Locking

- Two operations
  - acquire
  - release

- The code between acquire and release is called critical section

- Locking ensures mutual exclusion
  - i.e., two threads cannot execute the critical section at the same time

Locking

acquire ( )

critical section

release( )

Locking

- We have discussed uniprocessor locking so far
- We will discuss multiprocessor locking in future lectures
- For now, you can assume the implementation of the acquire and release as black box
- You can directly use these interfaces to ensure mutual exclusion

Lock interfaces with lock variables

- sema_down
- spin_lock
- etc.

Lock interfaces with lock variables

struct lock a, b;

thread 1:

```
lock_init (&a, 1);
lock_acquire (&a);
critical_section1();
lock_release (&a);
```

thread 2:

```
lock_init (&b, 1);
lock_acquire (&b);
critical_section2();
lock_release (&b);
```

critical_section1() and critical_section2() can execute in parallel.

Scheduling in xv6

- Sleep
  - waiting for an event
  - waiting for I/O
  - sleep system call

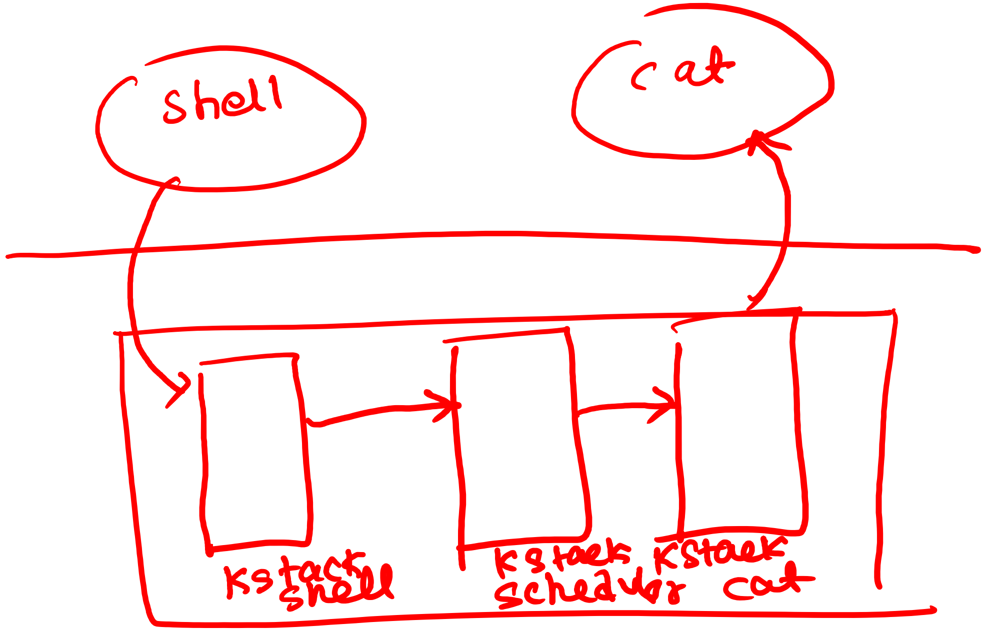
- Process has completed its time slice
  - e.g., on a timer interrupt

Challenges

- When to do context_switch
  - timer interrupt
  - sleep
- How to support concurrent calls to scheduler
  - xv6 uses locks

context_switch

Figure 5-1.



Scheduler

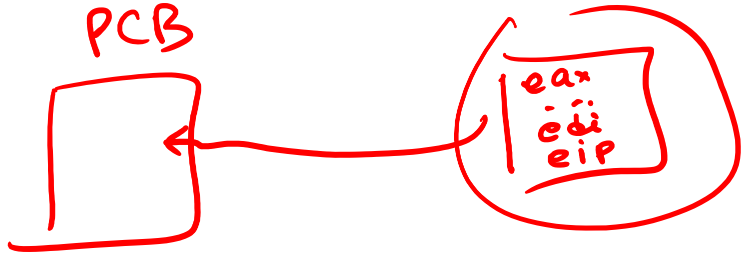
- Scheduler maintains a list of all processes
- In xv6, all processes are single threaded
- Scheduler maintains a list of process control blocks (PCB)
- PCB contains a pointer to the process context

