

# Parallelism

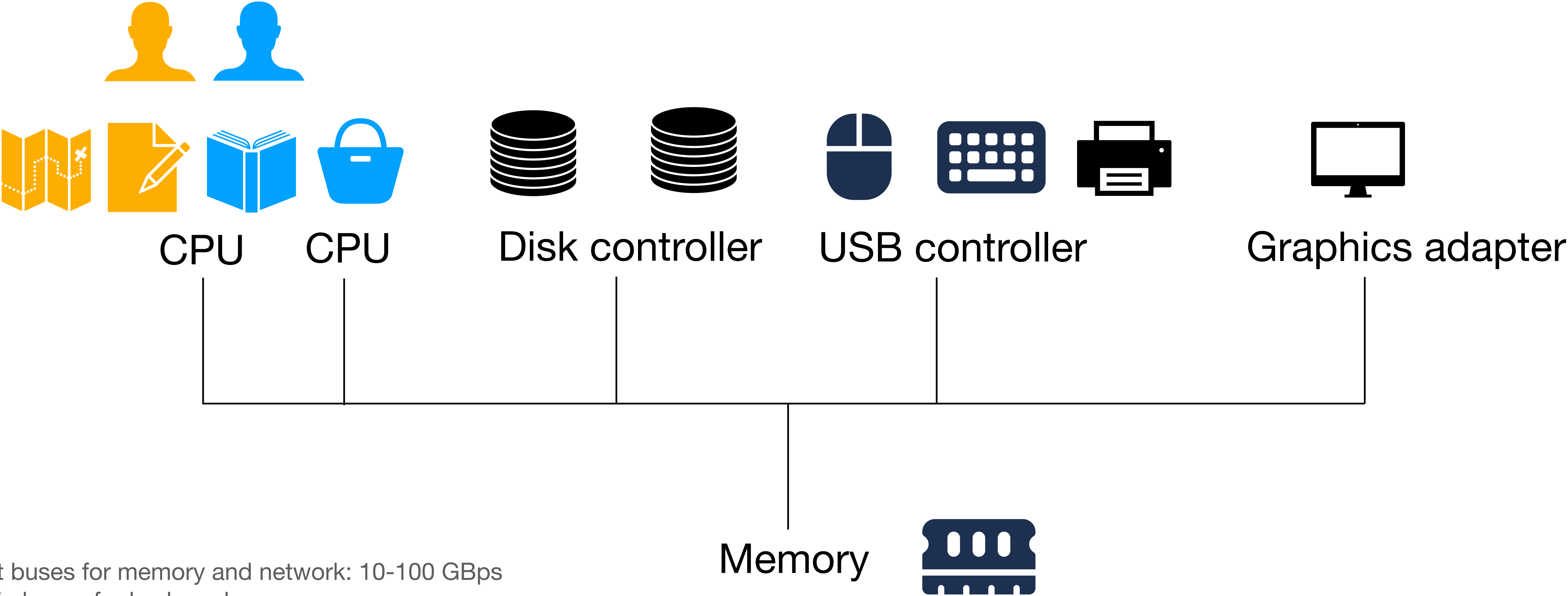
Abhilash Jindal

# Agenda

## xv6 book Ch4, OSTEP Ch 25-32

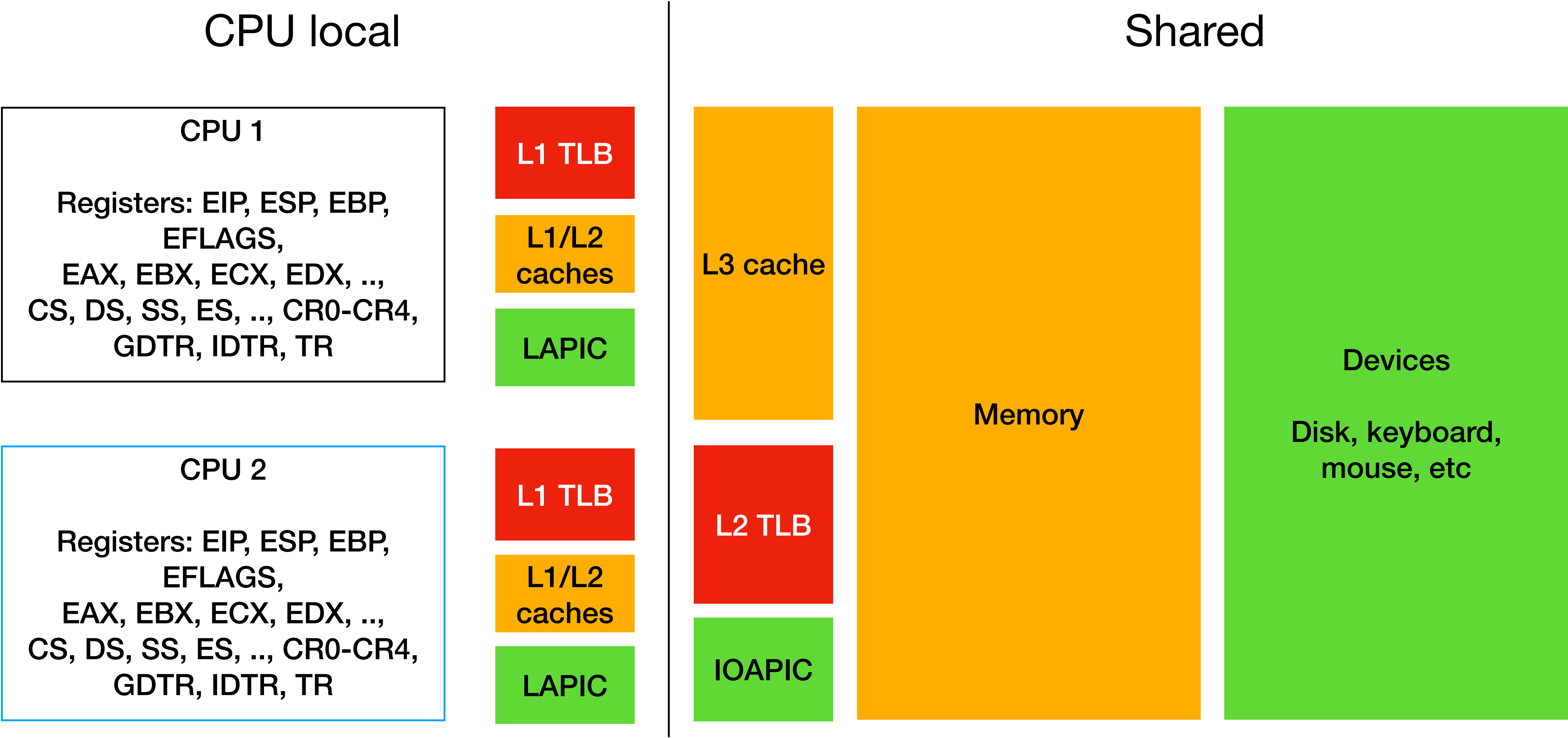
- Multi-processing hardware
  - xv6 setting up other processors
- Threads
  - Race conditions
- Design of locks
  - Spin locks, conditional waits, semaphores, read-write locks
  - Atomic instructions and memory consistency models
- Difficulties with using locks
  - Deadlocks and modularity

