UNIVERSITY OF PITTSBURGH
VERITAS 1787 VIRTUS

# CS 1550

## Week 5 – Synchronization with xv6

Teaching Assistant

Henrique Potter

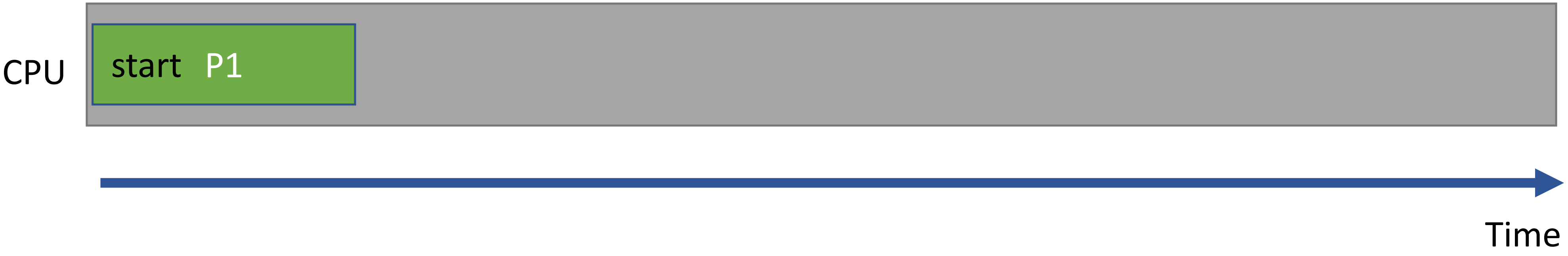
# CS 1550 – Lab 2 is out

- **Due**: Monday, February 17, 2020 @11:59pm
- **Late**: Wednesday, February 19, 2020 @11:59pm
  - 10% reduction per late day

# Keep in mind the different qemu

- qemu with xv6 (Labs)
  - Should be executed at **linux.cs.pitt.edu**
- qemu (Project 1)
  - Should be executed at **thoth.cs.pitt.edu**

# Locks – Processes without sharing CPU



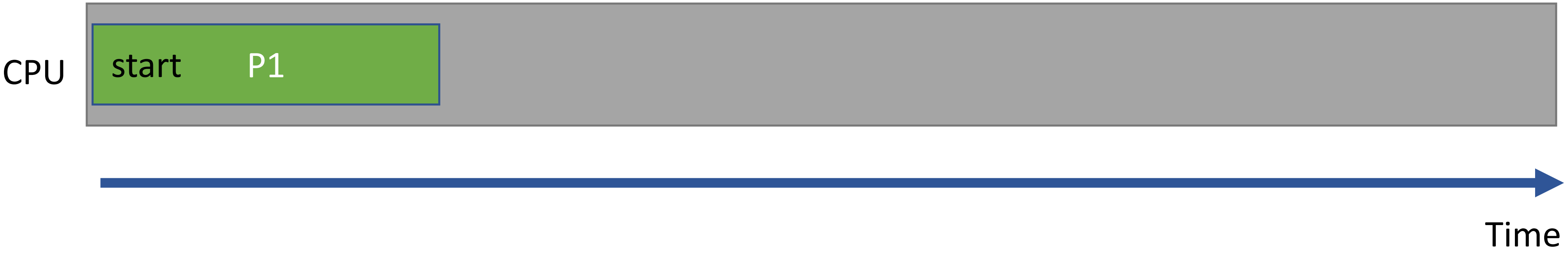
CPU

start P1

Time

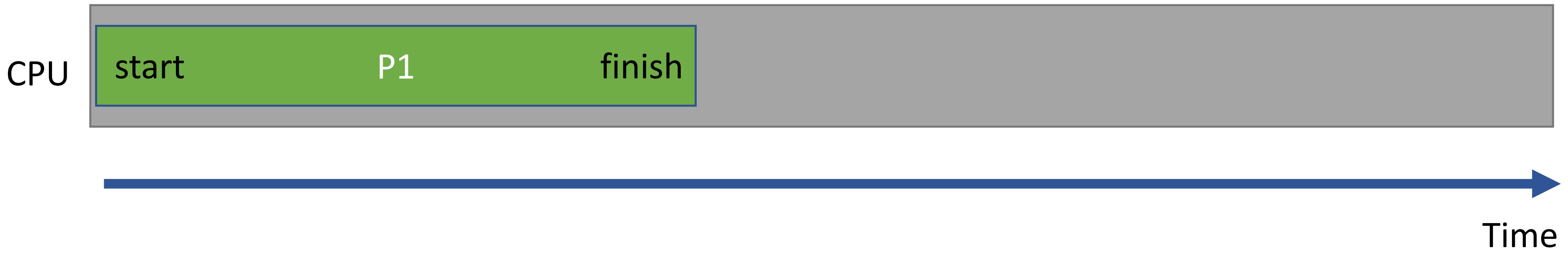
# Locks – Processes without sharing CPU

- With no sharing, process must run to completion.



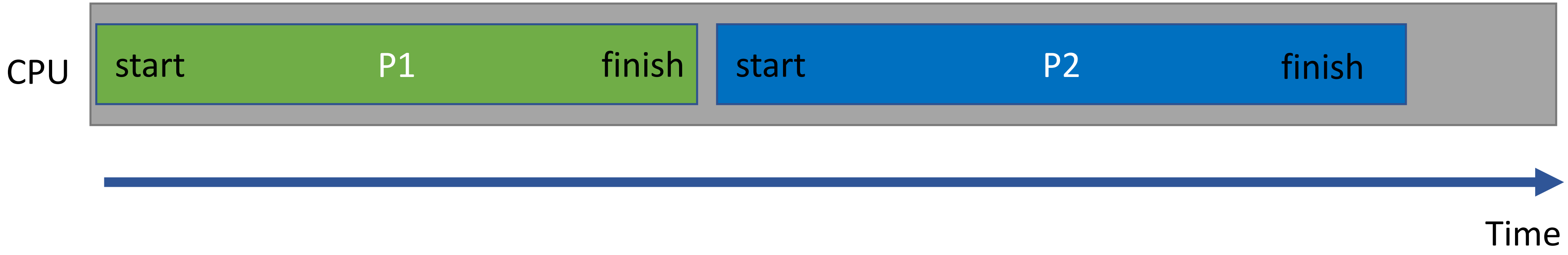
# Locks – Processes without sharing CPU

- With no sharing, process must run to completion.



