# Parallelism

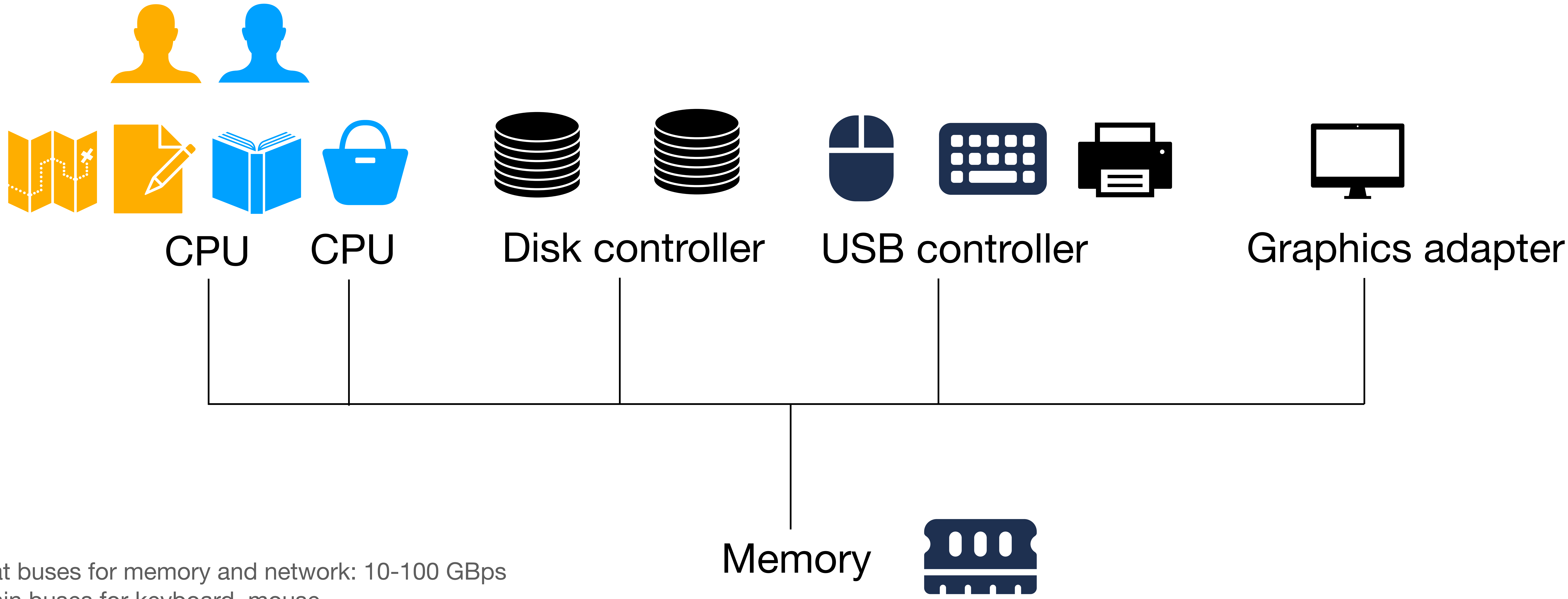
**Abhilash Jindal**

# Agenda

## xv6 book Ch4, OSTEP Ch 25-32

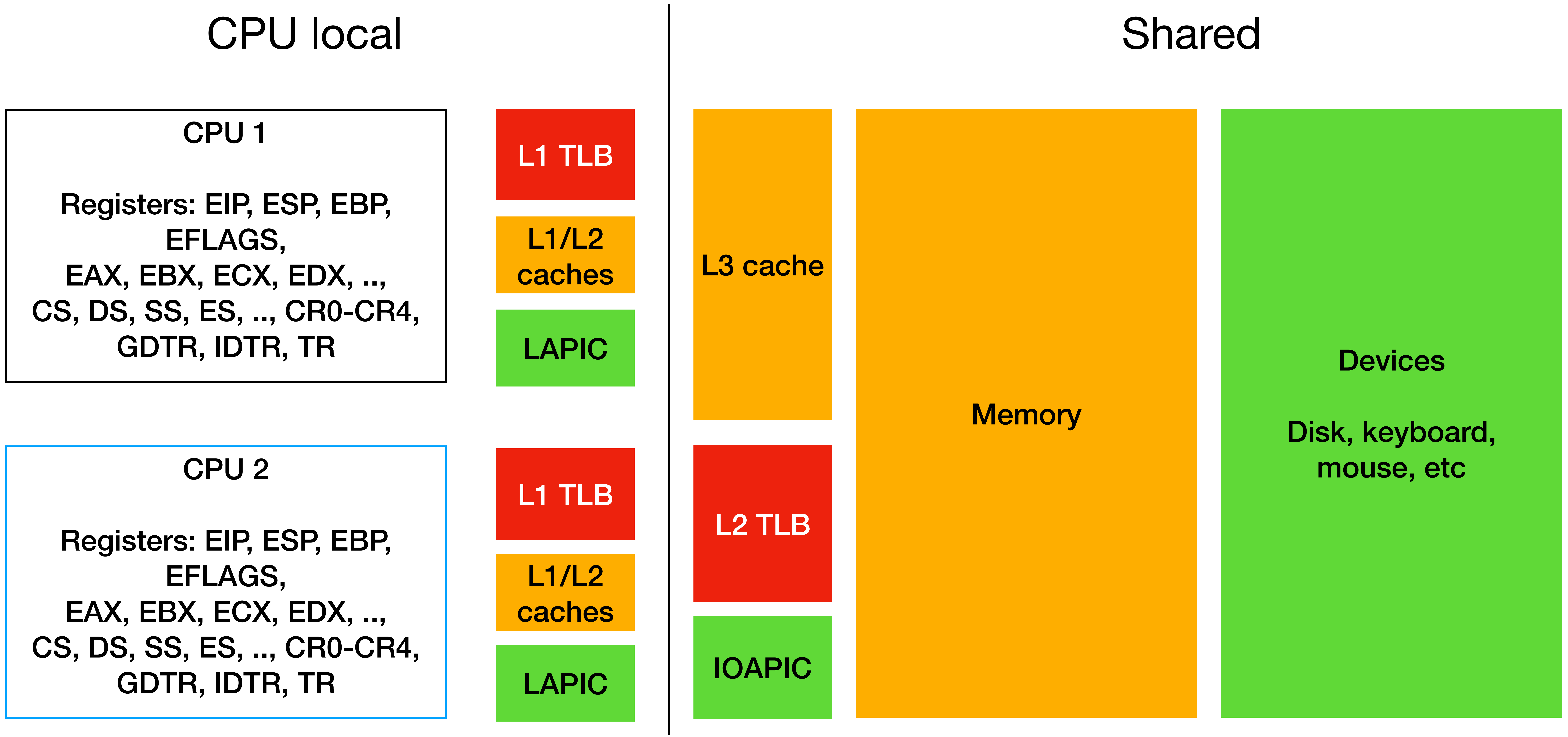
- Multi-processing hardware
    - xv6 setting up other processors
- Threads
    - Race conditions
- Design of locks
    - Spin locks, conditional waits, semaphores, read-write locks
