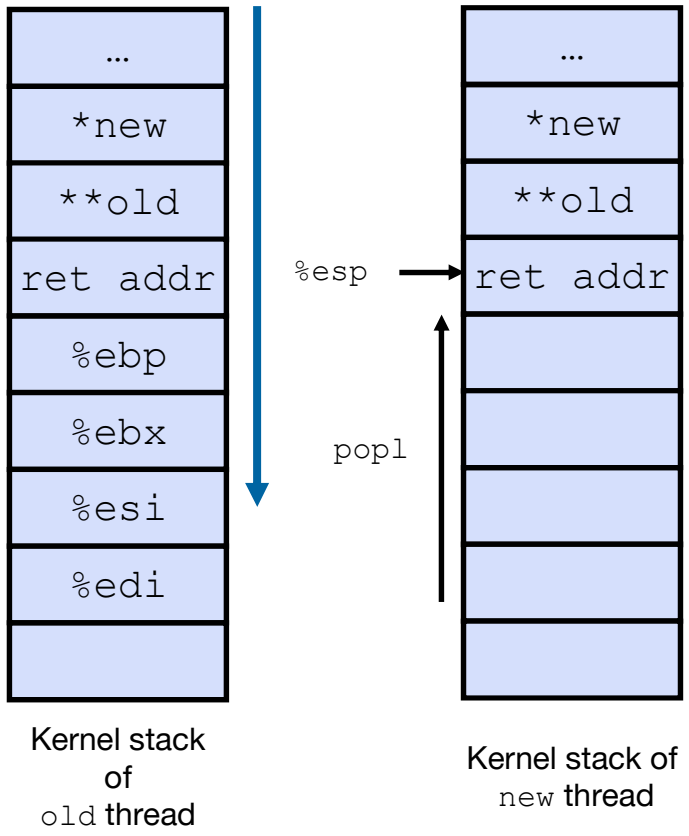
swtch(struct context **old, struct context *new)

```
# Load registers (Restore context of new)
popl %edi
popl %esi
popl %ebx
popl %ebp
```



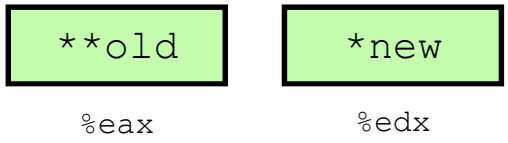


swtch(struct context **old, struct context *new)




```
# make return address as %eip
ret
```

It is returned to `sched()` or `scheduler()`!!!



Releasing the CPU

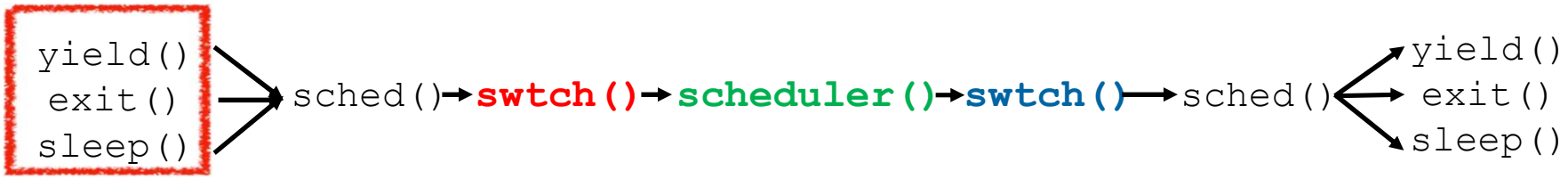
- The function to release the CPU

  `sched()`

- Cases to release the CPU
  - Determined by process's action: `yield()`
  - Exit of process: `exit()`
  - Wait for something such as IO completion or pipe IO: `sleep()`