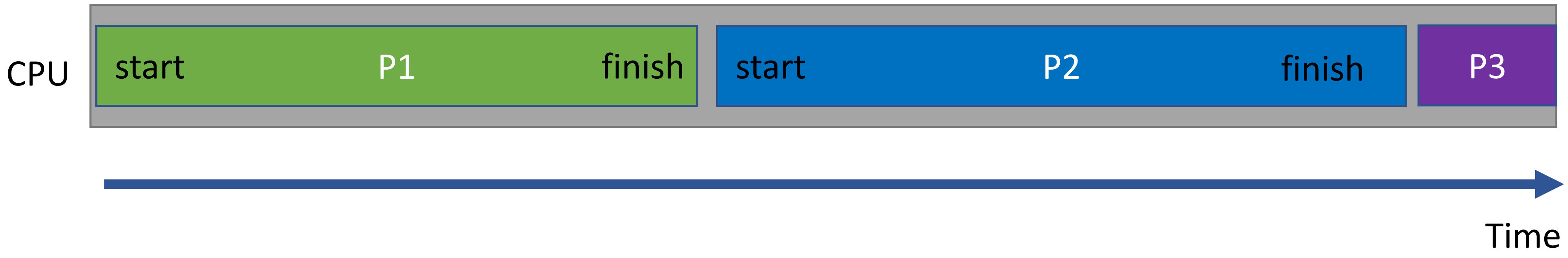
# Locks – Processes without sharing CPU

- **OS** chooses another processes to execute once the first finishes



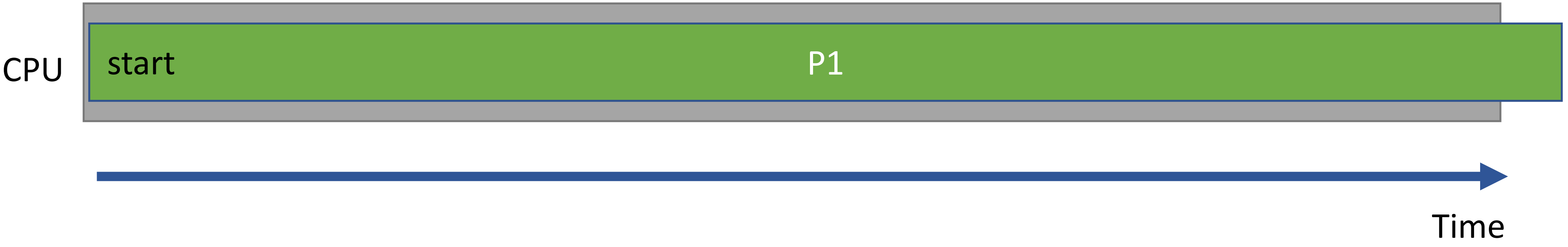
# Locks – Processes without sharing CPU

- **OS** chooses another processes to execute once the first finishes



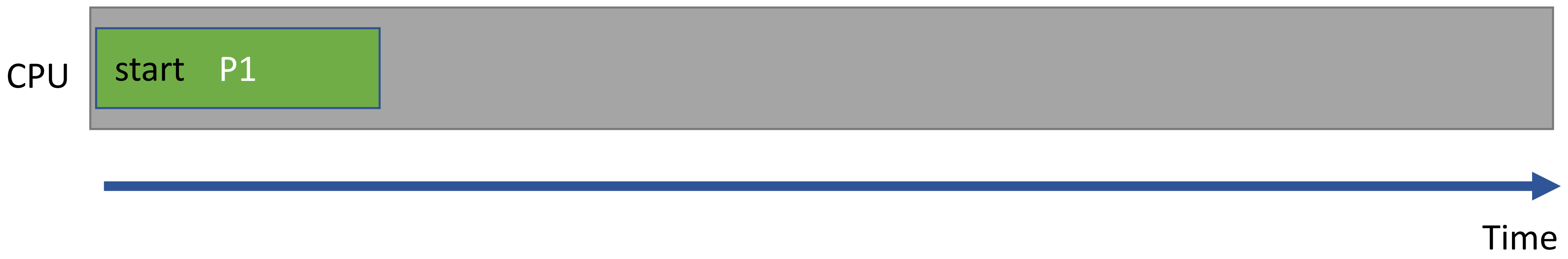
# Locks – Processes without sharing CPU

- What if P1 is a big process?



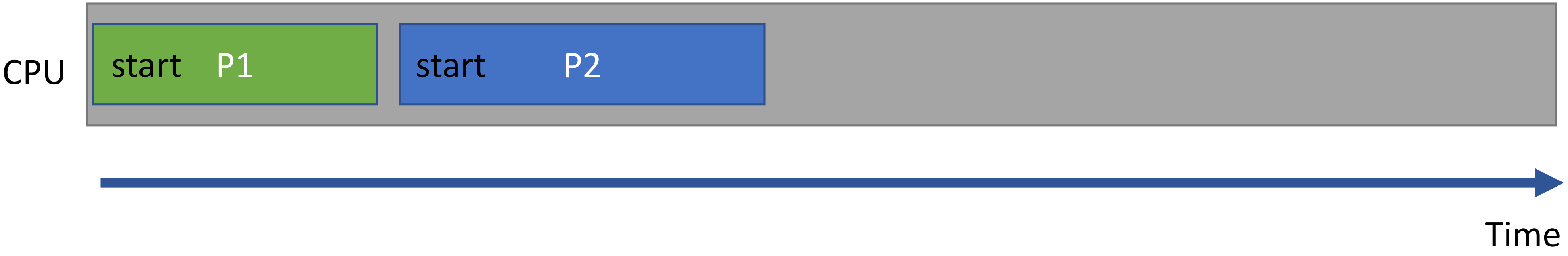
# Locks – Processes sharing CPU

- Solution switch processes during their execution.