    - Atomic instructions and memory consistency models
- Difficulties with using locks
    - Deadlocks and modularity

# Computer organization




Fat buses for memory and network: 10-100 GBps
Thin buses for keyboard, mouse

# Computer organization



# main calls startothers

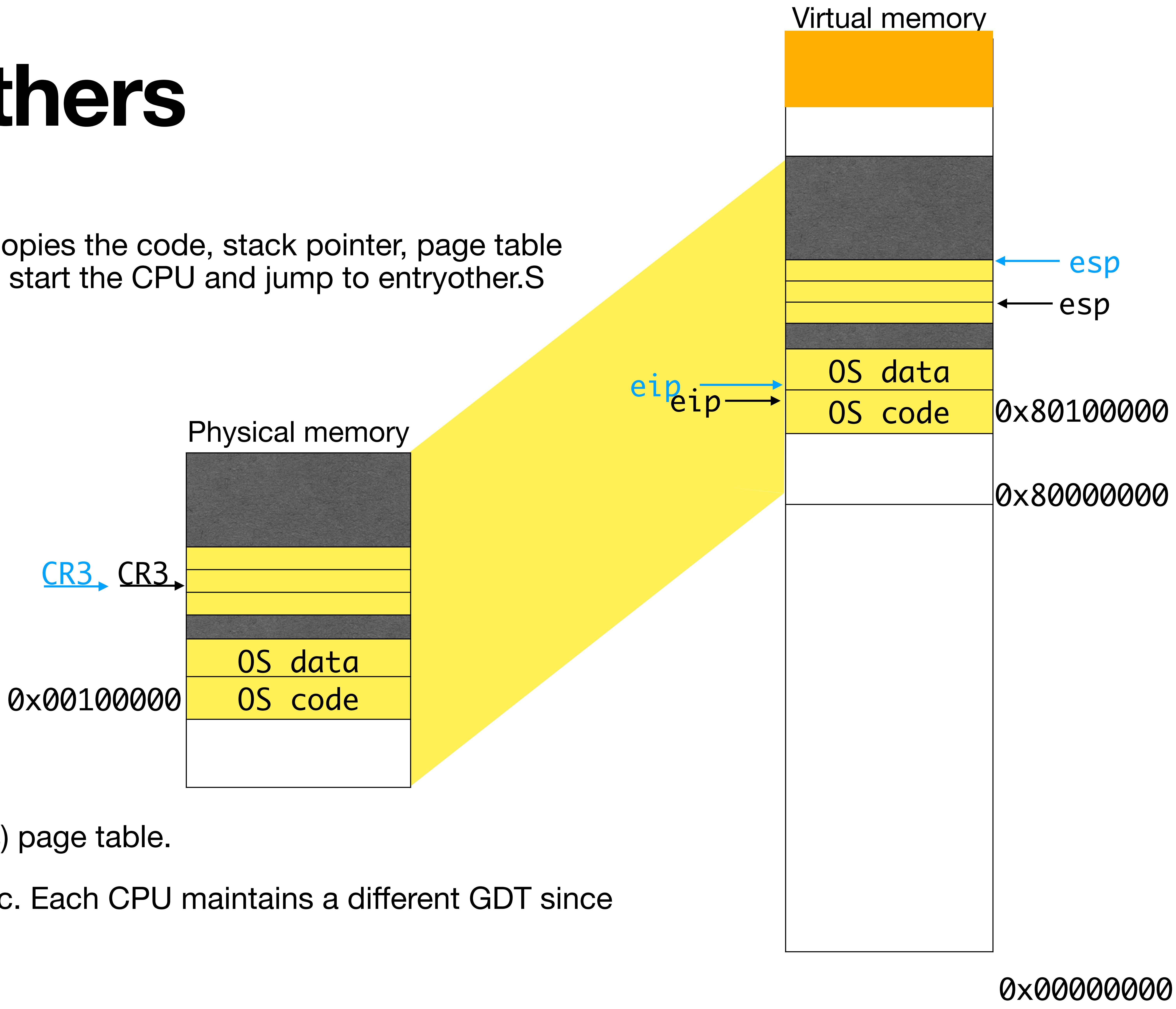
startothers:

- Allocates a separate stack for the other CPU, copies the code, stack pointer, page table pointer in low 1MB, asks other CPU's LAPIC to start the CPU and jump to entryother.S

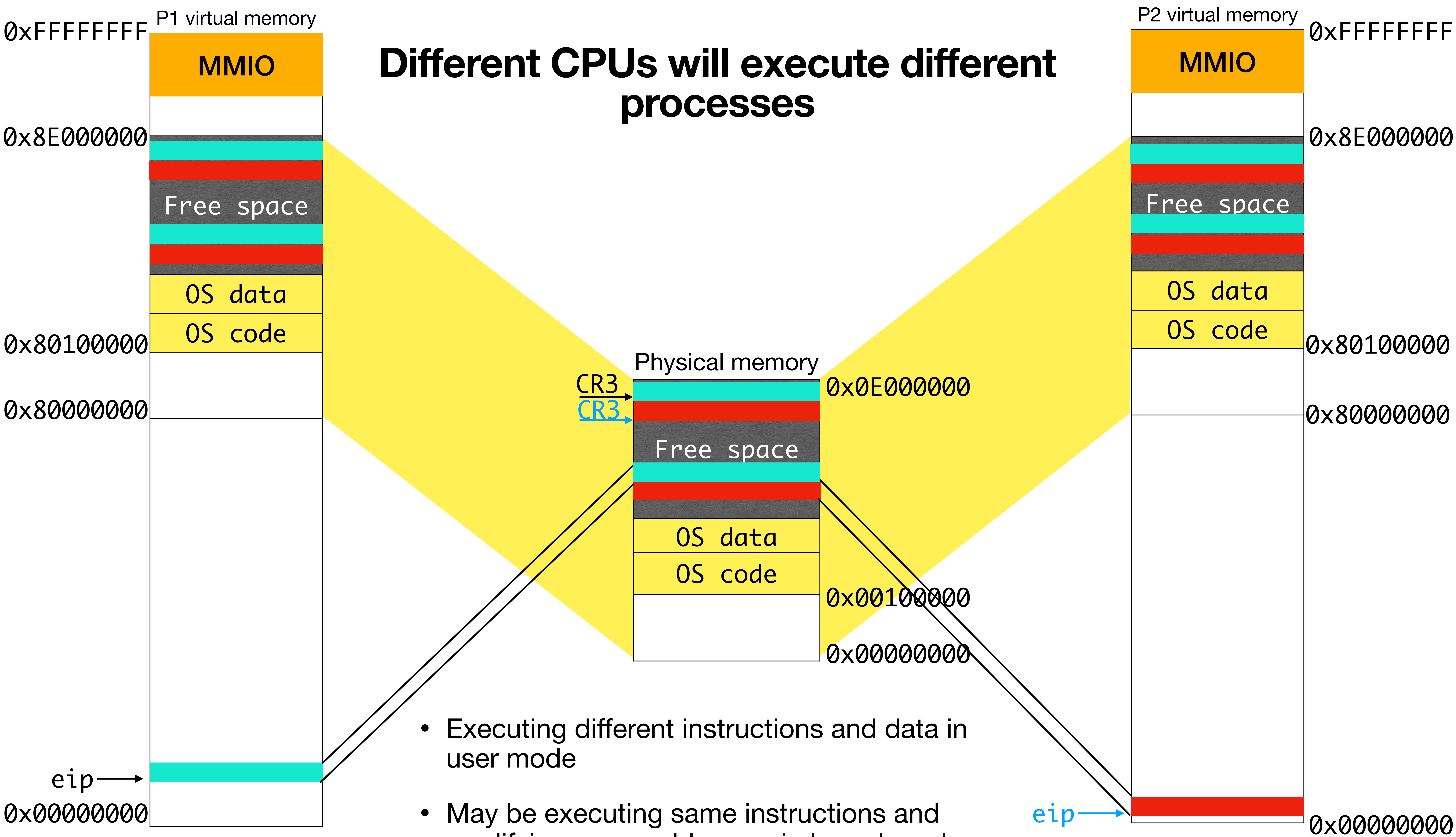
entryother.S does what bootloader+entry.S did