- Actually doing
  - Switch to `scheduler()`

Releasing the CPU

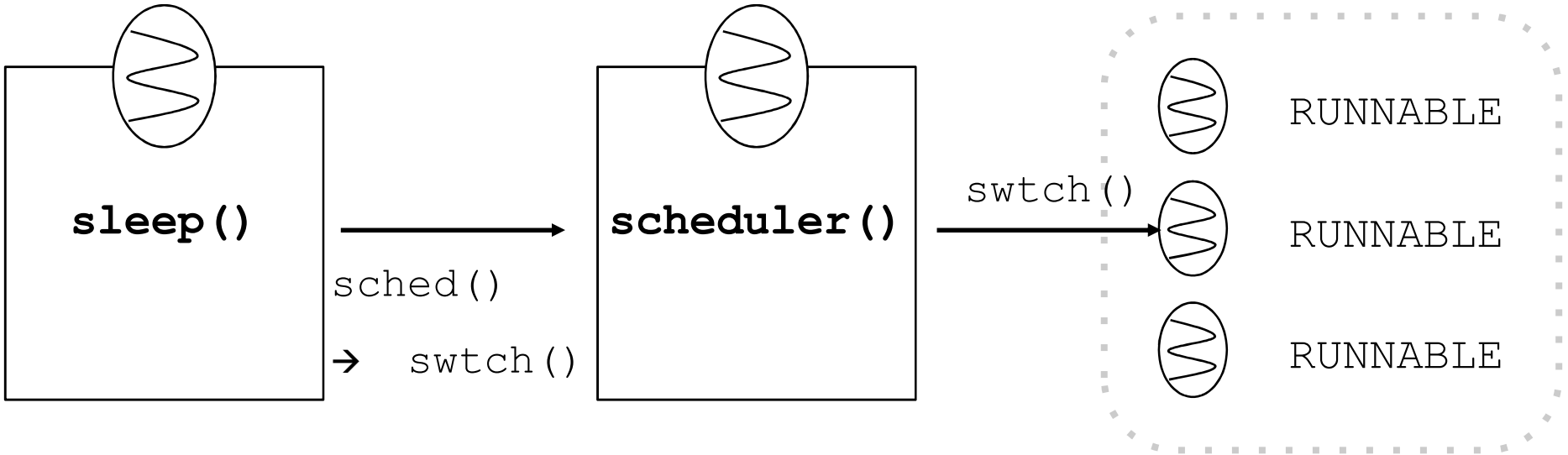


- A process that wants to give up the CPU must
  - acquire the process table lock `ptable.lock`.
  - release any other locks it is holding
  - update its own state (`proc->state`) to `RUNNABLE`.
  - and then call `sched()`.
- yield(), sleep() and exit() follow this convention.

yield()

```
// Give up the CPU for one scheduling round.
void yield(void) {
  acquire(&ptable.lock);  //DOC: yieldlock
  myproc()->state = RUNNABLE;
  sched();
  release(&ptable.lock);
}
```

sched()



- A thread in execution can call `sched()` in three cases.
  - `exit()`
  - `yield()`
  - `sleep()`

`sched()`

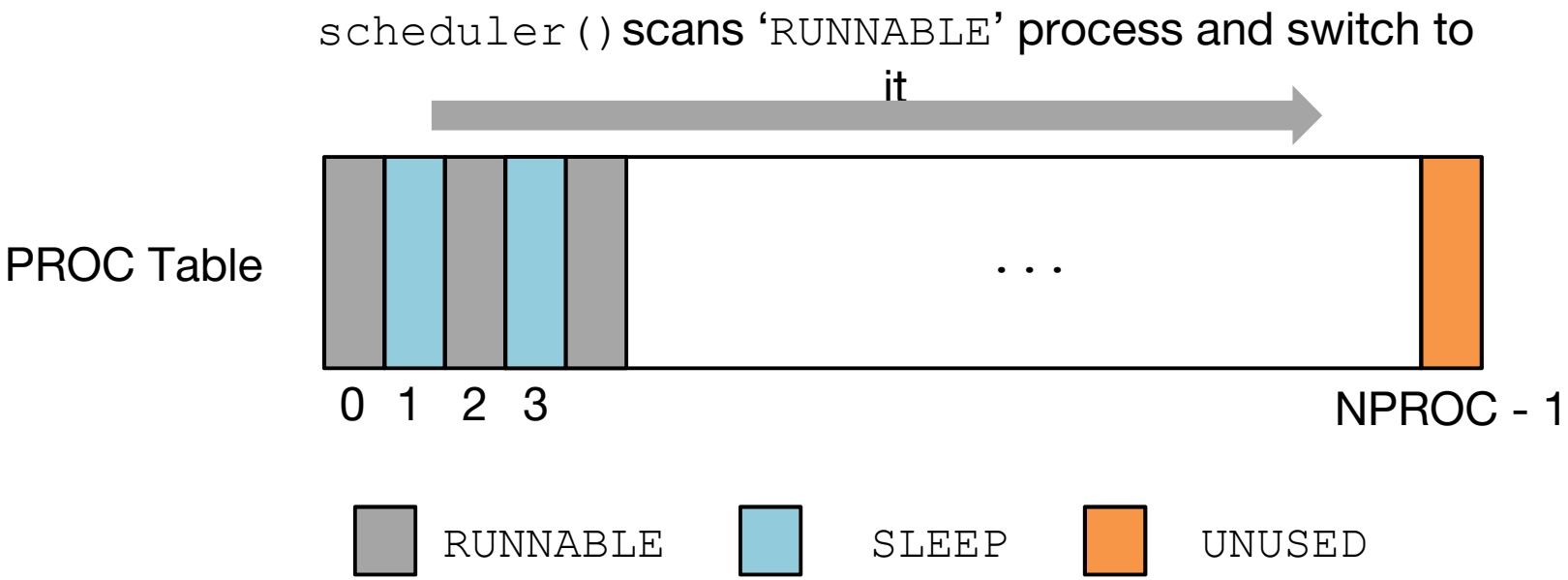
- hand the control over to the scheduler function.
- be sure that interrupts are disabled.
- save the current context in `proc->context` and switch to the scheduler context in `cpu->scheduler`.