# Computer organization



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

# Computer organization



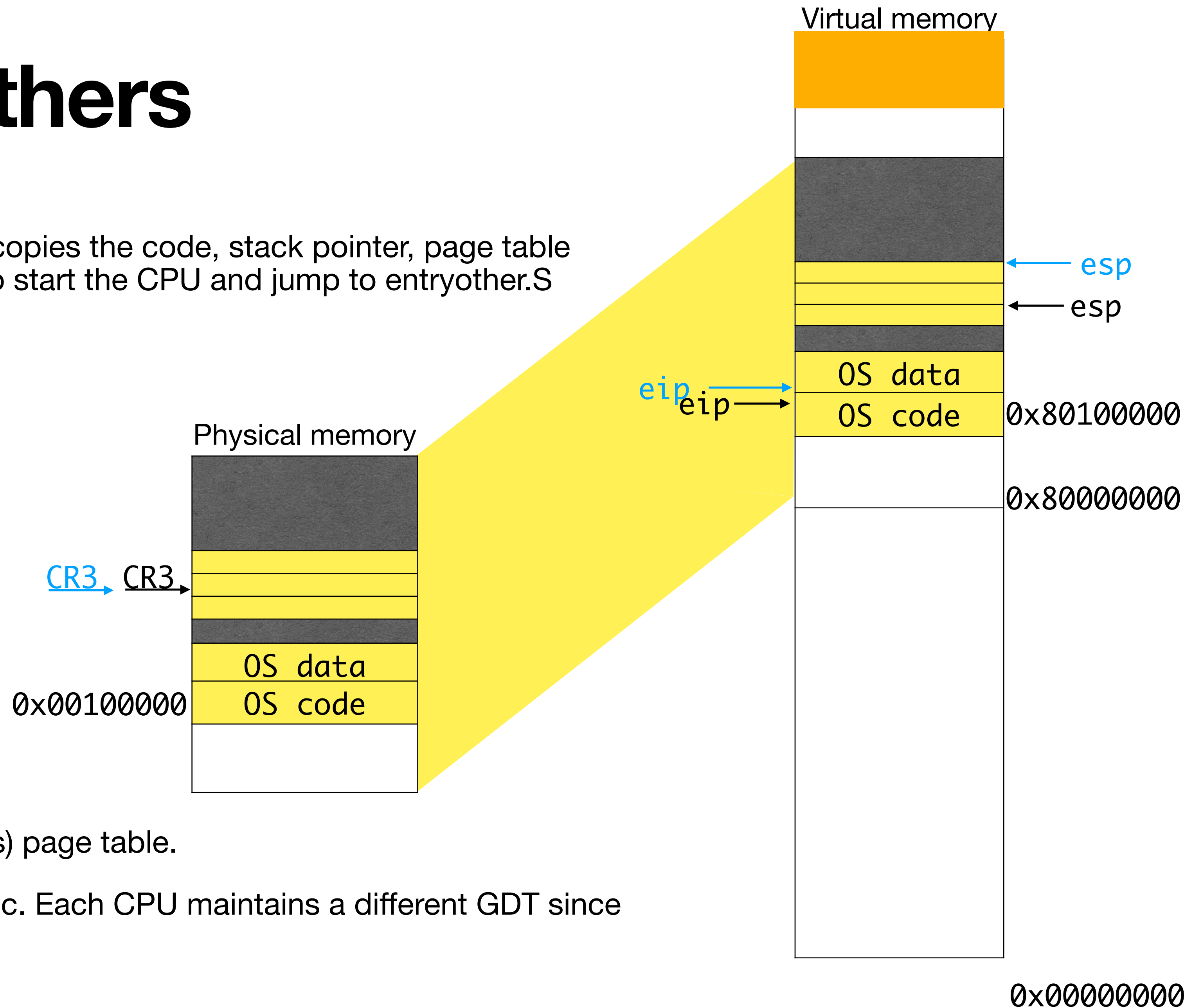
# main calls startothers

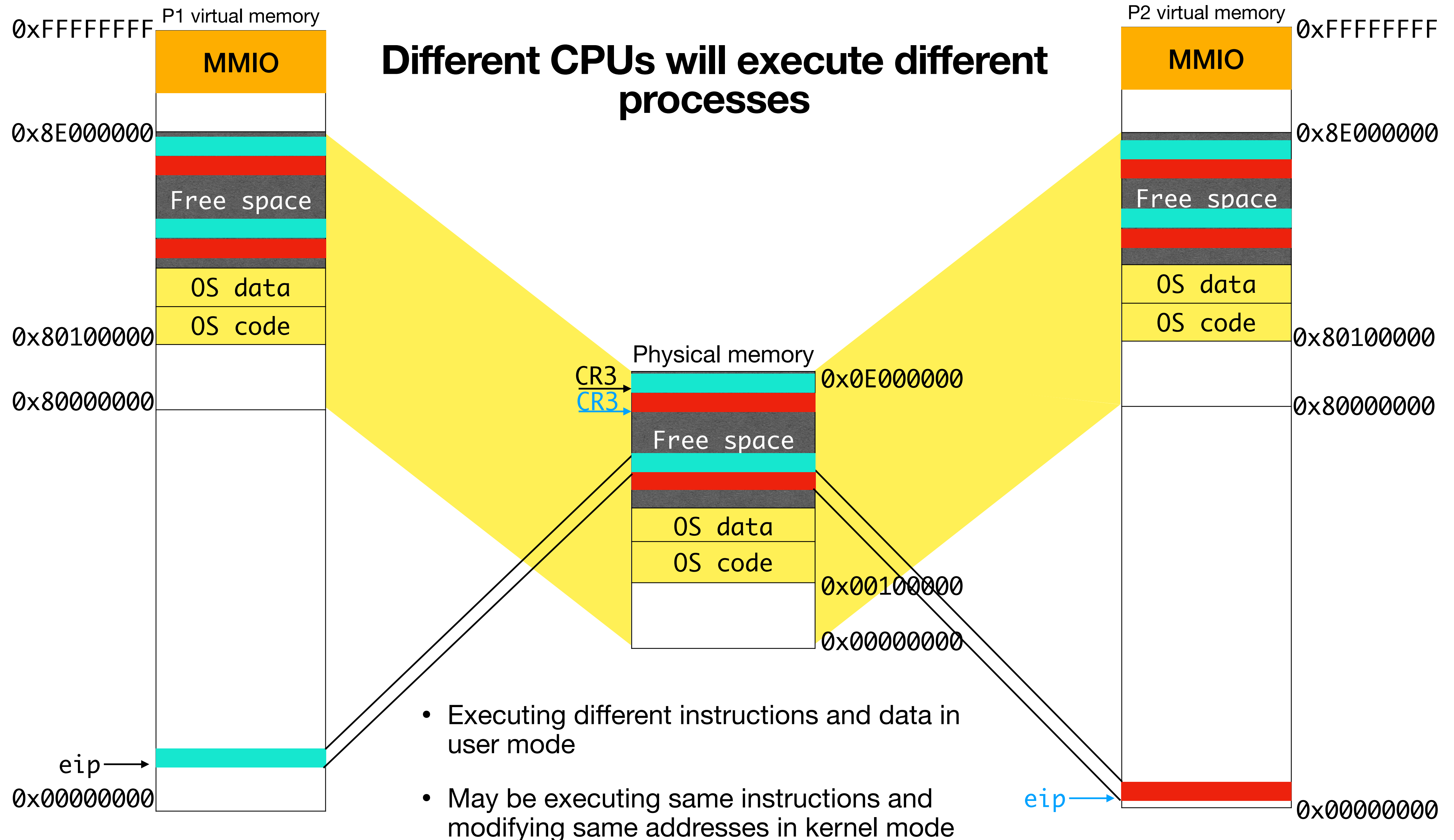
startothers:

- 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

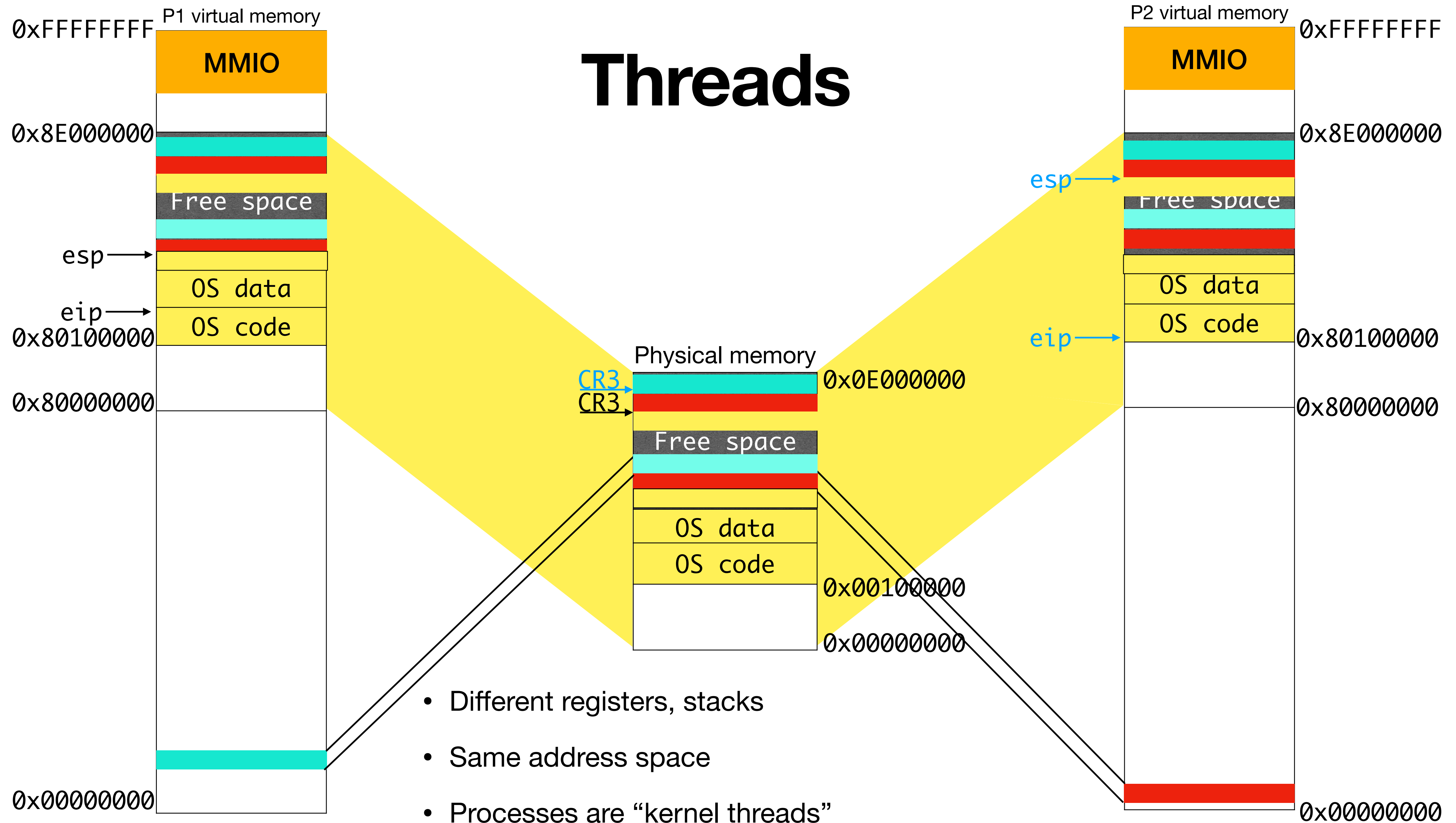
entryother.S does what bootloader+entry.S did