  - clears interrupts
  - sets up temporary GDT, GDTR
  - Switch to 32 bit mode, set segment selectors
  - enable paging with temporary page table, sets up stack pointer
  - jumps to mpenter

- mpenter does what main did
  - Switch to (OS only in high virtual addresses) page table.
  - Sets up new GDT with KCODE, UCODE, etc. Each CPU maintains a different GDT since each CPU will write TSS in its own GDT
  - Sets up IDTR and jump into scheduler

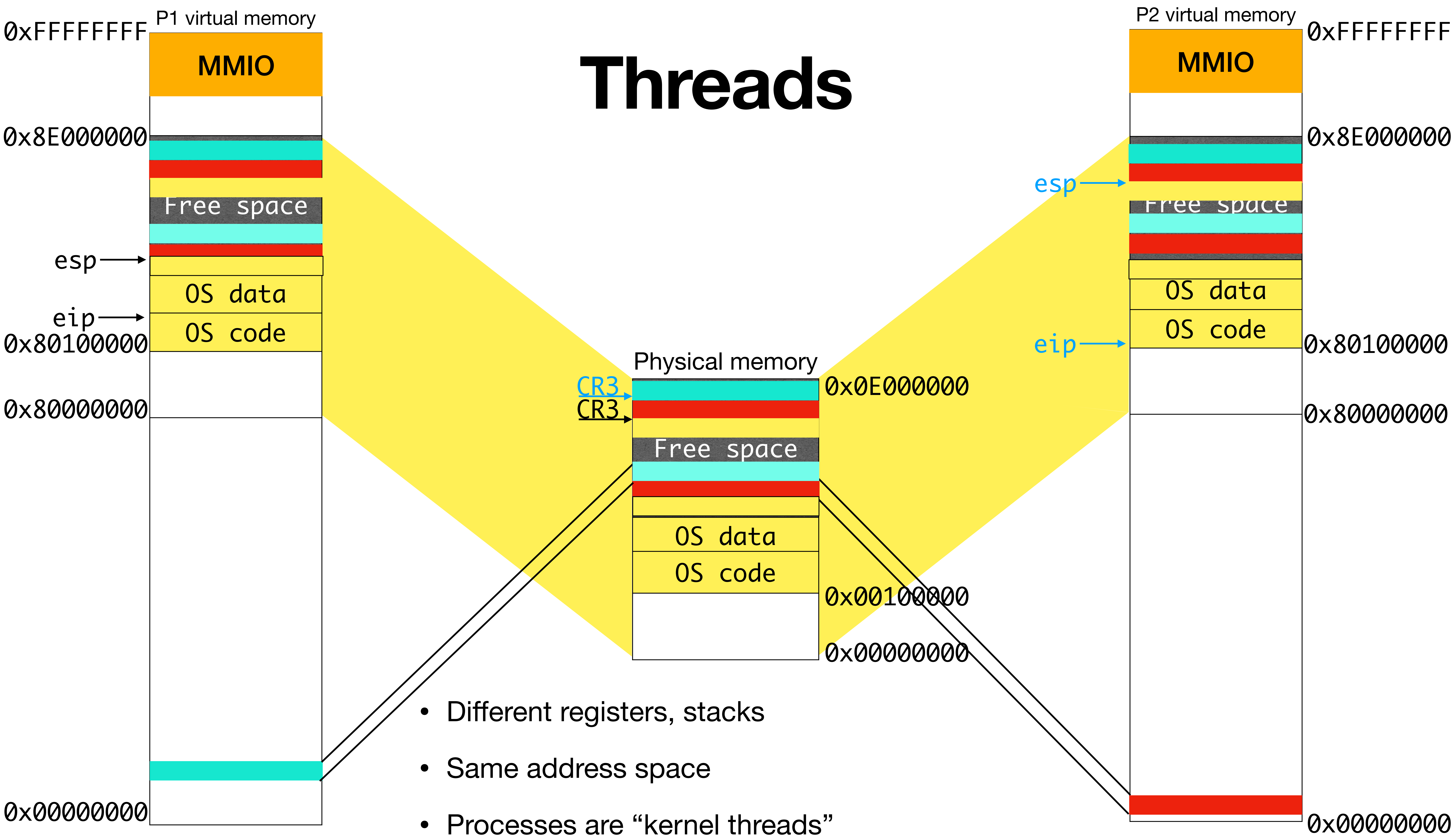


# Different CPUs will execute different processes




- Executing different instructions and data in user mode
- May be executing same instructions and modifying same addresses in kernel mode