sched()

```
void sched(void) {
  int intena;
  struct proc *p = myproc();

  if(!holding(&ptable.lock)) // make sure the ptable is locked.
    panic("sched ptable.lock");
  if(mycpu()->ncli != 1)// make sure interrupt is disabled.
    panic("sched locks");
  if(p->state == RUNNING)//sleep, yield, exit
    panic("sched running");
  if(readeflags()&FL_IF)
    panic("sched interruptible");
  intena = mycpu()->intena;
  swtch(&p->context, mycpu()->scheduler);
  mycpu()->intena = intena;
}
```

scheduler()

- Independent thread by each CPU
- Select CPU to run and switch to the thread



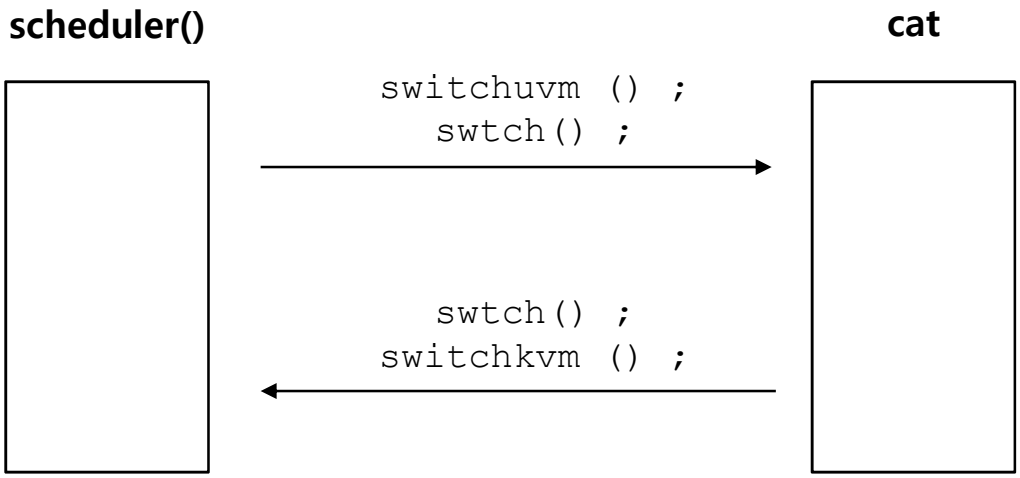
scheduler()

```
void scheduler(void) {
…
for(;;){
  …
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->state != RUNNABLE)
      continue;
  c->proc = p;
  switchuvm(p);
  p->state = RUNNING;
  swtch(&(c->scheduler), p->context);
  switchkvm();
  …
  }
…
}
```

Switching the address space

- `scheduler()` is responsible for switching the address space.
- When the `scheduler()` switches to the user process,
- Select the process to run. → Address space switch (`switchuvm()`)

  → Call `swtch()`
- Once it has returned from `swtch()` ? address space switch (`switchkvm()`)



RECAP : switchuvm()

- switchuvm() switches the stack from kernel to user.
  - It sets the entry at index 5 in GDT to Task state segment
  - It sets the %CR3 to the process's page directory