- 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





# Threads

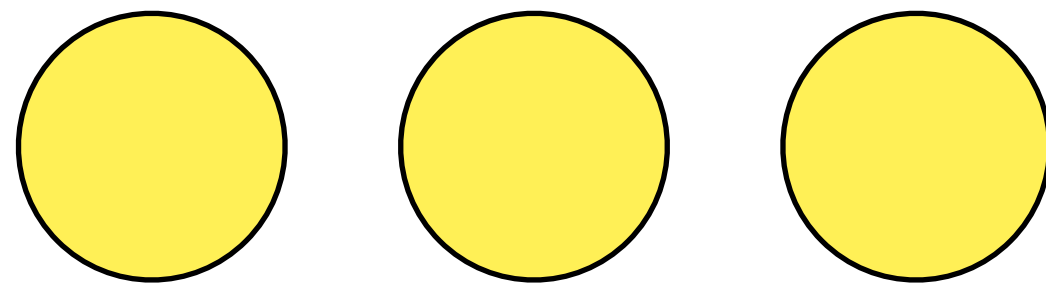




# Processes are kernel threads

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

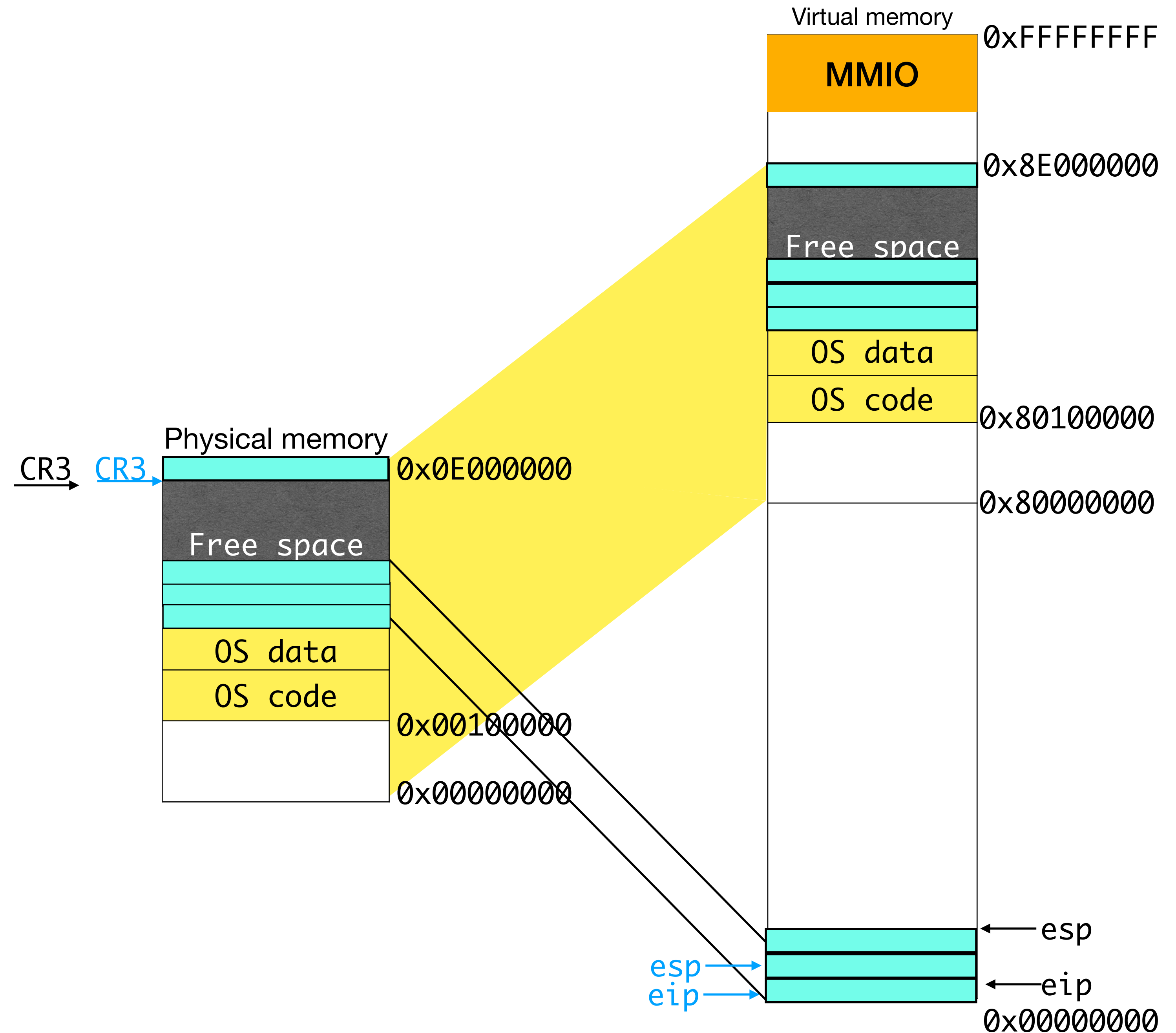
OS scheduler



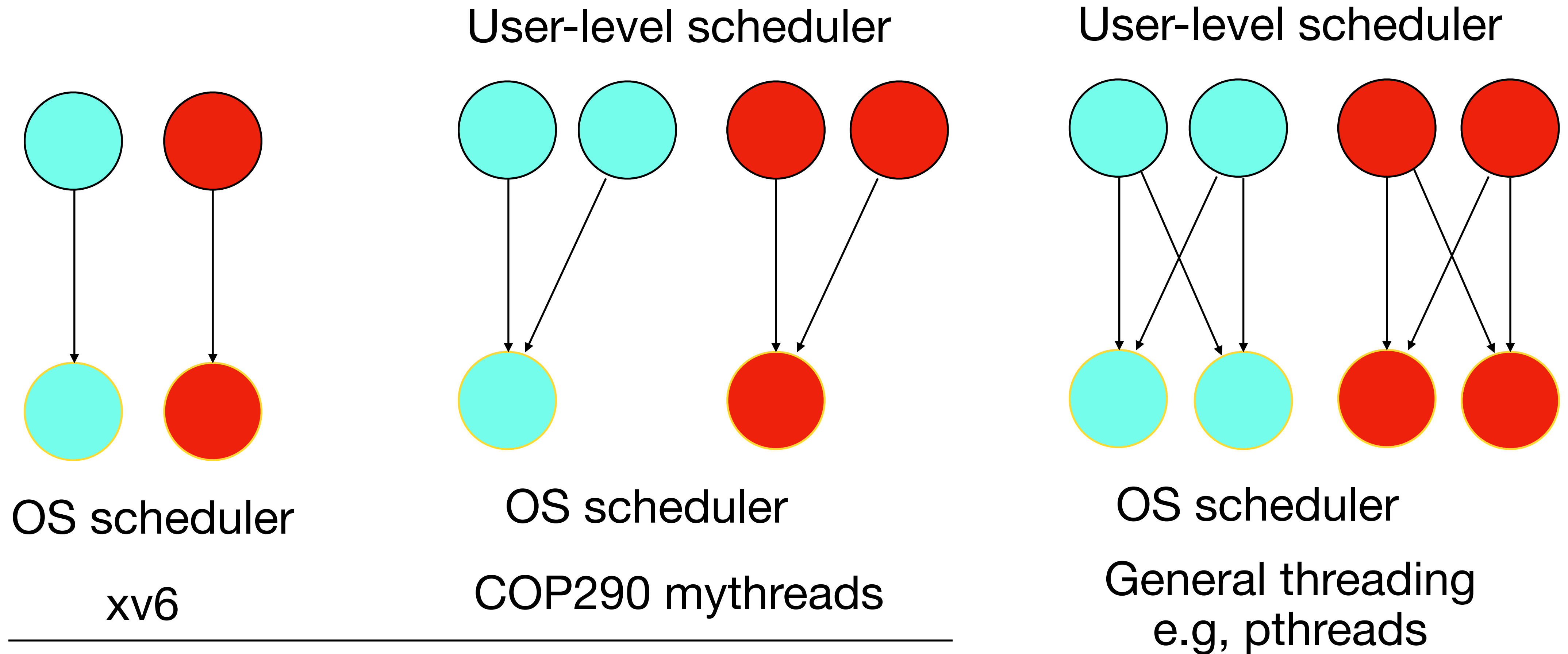


# User threads

- Different registers, stacks
- Same address space



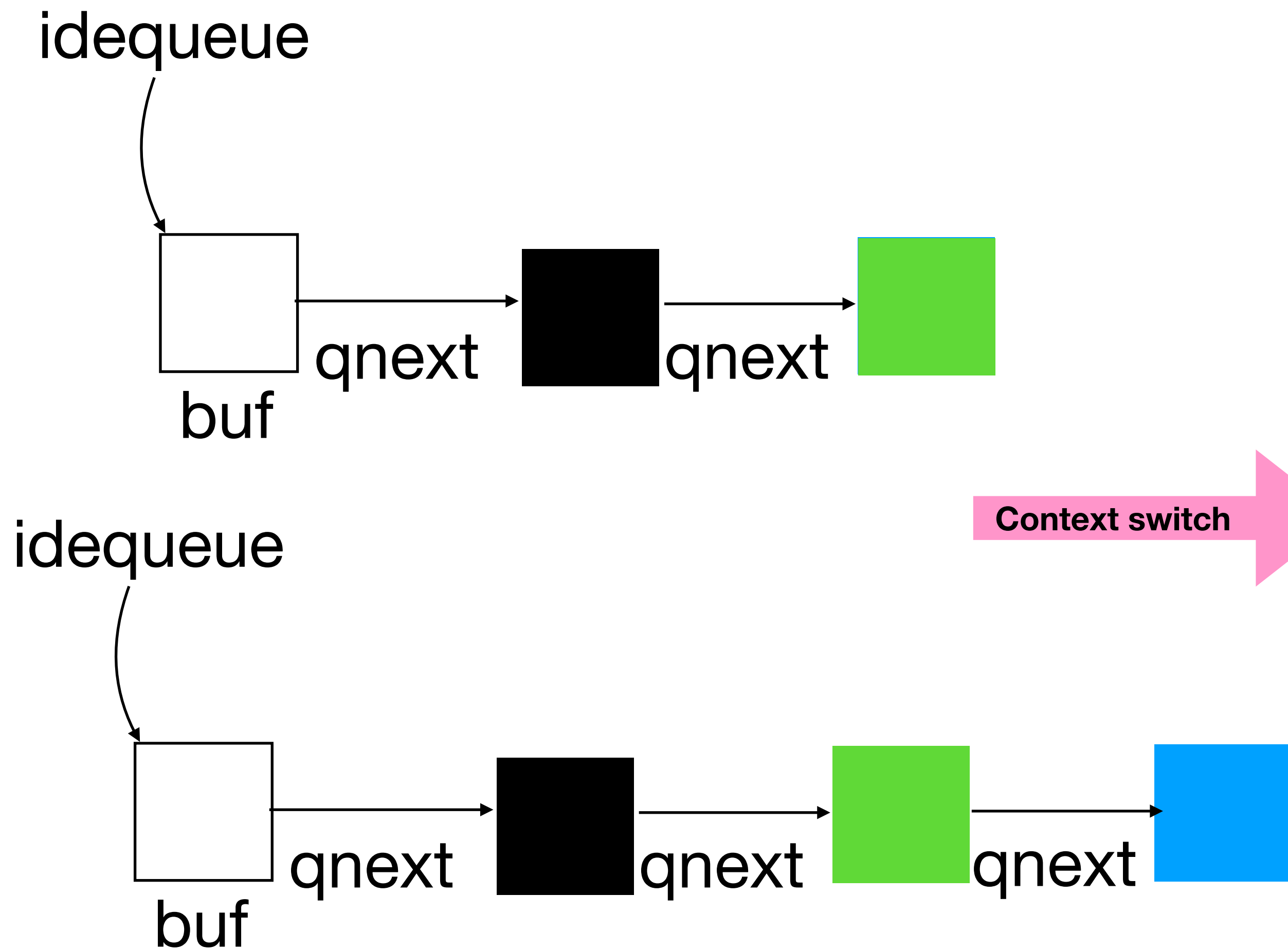