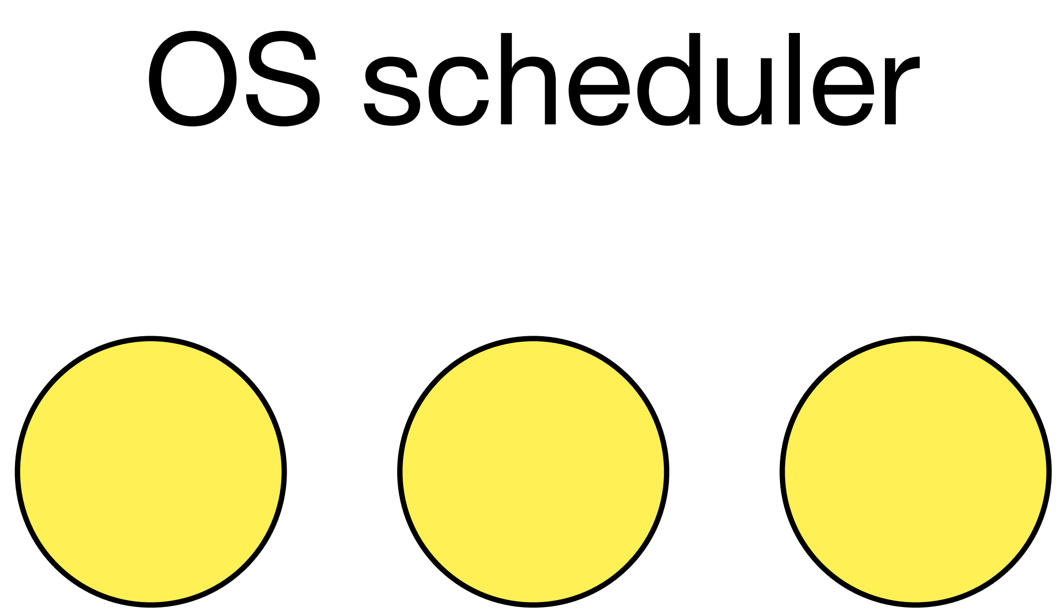
# Threads




- Different registers, stacks
- Same address space
- Processes are “kernel threads”