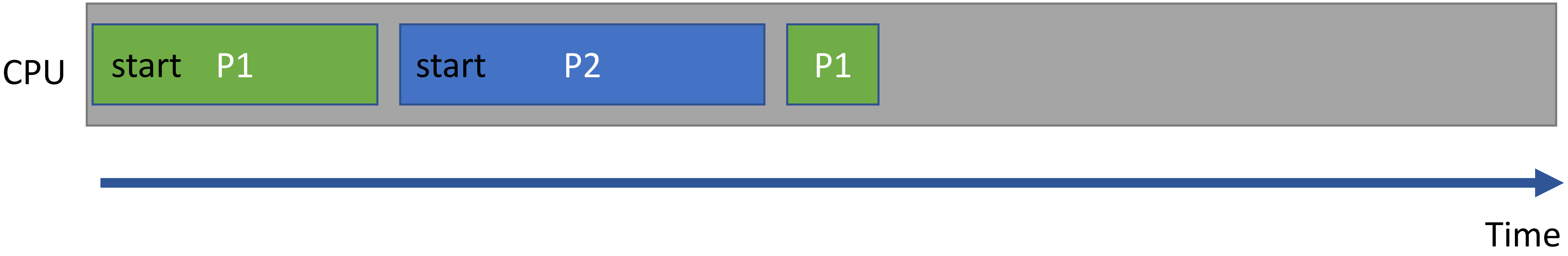
# Locks – Processes sharing CPU

- Solution switch processes during their execution.



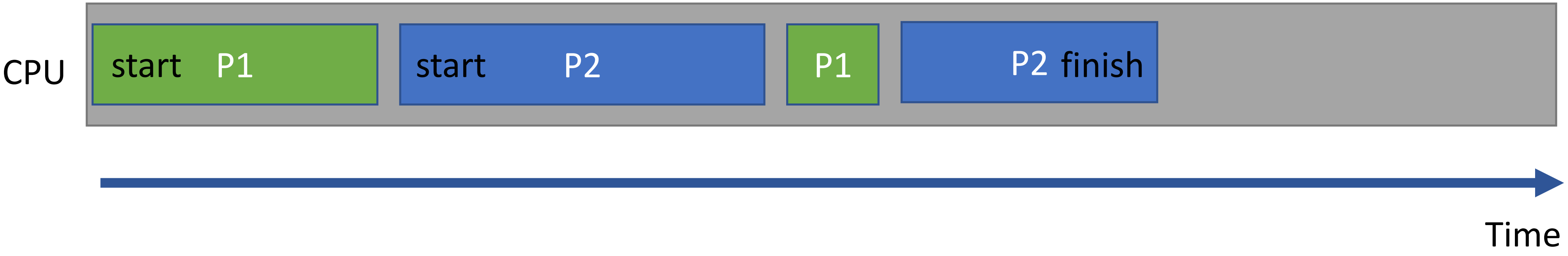
# Locks – Processes sharing CPU

- Solution switch processes during their execution.



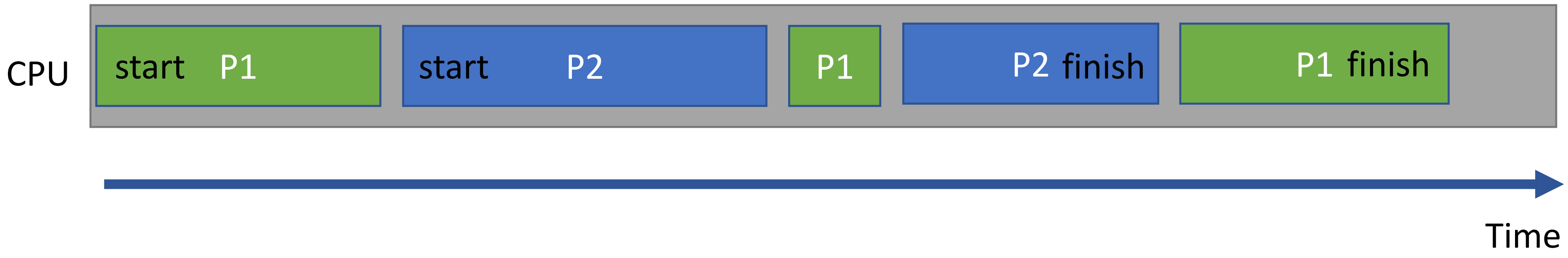
# Locks – Processes sharing CPU

- Solution switch processes during their execution.



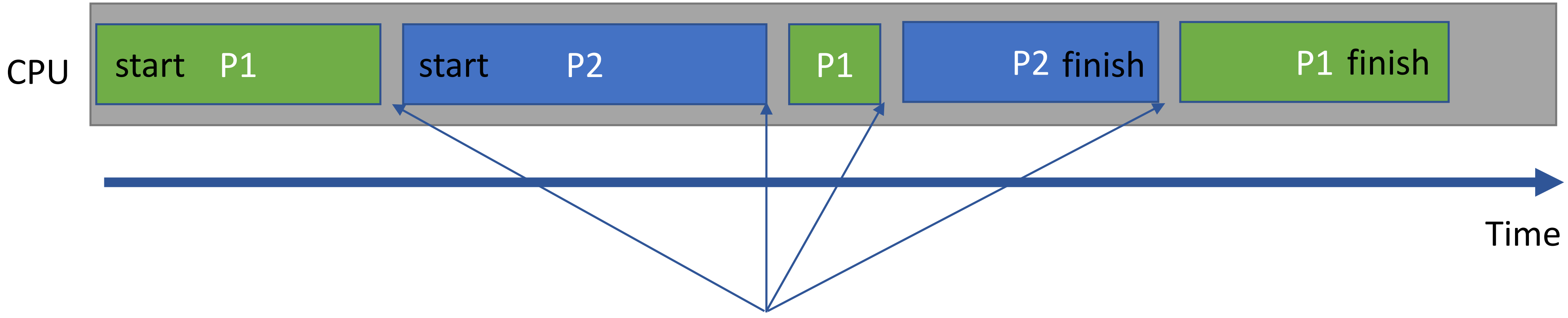
# Locks – Processes sharing CPU

- Solution switch processes during their execution.