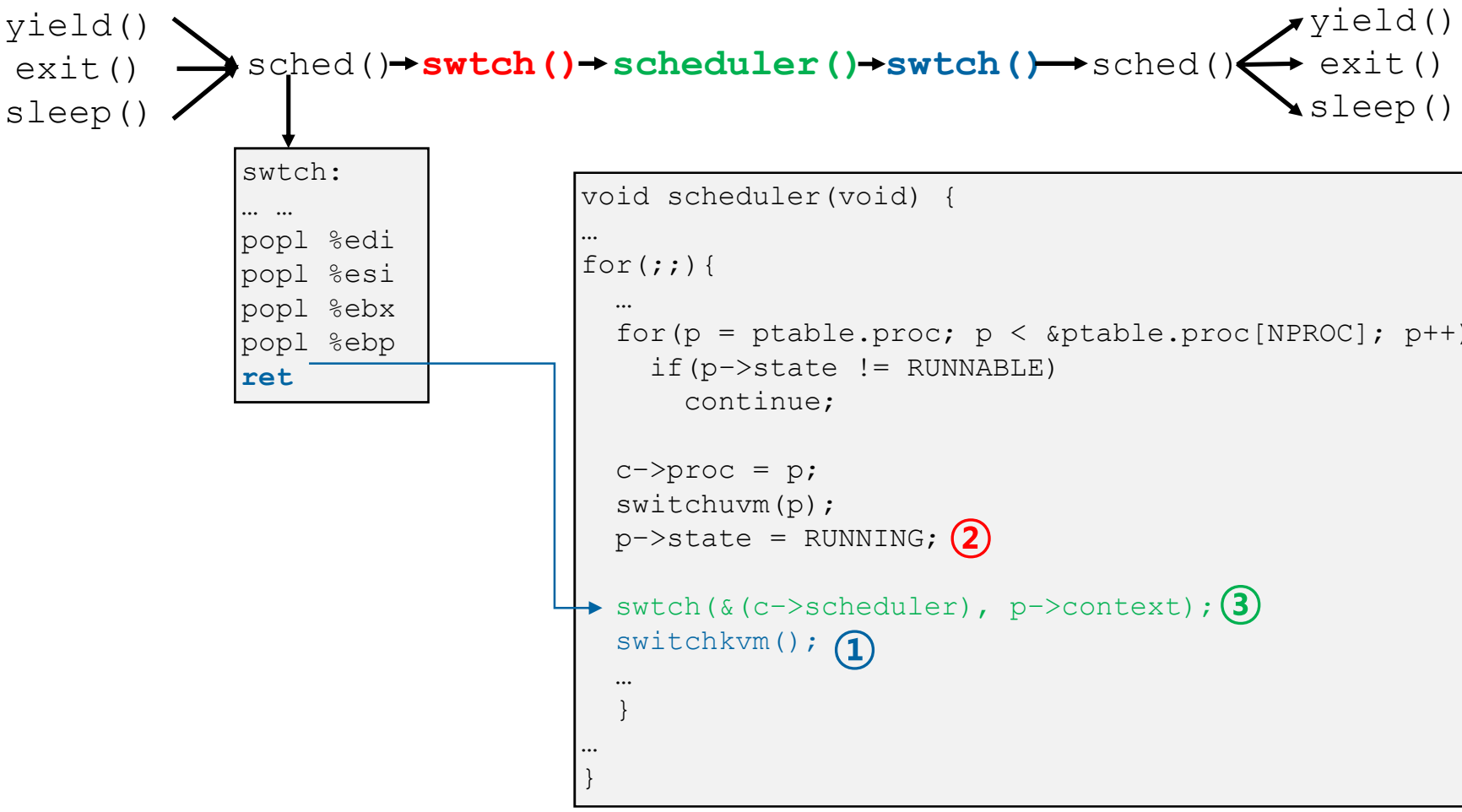
```
157 void switchuvm(struct proc *p) {
                                        …
166   pushcli();
167   mycpu()->gdt[SEG_TSS] = SEG16(STS_T32A, &mycpu()->ts,
168                                 sizeof(mycpu()->ts)-1, 0);
169   mycpu()->gdt[SEG_TSS].s = 0;
170   mycpu()->ts.ss0 = SEG_KDATA << 3;
171   mycpu()->ts.esp0 = (uint)p->kstack + KSTACKSIZE;
                                        …
175   ltr(SEG_TSS << 3);
176   lcr3(V2P(p->pgdir));  // switch to process's address space
177   popcli();
178 }
```

RECAP : switchkvm()