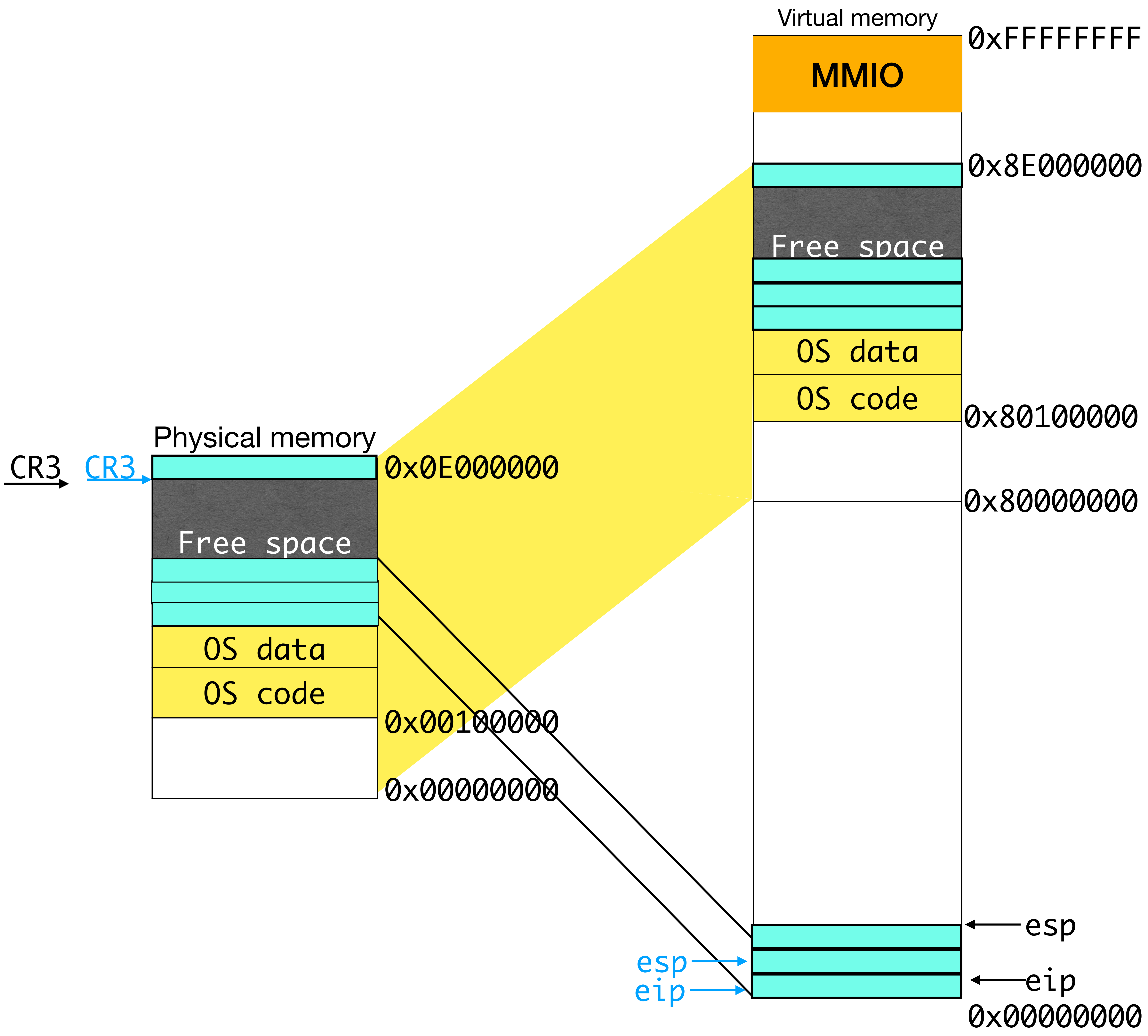
# Processes are kernel threads

- When executing in OS mode:
    - Different kernel stacks, different registers
    - Same address space