Process context

- Each thread has its own set of CPU registers, stack, and instruction pointer
- But the processor has only one set of CPU registers and instruction pointers
- The registers are multiplex among all the threads

Process context

- The registers are multiplexed by saving them in the process context
- The EIP is also saved in the process context
- On context switch, the process allocates space for the current process's context, save registers in process context, and save the address on process context in the PCB



What values need to be saved in the process context

eax
ecx
edx
esp
ebp
esi
edi
ebx
eip

What values need to be saved in the process context

eip

ebp

ebx

esi

edi

ecx

edx

eax

esp

swtch:2958

eip
ebp
ebx
esi
edi ← process_context == esp
~~ecx~~
~~edx~~
~~eax~~
esp

& PCB → context

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

& PCB_OLD → context,
PCB_New → context;

(struct context **old, struct context *new)
{
push ebp
push ebx
push esi
push edi
*old = esp
esp = new

pop edi
pop esi
pop ebx
pop ebp
pop

scheduler:2708

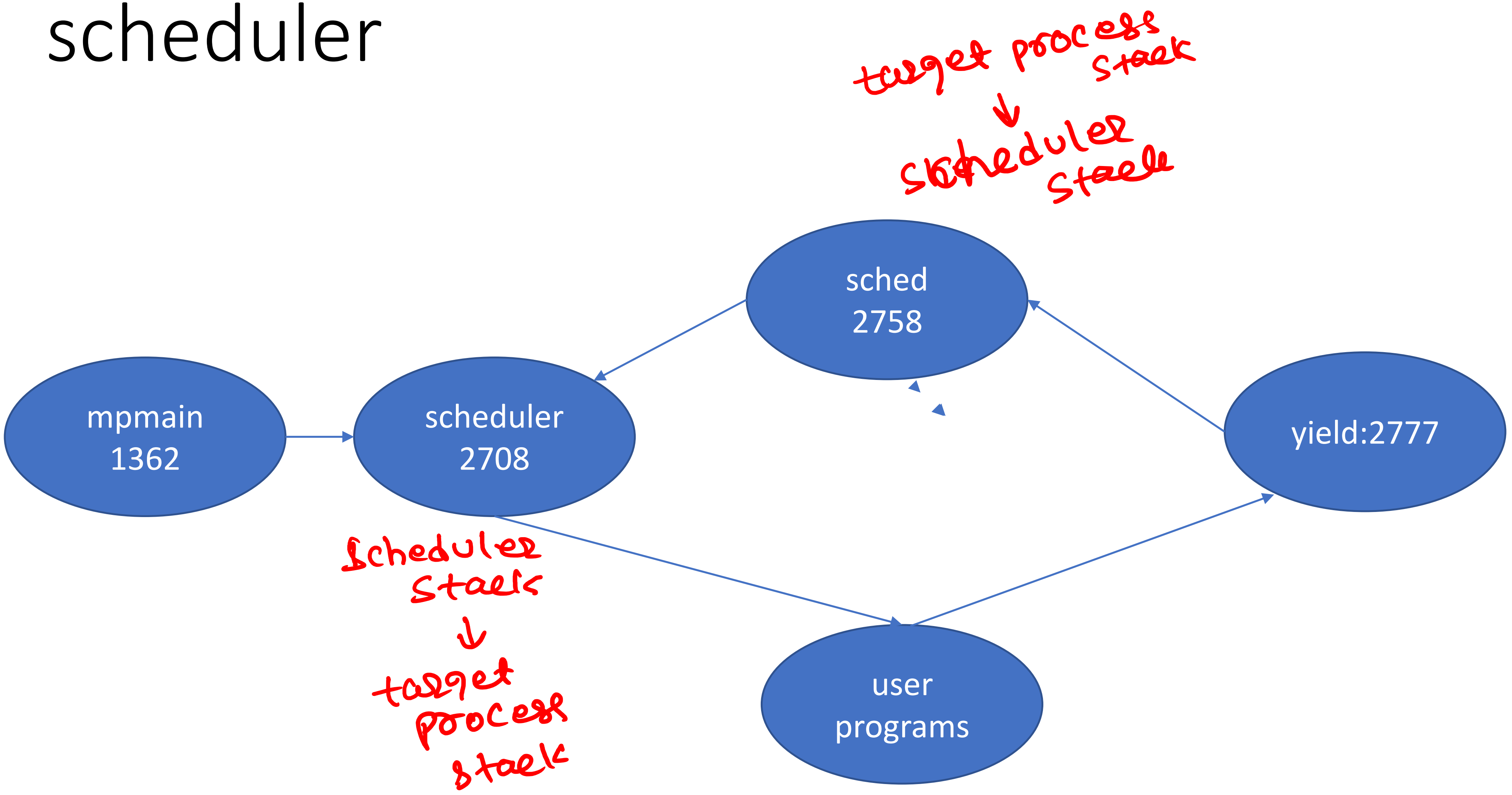
- scheduler is called on scheduler stack
  - scheduler is called when the first process is created