# Locks – Processes sharing CPU

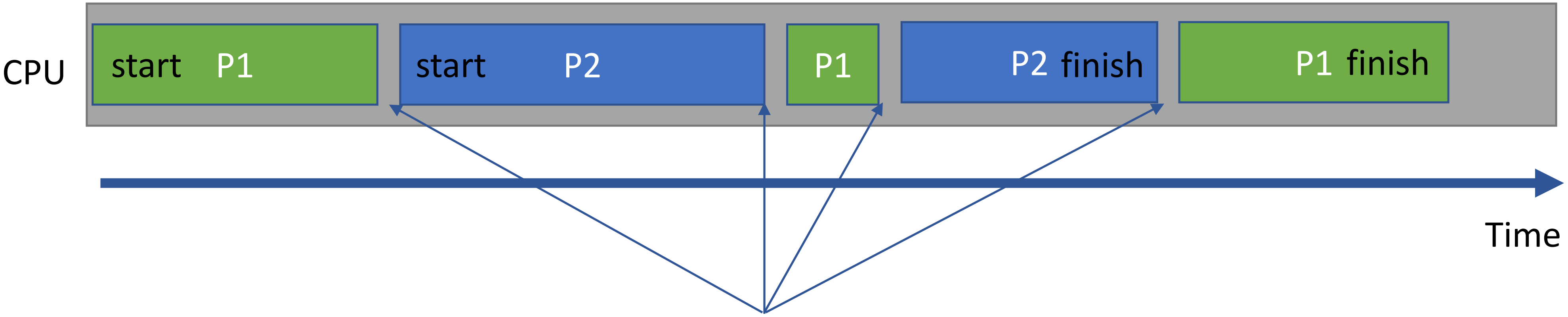
- Solution switch processes during their execution.



What is the little gap?

# Locks – Processes sharing CPU

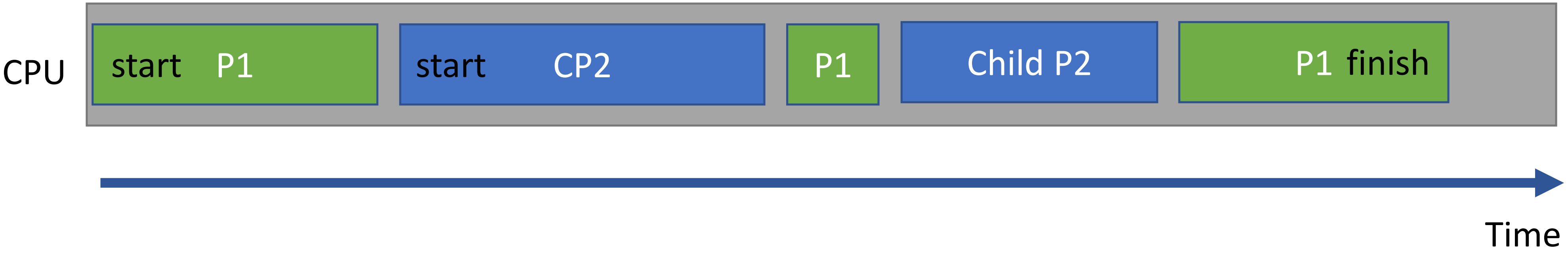
- Solution switch processes during their execution.



What is the little gap?
The OS **Scheduler**

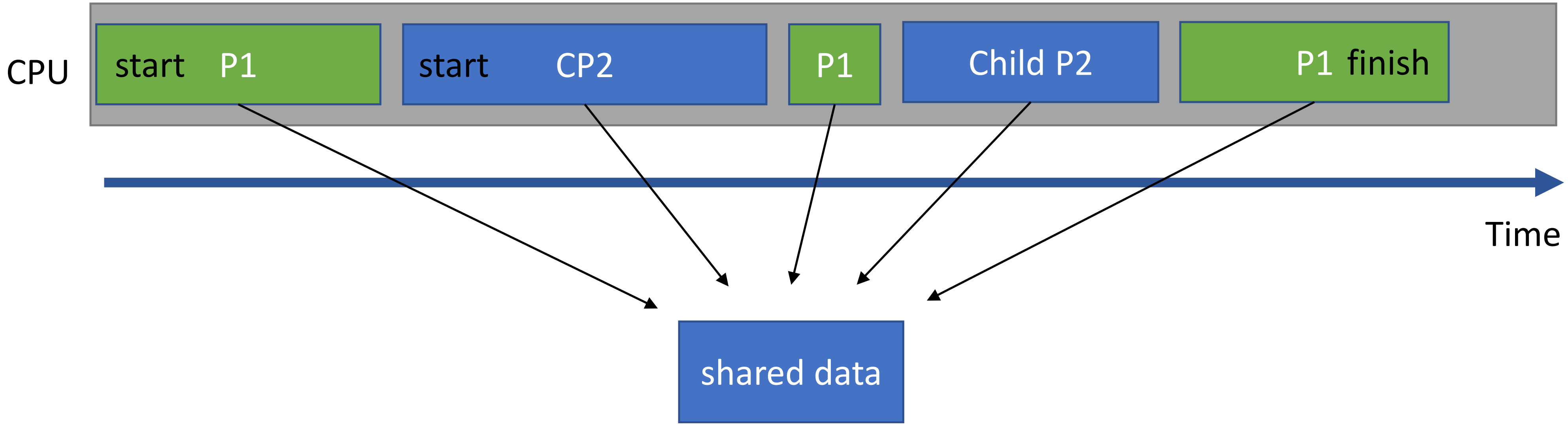
# Locks – Processes sharing CPU

- What happens in Parent-Child Process scenario?



# Locks – Processes sharing CPU

- What happens in Parent-Child Process scenario?
- How to keep integrity/correctness on race conditions?