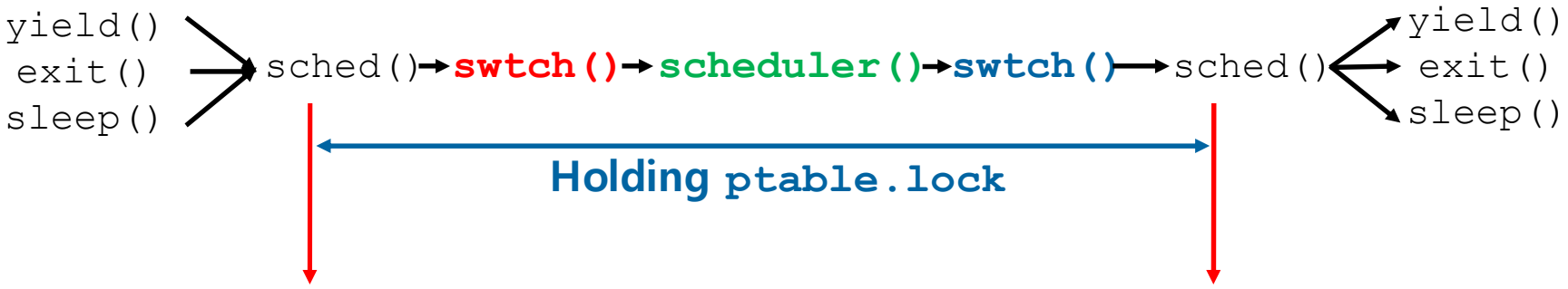
- switchkvm() switches from user to kernel address space.
  - It sets the %CR3 to the kernel's page directory

```
149 void
150 switchkvm(void)
151 {
152   lcr3(V2P(kpgdir));   // switch to the kernel page table
153 }
```

swtch() & scheduler()



acquire & release for `ptable.lock`




```
void yield(void){
  acquire(&ptable.lock);
  myproc()->state = RUNNABLE;
  sched();
  release(&ptable.lock);
}
```

```
void
sleep(void *chan, struct spinlock){
  ...
  sched();
  p->chan = 0;
  if(lk != &ptable.lock){
    release(&ptable.lock);
    acquire(lk);
  }
}
```

Managing `ptable.lock`

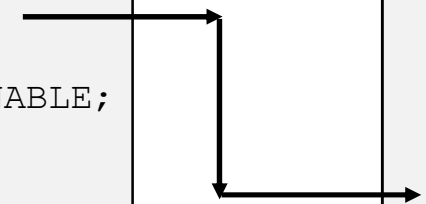
- Generally, the thread that has acquired a lock is responsible for releasing.
- However, thread that is calling `sched()` is not.
- `ptable.lock` is acquired before `swtch()` calling
- the thread that is scheduled newly release `ptable.lock`.

process A

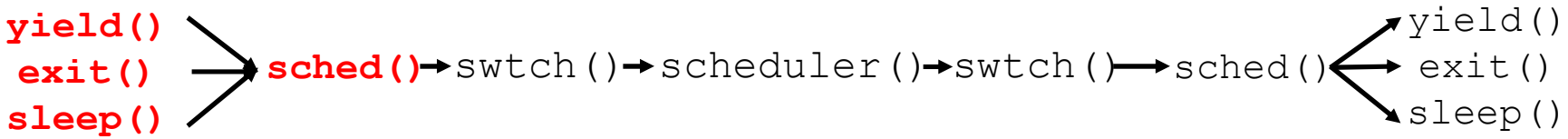
```
void yield(void){
  acquire(&ptable.lock);
  myproc()->state = RUNNABLE;
  sched();
  release(&ptable.lock);
}
```

process B

```
void yield(void){
  acquire(&ptable.lock);
  myproc()->state = RUNNABLE;
  sched();
  release(&ptable.lock);
}
```



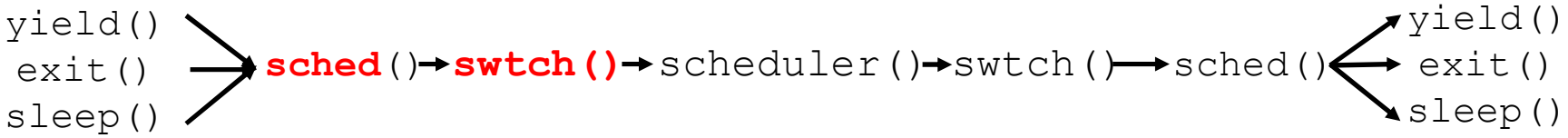
Scheduling pattern




```
void yield(void){
  acquire(&ptable.lock);
  myproc()->state = RUNNABLE;
  sched();
  release(&ptable.lock);
}
```

```
void sched(void){
  …
  swtch(&p->context, mycpu()->scheduler);
  mycpu()->intena = intena;
}
```