# User threads and kernel threads



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

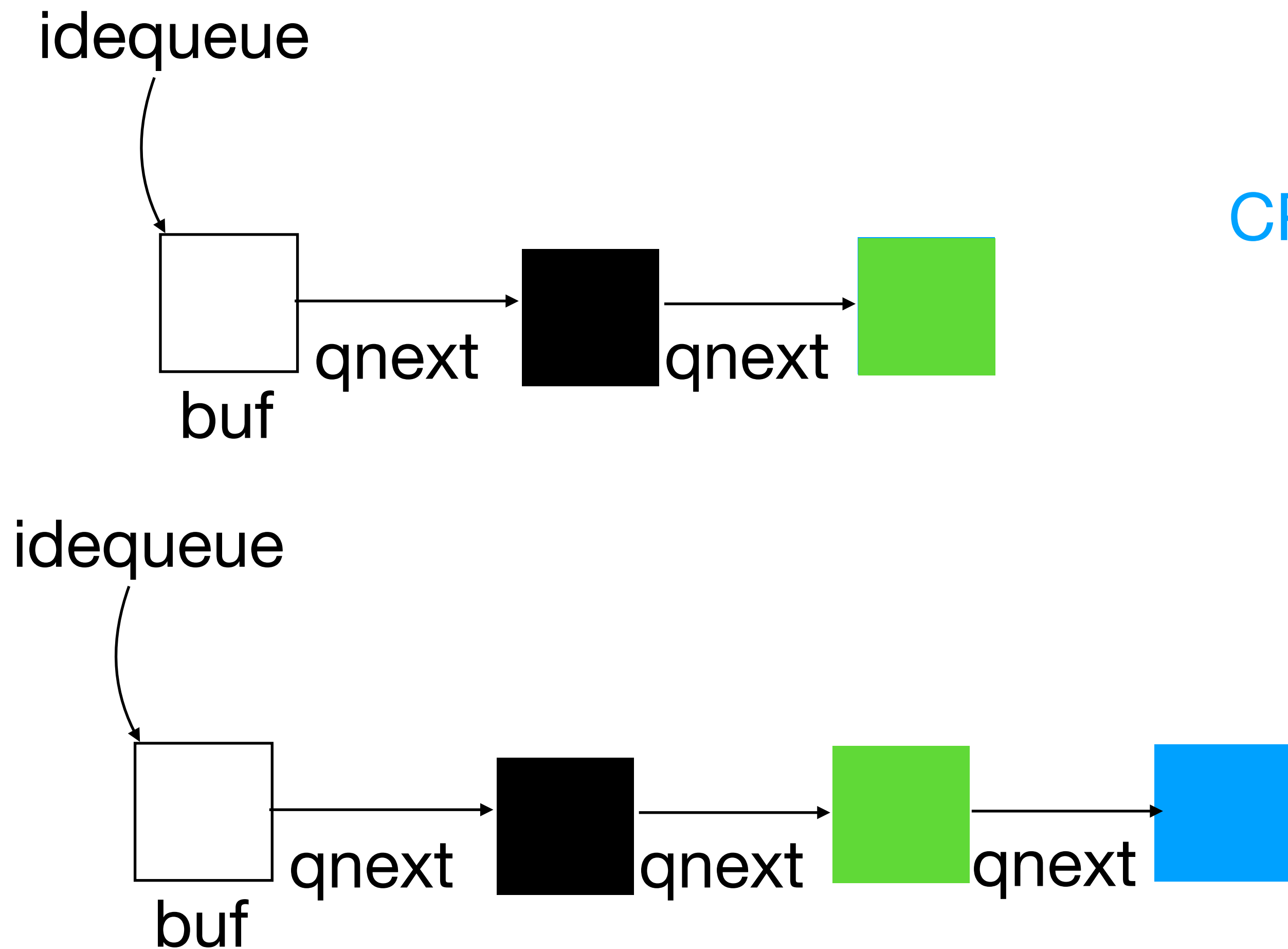
```
void acquire() {  
    pushcli();  
}  
void pushcli(void) {  
    int eflags = readeflags();  
    cli();  
    if(cpu->ncli == 0)  
        cpu->intena = eflags & FL_IF;  
    cpu->ncli += 1;  
}  
void release() {  
    popcli();  
}  
void popcli(void) {  
    cpu->ncli--;  
    if(cpu->ncli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

```
void iderw(struct buf *b) {
```

```
struct buf **pp;
```

```
b->qnext = 0;
```

```
cli();
```

```
for(pp=&idequeue; *pp; pp=&(*pp)->qnext);
```

```
*pp = b;
```

```
sti();
```

```
..
```

```
}
```

CPU 2



# Spin locks

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

CPU 1



CPU 2