# Locks – Processes sharing CPU

```
struct list {
  int data;
  struct list *next;
};
```

# Locks – Processes sharing CPU

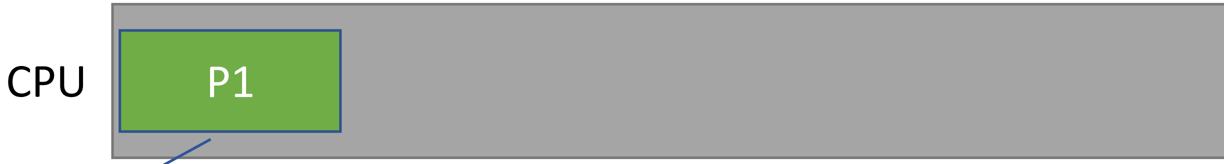
```
struct list {
  int data;
  struct list *next;
};

struct list *list = 0;
```

# Locks – Processes sharing CPU

```
struct list {
  int data;
  struct list *next;
};

struct list *list = 0;

void
insert(int data) {
      struct list *l;
      l = malloc(sizeof *l);
      l->data = data;
      l->next = list;
      list = l;
}
```

# Locks – Processes sharing CPU

```
struct list {
  int data;
  struct list *next;
};

struct list *list = 0;

void
insert(int data) {
      struct list *l;
      l = malloc(sizeof *l);
      l->data = data;
      l->next = list;
      list = l;
}
```

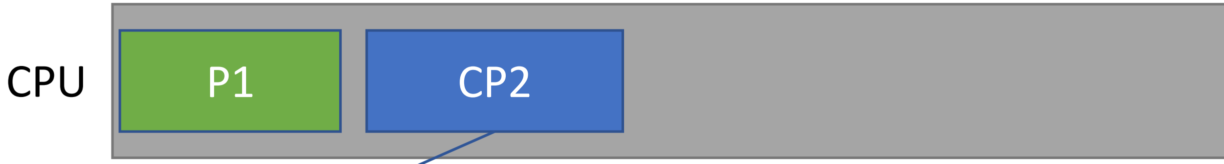


P1 stops here the
OS switches to P2

# Locks – Processes sharing CPU

```
struct list {
  int data;
  struct list *next;
};

struct list *list = 0;

void
insert(int data) {
      struct list *l;
      l = malloc(sizeof *l);
      l->data = data;
      l->next = list;
      list = l;
}
```

P1 stopped

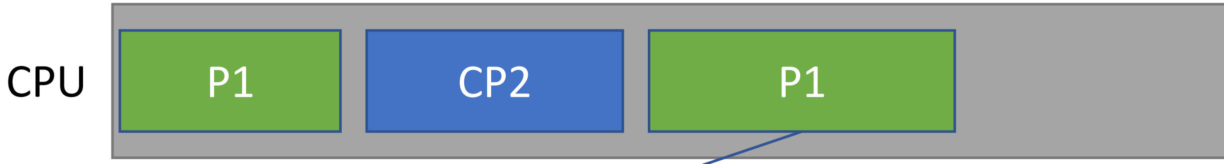


P2 gets the same reference to the same block of data of list and overwrites it

# Locks – Processes sharing CPU

```
struct list {
  int data;
  struct list *next;
};

struct list *list = 0;

void
insert(int data) {
      struct list *l;
      l = malloc(sizeof *l);
      l->data = data;
      l->next = list;
      list = l;
}
```

CP2 stopped



When P1 comes back it will have written the wrong data

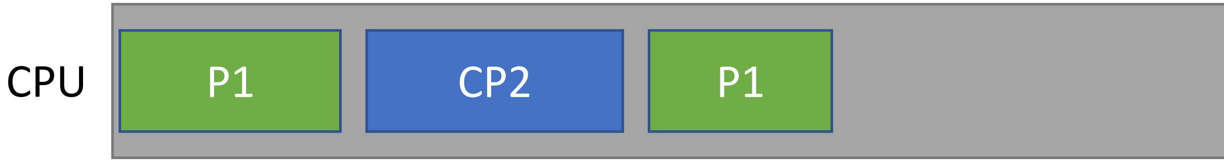
**Race condition**: A race condition is an undesirable condition that happened when having multiple processes running on a piece of data which does not use any exclusive locks to control access.

# Locks – Processes sharing CPU

- Sharing CPU among processes
- Ensuring data integrity/correctness
- Ensure that a **critical section** of your code is only executed by one process

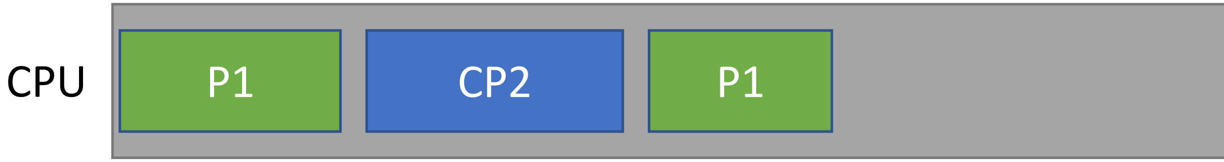
# Locks – Processes sharing CPU

```
struct list *list = 0;
struct spinlock listlock;

void
insert(int data)
{
        struct list *l;

        acquire(&listlock);
        l = malloc(sizeof *l);
        l->data = data;
        l->next = list;
        list = l;
        release(&listlock);
}
```



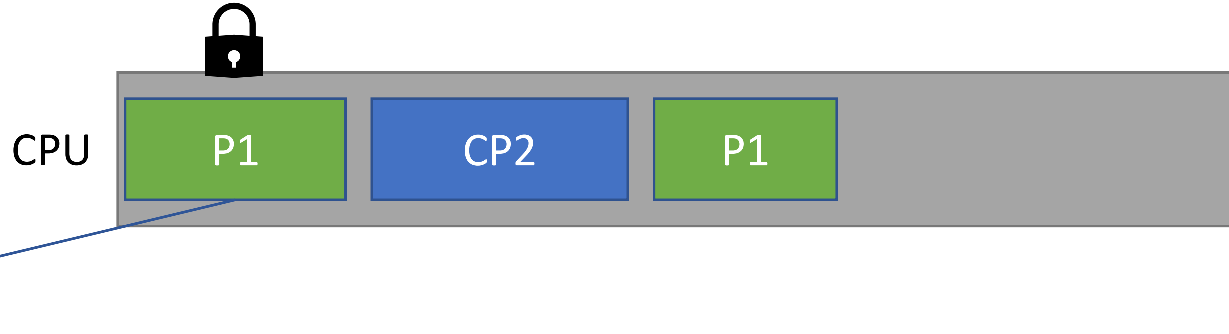
# Locks – Processes sharing CPU

```
struct list *list = 0;
struct spinlock listlock;

void
insert(int data)
{
      struct list *l;

      acquire(&listlock);
      l = malloc(sizeof *l);
      l->data = data;
      l->next = list;
      list = l;
      release(&listlock);
}
```