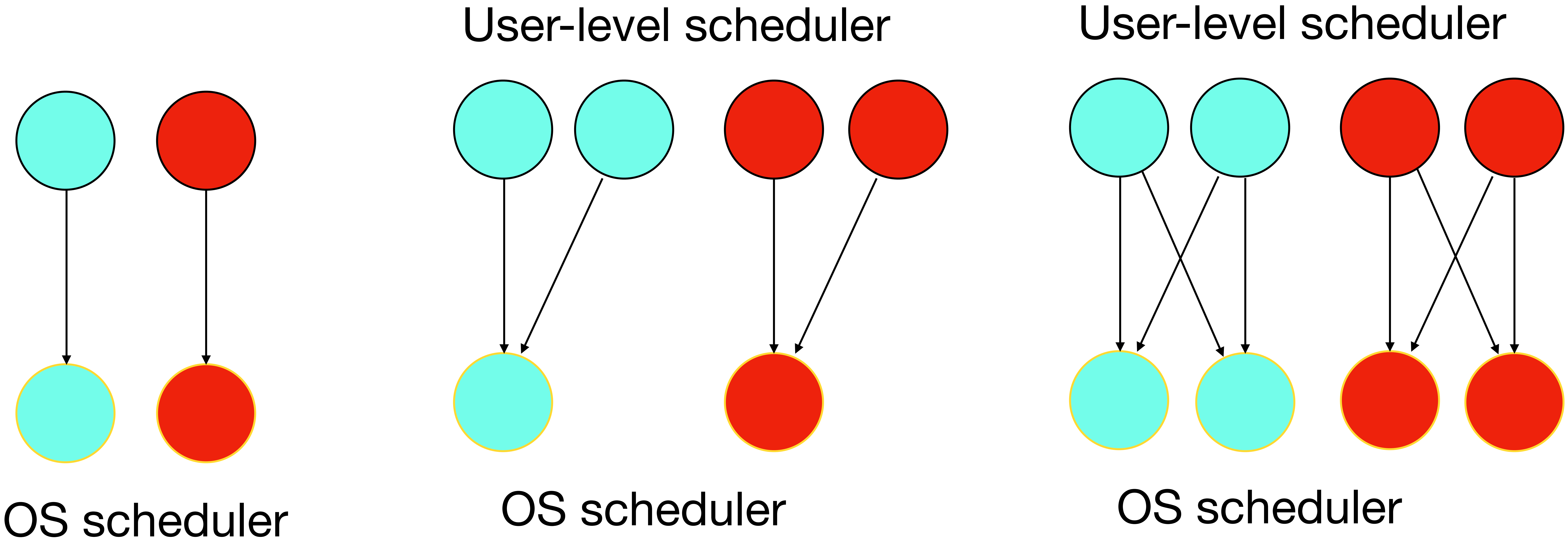


# User threads

- Different registers, stacks
- Same address space



# User threads and kernel threads




xv6

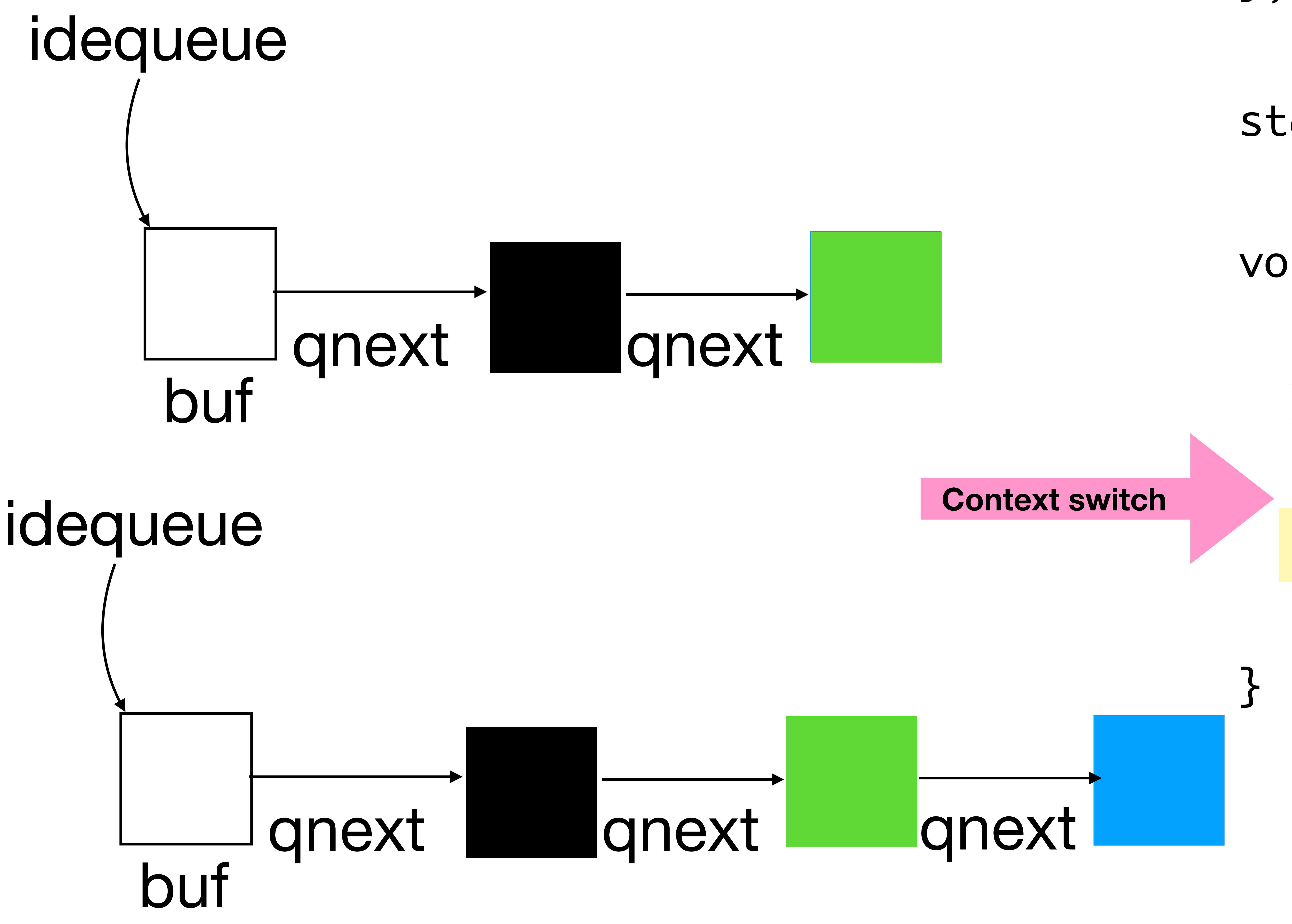
COP290 mythreads

General threading
e.g, pthreads

Cannot exploit parallelism within a process: two CPUs will never run with same address space in user code

# Race conditions

**Example: kernel threads reading a block in ide.c**




```
struct buf {
  ..
  struct buf *qnext; // disk queue
  uchar data[BSIZE];
};

static struct buf *idequeue;

void iderw(struct buf *b) {
  struct buf **pp;
  b->qnext = 0;
  for(pp=&idequeue; *pp; pp=&(*pp)->qnext);
  *pp = b;
  ..
}
```

# Race conditions and critical sections

- Similar races can happen in user threads. Example: 01/threads.c

| **Thread 1** | **Thread 2** |
|---|---|
| Read counter = 0 | |
| Write counter = 1 | |
| | Read counter = 1 |
| | Writer counter = 2 |
| Read counter = 2 | |
| | Read counter = 2 |
| | Writer counter = 3 |
| Writer counter = 3 | |

Read counter, writer counter needs to happen atomically

Critical section: “counter++”
threads-safe.c

# Lock implementation

```
void iderw(struct buf *b) {
  struct buf **pp;
  acquire();
  for(pp=&idequeue; *pp; pp=&(*pp)->qnext);
  *pp = b;
  ..
  release();
}
```

- Timer interrupt and hence context switch cannot happen between acquire and release