Scheduling pattern




```
void sched(void){
  …
  swtch(&p->context, mycpu()->scheduler);
  mycpu()->intena = intena;
}
```

```
swtch:
… …
popl %edi
popl %esi
popl %ebx
popl %ebp
ret
```

Scheduling pattern


```
swtch:
… …
popl %edi
popl %esi
popl %ebx
popl %ebp
ret
```

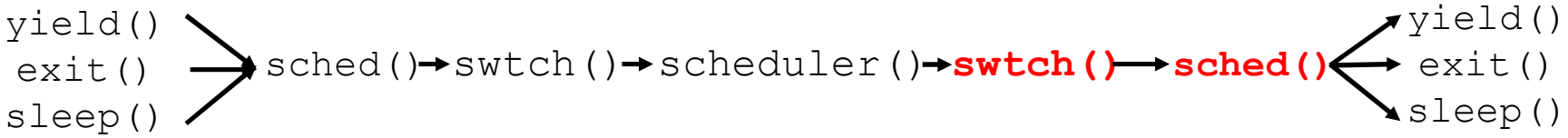
```
void scheduler(void) {
  …
  for(;;){
    …
    for(p=ptable.proc;p<&ptable.proc[NPROC];p++){
      …
      swtch(&(c->scheduler), p->context);
      switchkvm();
      …
    }
    …
  }
}
```

Scheduling pattern


```
void scheduler(void) {
  …
  for(;;){
    …
    for(p=ptable.proc;p<&ptable.proc[NPROC];p++){
      …
      swtch(&(c->scheduler), p->context);
      switchkvm();
      …
    }
    …
  }
}
```

```
swtch:
… …
popl %edi
popl %esi
popl %ebx
popl %ebp
ret
```

Scheduling pattern




```
swtch:
… …
popl %edi
popl %esi
popl %ebx
popl %ebp
ret
```

```
void sched(void){
  …
  swtch(&p->context, mycpu()->scheduler);
  mycpu()->intena = intena;
}
```

start scheduler

- `mpmain()` is started in `main()`.

```
int
main(void)
{
  ...
  startothers();   // start other processors
  kinit2(P2V(4*1024*1024), P2V(PHYSTOP)); // must come after startothers()
  userinit();      // first user process
  mpmain();        // finish this processor's setup
}
```

Start scheduler(Cont.)

- `scheduler()` is started in `mpmain()`.

```
static void
mpmain(void)
{
  cprintf("cpu%d: starting %d\n", cpuid(), cpuid());
  idtinit();     // load idt register
  xchg(&(mycpu()->started), 1); // tell startothers() we're up
  scheduler();  // start running processes
}
```

scheduler()

- The scheduler loops over the process table looking for a runnable process, one that has `p->state == RUNNABLE`.

- Once it finds a process
  - it sets the per-CPU current process variable `proc`
  - switches to the process's page table with `switchuvm()`
  - marks the process as `RUNNING`
  - and then calls `swtch()` to start running it

scheduler()

```
void scheduler(void){
  struct proc *p;
  struct cpu *c = mycpu();
  c->proc = 0;

  for(;;){
    // Enable interrupts on this processor.
    sti();

    // Loop over process table looking for process to run.
    acquire(&ptable.lock);
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      ...
      c->proc = p;
      switchuvm(p);
      p->state = RUNNING;

      swtch(&(c->scheduler), p->context);
      switchkvm();

      // Process is done running for now.
      // It should have changed its p->state before coming back.
      c->proc = 0;
    }
      ...
  }
}
```

scheduler()

```
void scheduler(void){
  struct proc *p;
  struct cpu *c = mycpu();
  c->proc = 0;

  for(;;){
    // Enable interrupts on this processor.
    sti();

    // Loop over process table looking for process to run.
    acquire(&ptable.lock);
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->state != RUNNABLE)
        continue;
      ...

      swtch(&(c->scheduler), p->context);
      switchkvm();

      ...
    }
    release(&ptable.lock);

  }
}
```