# Locks – Processes sharing CPU

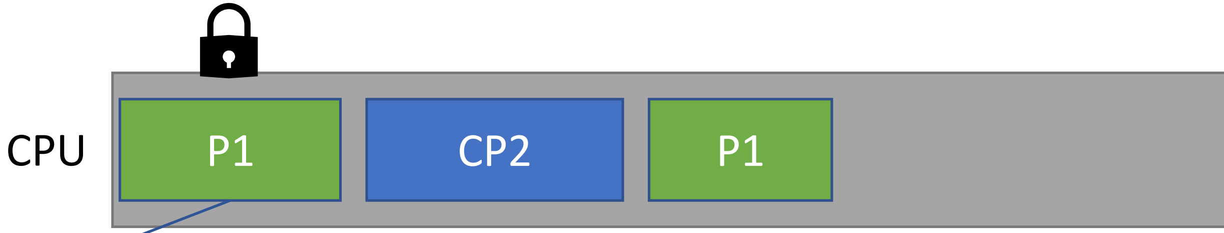
```
struct list *list = 0;
struct spinlock listlock;

void
insert(int data)
{
        struct list *l;

        acquire(&listlock);
        l = malloc(sizeof *l);
        l->data = data;
        l->next = list;
        list = l;
        release(&listlock);
}
```



P1 gets locks the lock

# Locks – Processes sharing CPU

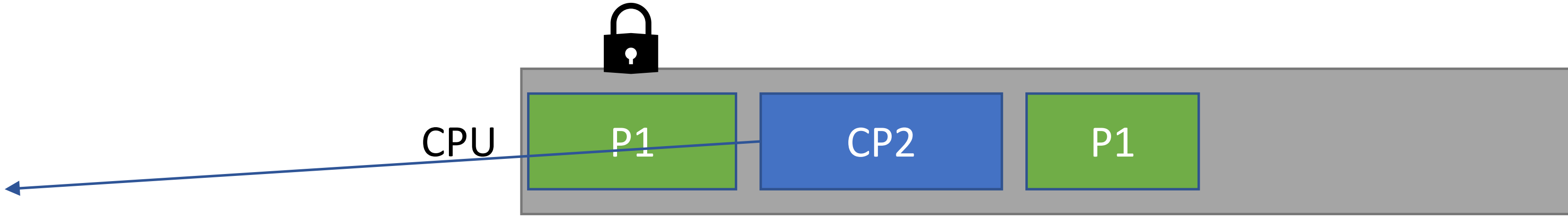
```
struct list *list = 0;
struct spinlock listlock;

void
insert(int data)
{
        struct list *l;

        acquire(&listlock);
        l = malloc(sizeof *l);
        l->data = data;
        l->next = list;
        list = l;
        release(&listlock);
}
```



P1 gets locks the lock

# Locks – Processes sharing CPU

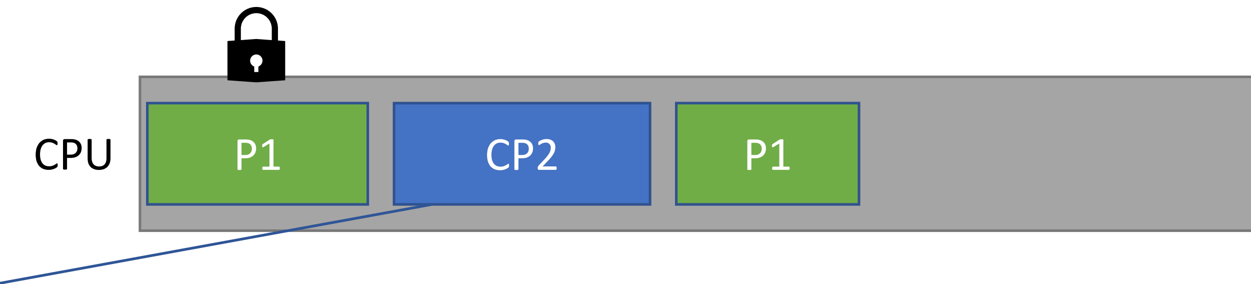
```
struct list *list = 0;
struct spinlock listlock;

void
insert(int data)
{
      struct list *l;

      acquire(&listlock);
      l = malloc(sizeof *l);
P1 stopped   l->data = data;
      l->next = list;
      list = l;
      release(&listlock);
}
```



When the OS schedule CP2

# Locks – Processes sharing CPU

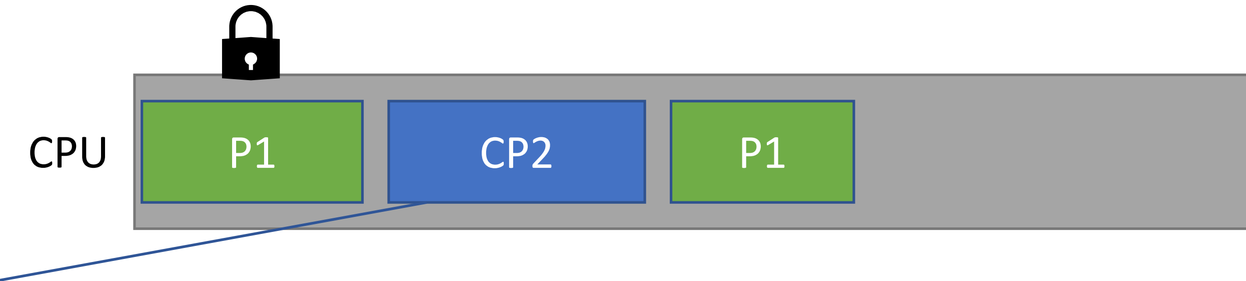
```
struct list *list = 0;
struct spinlock listlock;

void
insert(int data)
{
        struct list *l;

        acquire(&listlock);
        l = malloc(sizeof *l);
P1 stopped  l->data = data;
        l->next = list;
        list = l;
        release(&listlock);
}
```



It will try to get the lock but won’t.