mpmain → scheduler

yield:2777

- yield:2777 is called when the current process time slice expires
  - yield calls sched:2758 to save the context of current process and restore the context of scheduler

scheduler



target process stack
↓
scheduler stack
sched
2758
mpmain
1362
scheduler
2708
yield:2777
user
programs
scheduler stack
↓
target process stack

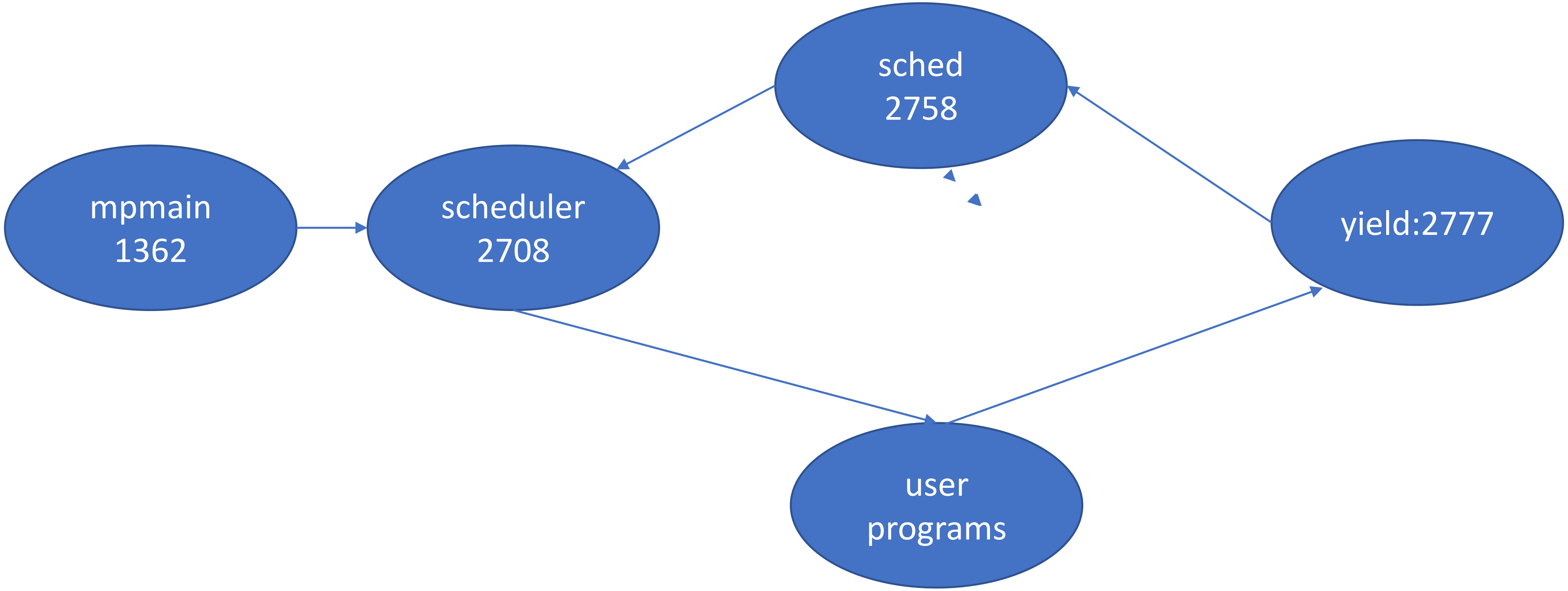
PCB list

- Scheduler maintains a list of PCBs of all processes
  - Why process not thread?

- Scheduler walks to PCB list to find a runnable process for scheduling

- fork:2556 creates a new process and add it to PCB list
  - fork system call is the only way to create a new process after the first user process is loaded in the user space

scheduler



sched
2758
mpmain
1362
scheduler
2708
yield:2777
user
programs

ptable.lock

- ptable.lock is used to protect the PCB

- what could go wrong if ptable.lock is not acquired in scheduler and yield
  - concurrent calls to yield and scheduler may try to schedule and yield the same process concurrently

process 1 yield

acquire (&ptable.lock);

sched();

switch (scheduler);

switchkvm();

switch (process 1);

2772

release (&ptable.lock);

Concurrency without ptable lock

- “process1” does the yield system call
- yield:2780 sets the state as RUNNABLE
- scheduler:2708 is called on the other core
- scheduler finds “process1” RUNNABLE
- Both scheduler and sched call swtch to restore/save “process1” context at the same time

What is cpu->scheduler

- CPU1:

yield()

acquire (&ptable.lock)

sched:

load cpu->scheduler context

schedule process1