`scheduler()`

- Outer loop
  - Interrupt is enabled.: if the scheduler left interrupts disabled all the time, the I/O would never arrive.
  - `ptable.lock` is released at the end of each iteration.
    - If an idling scheduler looped with the lock continuously held, no other CPU that was running a process could ever perform a context switch or any process-related system call.
    - can never mark a process as `RUNNABLE` so as to break the idling CPU out of its scheduling loop.

mycpu() and myproc()

- struct cpu
  - contains per-processor state: the currently running process, hardware id for that processor (apicid).
  - When a processor must find it's per-cpu state, it reads its identifier from its local APIC and uses that identifier to find its state in the array.

```
// Per-CPU state
struct cpu {
  uchar apicid;                // Local APIC ID
  struct context *scheduler;   // swtch() here to enter scheduler
  struct taskstate ts;         // Used by x86 to find stack for interrupt
  struct segdesc gdt[NSEGS];   // x86 global descriptor table
  volatile uint started;       // Has the CPU started?
  int ncli;                    // Depth of pushcli nesting.
  int intena;                  // Were interrupts enabled before pushcli?
  struct proc *proc;           // The process running on this cpu or null
};
```

mycpu()

- Scan the array of a `struct cpu` and returns the address of `struct cpu`.
- inefficient!!!

```
struct cpu* mycpu(void) {
  int apicid, i;
  if(readeflags()&FL_IF)
    panic("mycpu called with interrupts enabled\n");

  apicid = lapicid();

  for (i = 0; i < ncpu; ++i) {
    if (cpus[i].apicid == apicid)
      return &cpus[i];
  }
  panic("unknown apicid\n");
}
```

myproc()

- find the `struct proc` for the process that is running on the current processor.
- `myproc()` disables interrupts, and invokes `mycpu()`.

```
struct proc* myproc(void) {
        struct cpu *c;
        struct proc *p;
        pushcli();
        c = mycpu();
        p = c->proc;
        popcli();
        return p;
}
```

`sleep` and `wakeup`

- Let the processes to interact with each other!
- Sleep and wakeup allows one process to sleep waiting for an event and another process to wake it up once the event has happened.
- Sleep and wakeup are often called sequence coordination or conditional synchronization mechanisms.
- Make sure that they do not miss each other!!!

producer/consumer with busy waiting

- Operation
  - `send()`: loops until the queue is empty and then puts the pointer p in the queue.
  - `recv()`: loops until the queue is non-empty and takes the pointer out.
- Problem: waste of CPU
  - If the sender sends rarely, the receiver will spend most of its time spinning(busy).

```
100 struct q {
101     void *ptr;
102 };
103
104 void*
105 send(struct q *q, void *p)
106 {
107     while(q->ptr != 0)
108         ;
109     q->ptr = p;
110 }
```

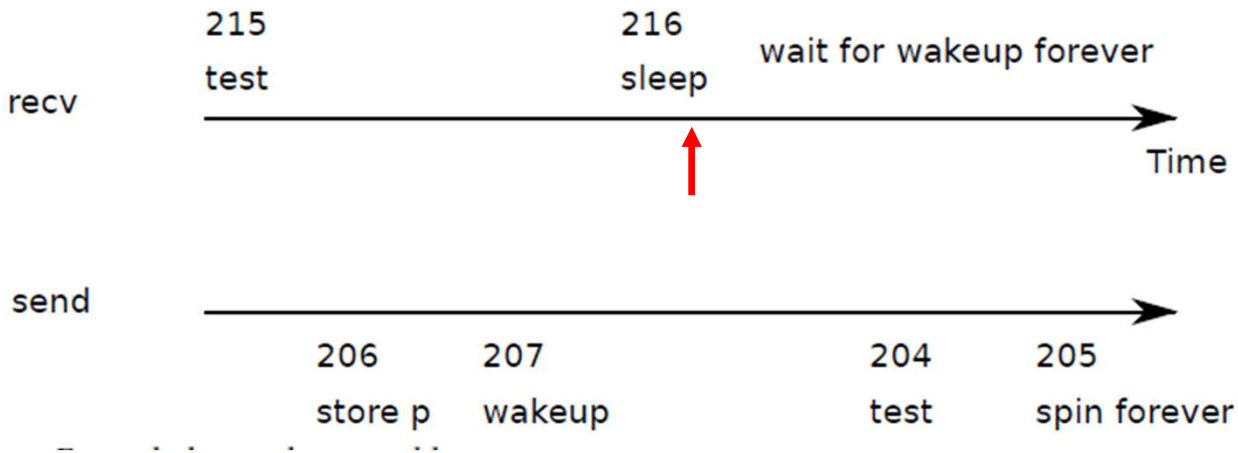
```
112 void*
113 recv(struct q *q)
114 {
115     void *p;
116
117     while((p = q->ptr) == 0)
118         ;
119     q->ptr = 0;
120     return p;
121 }
```