# Locks – Processes sharing CPU

```
struct list *list = 0;
struct spinlock listlock;

void
insert(int data)
{
        struct list *l;

        acquire(&listlock);
        l = malloc(sizeof *l);
P1 stopped   l->data = data;
        l->next = list;
        list = l;
        release(&listlock);
}
```

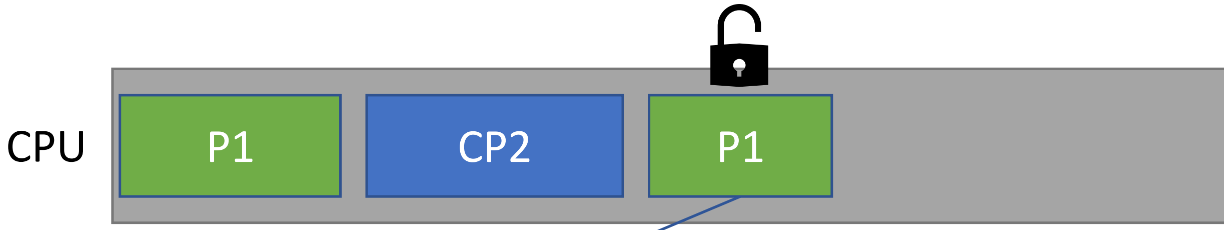


It will try to get the lock but won’t.

It will be constantly try to get it ( in a loop).
Until the OS switches back to P1

# Locks – Processes sharing CPU

```
struct list *list = 0;
struct spinlock listlock;

void
insert(int data)
{
      struct list *l;

CP2 stopped   acquire(&listlock);
      l = malloc(sizeof *l);
      l->data = data;
      l->next = list;
      list = l;
      release(&listlock);
}
```

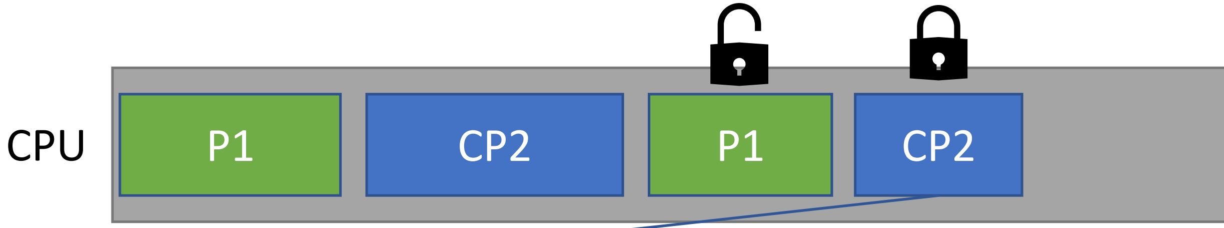


P1 release the lock P2 will finally be able to execute, once scheduled

# Locks – Processes sharing CPU

```
struct list *list = 0;
struct spinlock listlock;

void
insert(int data)
{
        struct list *l;

        acquire(&listlock);
        l = malloc(sizeof *l);
        l->data = data;
        l->next = list;
        list = l;
        release(&listlock);
}
```

CP2 proceeds



P1 release the lock P2 will finally be able to execute, once scheduled

# Locks – Processes sharing CPU

- SpinLock

```
Void
acquire(struct spinlock *lk)
{
    for(;;) {
        if(!lk->locked) {
            lk->locked = 1;
            break;
        }
    }
}
```

- Keep spinning until find lock is released

- But we can have the same issue as before

- We need to check and lock atomically

# Locks – Processes sharing CPU

- SpinLock

```
    Void
    acquire(struct spinlock *lk)
    {
        for(;;) {
P1              if(!lk->locked) {
                    lk->locked = 1;
                    break;
                }
        }
    }
```

- Keep spinning until find lock is released

- But we can have the same issue as before

- We need to check and lock atomically

# Locks – Processes sharing CPU

- SpinLock

```
    Void
    acquire(struct spinlock *lk)
    {
        for(;;) {
P1 CP1          if(!lk->locked) {
                    lk->locked = 1;
                    break;
                }
        }
    }
```

- Keep spinning until find lock is released
- But we can have the same issue as before
- We need to check and lock atomically

# Locks – Processes sharing CPU

- SpinLock

```
       Void
       acquire(struct spinlock *lk)
       {
           for(;;) {
               if(!lk->locked) {
P1 CP1                 lk->locked = 1;
                   break;
               }
           }
       }
```

- Keep spinning until find lock is released

- But we can have the same issue as before

- We need to check and lock atomically

# Locks – Processes sharing CPU

- XV6 relies on a special 386 hardware instruction, xchg
- Atomically check and change a register value
  - **`xchg(&lk->locked, 1)`**