save cpu->scheduler context

release (&ptable.lock)

- CPU2:

yield()

acquire (&ptable.lock);

sched:

load cpu->scheduler context

but cpu->scheduler contains context of CPU1

“cpu” is per-CPU variable

- “cpu” points to different memory locations for different CPUs

- mpmain() is called for each CPU that eventually calls the scheduler on each CPU

scheduler and yield disable interrupts

- acquire(&ptable.lock) internally disables interrupt
  - why we need to disable interrupts in scheduler

scheduler:2708

- Why acquire() and release() inside the outer for loop?
  - Why not outside the outer for loop?

Sleep and wakeup

```
struct q {
    void *ptr;
};

void *send (struct q *q, void *p) {
   while (q->ptr != 0){
   }
   q->ptr = p;
}
```

```
void *recv (struct q *q) {
    void *p;
    while ((p = q->ptr) == 0) {
    }
    q->ptr = 0;
     return p;
}
```

Not good if sender sends rarely!

Sleep and wakeup

```
struct q {
    void *ptr;
};

void *send (struct q *q, void *p) {
   while (q->ptr != 0) {
   }
   q->ptr = p;
   wakeup (q);
}
```

```
void *recv (struct q *q) {
    void *p;
    while ((p = q->ptr) == 0) {
        sleep (q);
    }
    q->ptr = 0;
     return p;
}
```

Sleep and wakeup

```
struct q {
    struct spinlock lock;
    void *ptr;
};

void *send (struct q *q, void *p) {
   acquire (&q->lock);
   while (q->ptr != 0) {
   }
   q->ptr = p;
   wakeup (q);
   release (&q->lock);
}
```

```
void *recv (struct q *q) {
    void *p;
    acquire (&q->lock);
    while ((p = q->ptr) == 0) {
        sleep (q);
    }
    q->ptr = 0;
    release (&q->lock);
    return p;
}
```