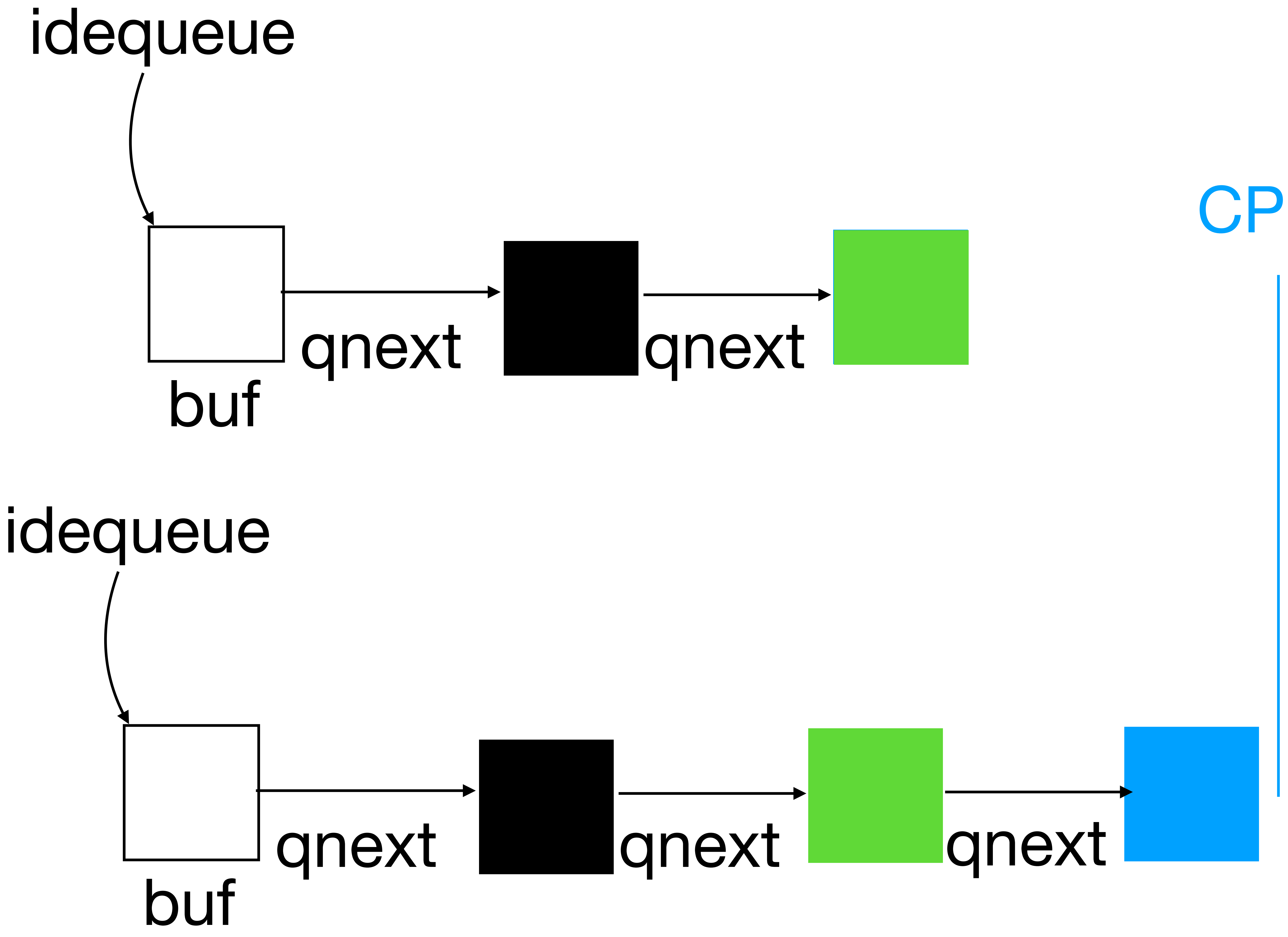
```
void acquire() {
  pushcli();
}
void pushcli(void) {
  int eflags = readeflags();
  cli();
  if(cpu->ncli == 0)
    cpu->intena = eflags & FL_IF;
  cpu->ncli += 1;
}

void release() {
  popcli();
}
void popcli(void) {
  cpu->ncli--;
  if(cpu->ncli == 0 && cpu->intena)
    sti();
}
```

# Problems with disabling interrupts

- For user-level code:
  - After acquiring lock, threads goes into infinite loop
  - OS lost control of the CPU
- Does not work on multiple processor

# Race conditions

**Example: kernel threads reading a block in ide.c**




```
struct buf {
  ..
  struct buf *qnext; // disk queue
  uchar data[BSIZE];
};

static struct buf *idequeue;
```

CPU 1

CPU 2

```
void iderw(struct buf *b) {
  struct buf **pp;
  b->qnext = 0;
  cli();
  for(pp=&idequeue; *pp; pp=&(*pp)->qnext);
  *pp = b;
  sti();
  ..
}
```

# Spin locks

Call to lock spins while waiting for the other thread to unlock

CPU 1

```
typedef struct __lock_t { int flag; } lock_t;

void init(lock_t *mutex) {
    // 0 -> lock is available, 1 -> held
    mutex->flag = 0;
}

void lock(lock_t *mutex) {
    while (mutex->flag == 1)   // TEST the flag
        ; // spin-wait (do nothing)
    mutex->flag = 1;           // now SET it!
}

void unlock(lock_t *mutex) {
    mutex->flag = 0;
}
```

CPU 2

# Write to two different flags to avoid races?

```
int flag[2];

void init() {
  flag[0] = flag[1] = 0; // indicates that you want to hold the lock
}
```

CPU 1

CPU 2

```
void lock() {
  flag[self] = 1; // self: thread ID of caller
  while (flag[1 - self] == 1) ; // spin-wait
}

void unlock() {
  flag[self] = 0; // simply undo your intent
}
```

Deadlock

| | |
|---|---|
| flag[0] = 1 | flag[1] = 1 |
| while(flag[1] == 1); | while(flag[0] == 1); |