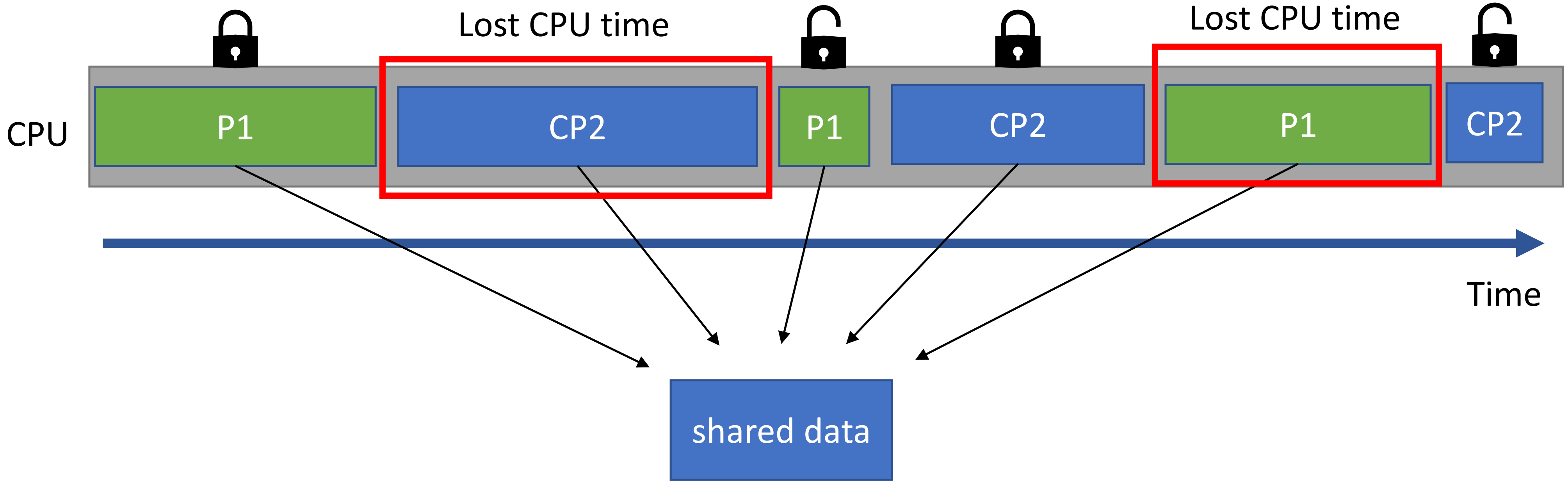
# Locks – Processes sharing CPU

- Swap a word in memory with the contents of a register
- In acquire function:
  - loop xchg instruction
  - Each round atomically read lock and set the lock to 1

```
void
acquire(struct spinlock *lk)
{
    pushcli(); // disable interrupts to
avoid deadlock.
        …

// The xchg is atomic.
    while(xchg(&lk->locked, 1) != 0);

        …

// Record info about lock acquisition for
debugging.
    lk->cpu = mycpu();
    getcallerpcs(&lk, lk->pcs);
}
```

# Locks – Processes sharing CPU

- But the we have another issue
  - Busy waiting



# Locks – Processes sharing CPU

- Spin Lock
  - Busy waiting
  - Useful for short critical sections
    - E.g. increment a counter, access an array element, etc.
  - Not useful, when the period of wait is unpredictable or will take a long time
    - E.g. read page from disk

# Locks – Processes sharing CPU

- Sleep Locks
  - For code need to hold a lock for a long time (read/write to disk)
- Avoids the schedule of “spin locked” processes

# Locks – Processes sharing CPU

- Sleep Locks
    - For code need to hold a lock for a long time (read/write to disk)
- Avoids the schedule of “spin locked” processes

```
void
acquiresleep(struct sleeplock *lk)
{
    acquire(&lk->lk);
    while (lk->locked) {
        sleep(lk, &lk->lk);
    }
    lk->locked = 1;
    lk->pid = myproc()->pid;
    release(&lk->lk);
}
```

```
void
releasesleep(struct sleeplock *lk)
{
    acquire(&lk->lk);
    lk->locked = 0;
    lk->pid = 0;
    wakeup(lk);
    release(&lk->lk);
}
```

# Locks – Processes sharing CPU

- Sleep Locks
  - For code need to hold a lock for a long time (read/write to disk)
- Avoids the schedule of “spin locked” processes

```
void
acquiresleep(struct sleeplock *lk)
{
    acquire(&lk->lk);
    while (lk->locked) {
        sleep(lk, &lk->lk);
    }
    lk->locked = 1;
    lk->pid = myproc()->pid;
    release(&lk->lk);
}
```

```
void
releasesleep(struct sleeplock *lk)
{
    acquire(&lk->lk);
    lk->locked = 0;
    lk->pid = 0;
    wakeup(lk);
    release(&lk->lk);
}
```

# Locks – Processes sharing CPU

- Put one process to sleep waiting for event
- Mark current process as sleeping
- Call **sched()** to release the processor

```
void
sleep(void *chan, struct spinlock *lk)
{
  struct proc *p = myproc();
    …
  p->state = SLEEPING;

  sched();
    …
}
```

- Put one process to sleep waiting for event

- Mark current process as sleeping

- Call **sched()** to release the processor

```
void
sleep(void *chan, struct spinlock *lk)
{
  struct proc *p = myproc();

  if(p == 0)
    panic("sleep");

  if(lk == 0)
    panic("sleep without lk");
  if(lk != &ptable.lock){
    acquire(&ptable.lock);
    release(lk);
  }
  p->chan = chan;
  p->state = SLEEPING;

  sched();
  p->chan = 0
  if(lk != &ptable.lock){
    release(&ptable.lock);
    acquire(lk);
  }
}
```

*Sanity Checks*

- *Must be a current process*
- *Must have been passed a lock*

- Put one process to sleep waiting for event

- Mark current process as sleeping

- Call **sched()** to release the processor

```
void
sleep(void *chan, struct spinlock *lk)
{
  struct proc *p = myproc();

  if(p == 0)
    panic("sleep");

  if(lk == 0)
    panic("sleep without lk");

  if(lk != &ptable.lock){
    acquire(&ptable.lock);
    release(lk);
  }
  p->chan = chan;
  p->state = SLEEPING;

  sched();
  p->chan = 0
  if(lk != &ptable.lock){
    release(&ptable.lock);
    acquire(lk);
  }
}
```

*Hold the ptable.lock, it is safe to release lk*

# Locks – Processes sharing CPU

- Wake up process when event happened
- Mark a waiting process as runnable

```
static void
wakeup(void *chan)
{
    struct proc *p;
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
        if(p->state == SLEEPING && p->chan == chan)
            p->state = RUNNABLE;
}
```

# Locks – Processes sharing CPU

- Who needs to be a syscall?
  - SpinLocks
  - SleepLocks

# CS 1550 – Lab exercise 2

- **PROCESS SYNCHRONIZATION IN XV6**
  - **Due**: Monday, February 17, 2020 @11:59pm

  - Part 2 - step 5: user.h
    - Add declaration for init_lock()
      - void init_lock(struct spinlock *);
    - struct condvar;
    - struct spinlock;

  - Part 3 - step 8: defs.h
    - Add declaration for sleep1()

# CS 1550

## Week 5 – Synchronization with xv6

Teaching Assistant

Henrique Potter