Sleep and wakeup

```
struct q {
    struct spinlock lock;
    void *ptr;
};

void *send (struct q *q, void *p) {
   acquire (&q->lock);
   while (q->ptr != 0) {
   }
   q->ptr = p;
   wakeup (q);
   release (&q->lock);
}
```

```
void *recv (struct q *q) {
    void *p;
    acquire (&q->lock);
    while ((p = q->ptr) == 0) {
        sleep (q, &q->lock);
    }
    q->ptr = 0;
    release (&q->lock);
    return p;
}
```

sleep:2809

- How does sleep ensure that wakeup will not be lost?

- What is the check at 2823?

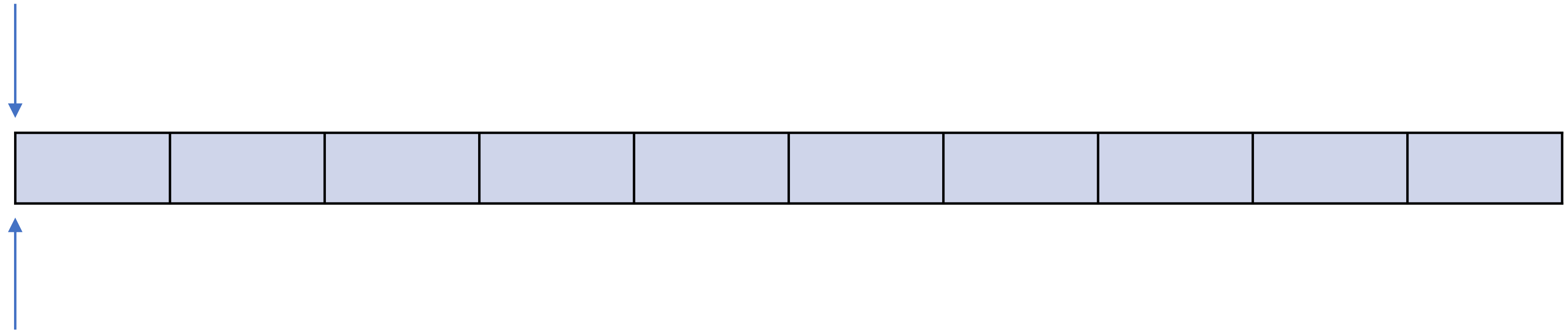
wakeup:2864

- What could go wrong if wakeup does not acquire ptable.lock?
- Can we do it more efficiently?

pipe

- pipe is a circular buffer of size PIPESIZE
  - pipe_is_empty = nread == nwrite
  - pipe_is_full = nwrite - nread == PIPESIZE

nread = number of bytes read



nwrite = number of bytes written

piperead:6551

- piperead (struct pipe *p, char *addr, int n);
  - tries to read n bytes from the pipe (p) to buffer (addr)
  - returns the number of bytes read
  - if the pipe is empty wait until some bytes are written to the pipe

pipewrite:6530

- pipewrite (struct pipe *p, char *addr, int n);
  - writes n bytes from the buffer (addr) to pipe (p)
  - if the pipe is full waits until a reader consumes some bytes

reader
is
sleeping

exit and wait

exit

- Mark the status of the current process as zombie
- If the parent has already exited, change the parent to init process
- wakeup the parent process
- call schedule

wait

- If any of the children is in zombie state, releases the resources of the child and returns the child pid
- If none of the children has exited, sleep until the child process wakes up

exit:2604

- Can exit free all the resources in this routine?

- Can wait:2653 free the child resources just after the exit:2604 sets the status of the exiting process to zombie?

kill:2875

- kill system call allows a process to kill another process
- killing a process at arbitrary points is tricky because the target process might be in the middle of something
  - e.g., holding a kernel lock
- kill:2875 simply sets a field “killed” in the target process PCB
- trap:3351 checks the killed flag before returning to user mode
  - if the current process was killed, trap calls the exit