producer/consumer with `sleep` and `wake up`

- Problem: lost wakeup

```
201 void*
202 send(struct q *q, void *p)
203 {
204       while(q->ptr != 0)
205             ;
206       q->ptr = p;
207       wakeup(q);
208 }
```

```
210 void*
211 recv(struct q *q)
212 {
213       void *p;
214   test
215       while((p = q->ptr) == 0)
216             sleep(q);
217       q->ptr = 0;
218       return p;
219 }
```



producer/consumer in `sleep` and `wake up`

- Incorrect solution to lost wakup: deadlock
  - While receiver is waiting, the send cannot send.

```
400 struct q {
401       struct spinlock lock;
402       void *ptr;
403 };
404
405 void*
406 send(struct q *q, void *p)
407 {
408       acquire(&q->lock);
409       while(q->ptr != 0)
410             ;
411       q->ptr = p;
412       wakeup(q);
413       release(&q->lock);
414 }
```

???

```
415
416 void*
417 recv(struct q *q)
418 {
419       void *p;
420
421       acquire(&q->lock);
422       while((p = q->ptr) == 0)
423             sleep(q);
424       q->ptr = 0;
425       release(&q->lock);
426       return p;
427 }
```

Sleep with lock held.

producer/consumer with `sleep` and `wakeup`

within sleep, release the lock before it sets the process to sleep.

```
400 struct q {
401       struct spinlock lock;
402       void *ptr;
403 };
404
405 void*
406 send(struct q *q, void *p)
407 {
408       acquire(&q->lock);
409       while(q->ptr != 0)
410             ;
411       q->ptr = p;
412       wakeup(q);
413       release(&q->lock);
414 }
```

```
415
416 void*
417 recv(struct q *q)
418 {
419       void *p;
420
421       acquire(&q->lock);
422       while((p = q->ptr) == 0)
423             sleep(q, &q->lock);
424       q->ptr = 0;
425       release(&q->lock);
426       return p;
427 }
```

code: `sleep`

- `sleep()`: mark the current process as `SLEEPING` and then call `sched()` to release the processor.

sleep()

```
2873 void
2874 sleep(void *chan, struct spinlock *lk) {
2876       struct proc *p = myproc();
2877
2878       if(p == 0)
2879             panic("sleep");
2881       if(lk == 0)
2882             panic("sleep without lk");
2890       if(lk != &ptable.lock){
2891             acquire(&ptable.lock);
2892             release(lk);//release the lock.
2893       }
2895       p->chan = chan;
2896       p->state = SLEEPING;
2898       sched();
2901       p->chan = 0;
2904       if(lk != &ptable.lock){
2905             release(&ptable.lock);
2906             acquire(lk);//acquire the lock.
2907       }
2908 }
```

Hand CPU over.

Get back from sleep and wake up.

Code: `sleep()` and `wakeup()`

- `wakeup()` looks for a process sleeping on the given wait channel and marks it as `RUNNABLE`.

```
2962 // Wake up all processes sleeping on chan.
2963 void
2964 wakeup(void *chan)
2965 {
2966      acquire(&ptable.lock);
2967      wakeup1(chan);
2968      release(&ptable.lock);
2969 }
```

wakeup

```
2950 // Wake up all processes sleeping on chan.
2951 // The ptable lock must be held.
2952 static void
2953 wakeup1(void *chan)
2954 {
2955      struct proc *p;
2956
2957      for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
2958           if(p->state == SLEEPING && p->chan == chan)
2959                p->state = RUNNABLE;
2960 }
```

- Put all threads waiting for the channel to `RUNNABLE`.
- Inefficient: O(n)

Summary

- `switch`: It switch the process to another process for running on the CPU.
- `sched()`: Release the CPU and switch to scheduler thread.
- `schedule()`: Called by Independent thread by each CPU, switch to new `RUNNABLE` thread
- `sleep()` and `wakeup()`