```
1  typedef struct __lock_t { int flag; } lock_t;
2
3  void init(lock_t *mutex) {
4      // 0 -> lock is available, 1 -> held
5      mutex->flag = 0;
6  }
7
8  void lock(lock_t *mutex) {
9      while (mutex->flag == 1) // TEST the flag
10         ; // spin-wait (do nothing)
11     mutex->flag = 1; // now SET it!
12 }
13
14 void unlock(lock_t *mutex) {
15     mutex->flag = 0;
16 }
```

# 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[0] == 1);
while(flag[1] == 1);	

# Safety and liveness

- Safety: Bad things never happen
  - Two threads shall never *simultaneously* acquire the lock
- Liveness: Good things eventually happen
  - Some thread (trying to lock) eventually gets to acquire the lock

# Trivial useless locks

Safe but not live

```
void lock() {  
    while(1);  
}
```

Live but not safe

```
void lock() {  
    return;  
}
```

# Peterson's algorithm\*

turn breaks the tie

<pre>while(flag[1] == 1       &amp;&amp; turn == 1);</pre>	<pre>while(flag[0] == 1       &amp;&amp; turn == 0);</pre>
--	--

```
int flag[2];
int turn = 0; // whose turn? (thread 0 or 1?)
```

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

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

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

# Peterson's algorithm\*

```
void lock() {
  flag[0] = 1; // A
  turn = 0;    // B
  while ((flag[1] == 1) && (turn == 1)); //C
}
```

```
void lock() {
  flag[1] = 1; // D
  turn = 1;    // E
  while ((flag[0] == 1) && (turn == 0)); // F
}
```

B: turn = 0	B: turn = 0	B: turn = 0	E: turn = 1	E: turn = 1	E: turn = 1
E: turn = 1	E: turn = 1	C: turn != 1	B: turn = 0	B: turn = 0	F: turn != 0
C: turn != 1	F: turn != 0	E: turn = 1	F: turn != 0	C: turn != 1	B: turn = 0
F: turn != 0	C: turn != 1	F: turn != 0	C: turn != 1	F: turn != 0	C: turn != 1

# Peterson's algorithm

```
int flag[2];
int turn = 0; // whose turn? (thread 0 or 1?)

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

void lock() {
    flag[self] = 1; // self: thread ID of caller

    turn = 1 - self; // make it other thread's turn
    while ((flag[1 - self] == 1) && (turn == 1 - self)) ; // spin-wait
}

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



# Peterson's algorithm

```
void lock() {
    flag[0] = 1; // A
    turn = 1;    // B
    while ((flag[1] == 1) && (turn == 1)); //C
}
```

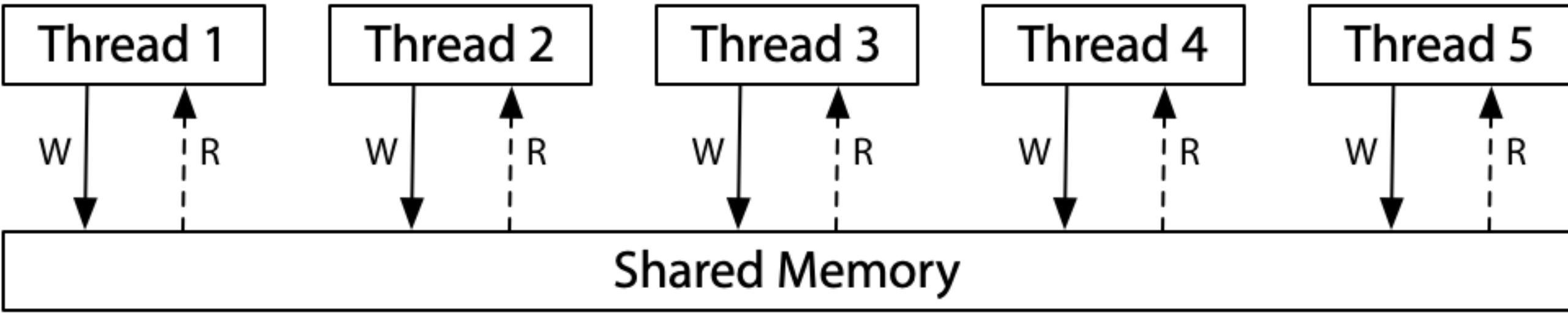
```
void lock() {
    flag[1] = 1; // D
    turn = 0;    // E
    while ((flag[0] == 1) && (turn == 0)); // F
}
```

B: turn = 1	B: turn = 1	B: turn = 1	E: turn = 0	E: turn = 0	E: turn = 0
E: turn = 0	E: turn = 0	C: turn != 1	B: turn = 1	B: turn = 1	F: turn != 0
C: turn != 1	F: turn != 0	E: turn = 0	F: turn != 0	C: turn != 1	B: turn = 1
F: turn != 0	C: turn != 1	F: turn != 0	C: turn != 1	F: turn != 0	C: turn != 1
		C: turn != 1			F: turn != 0

# Peterson algorithm breaks on x86 [peterson-breaks.c](http://peterson-breaks.c)

B: turn = 1	B: turn = 1	B: turn = 1	E: turn = 0	E: turn = 0	E: turn = 0
E: turn = 0	E: turn = 0	C: turn != 1	B: turn = 1	B: turn = 1	F: turn != 0
C: turn != 1	F: turn != 0	E: turn = 0	F: turn != 0	C: turn != 1	B: turn = 1
F: turn != 0	C: turn != 1	F: turn != 0	C: turn != 1	F: turn != 0	C: turn != 1
		C: turn != 1			F: turn != 0

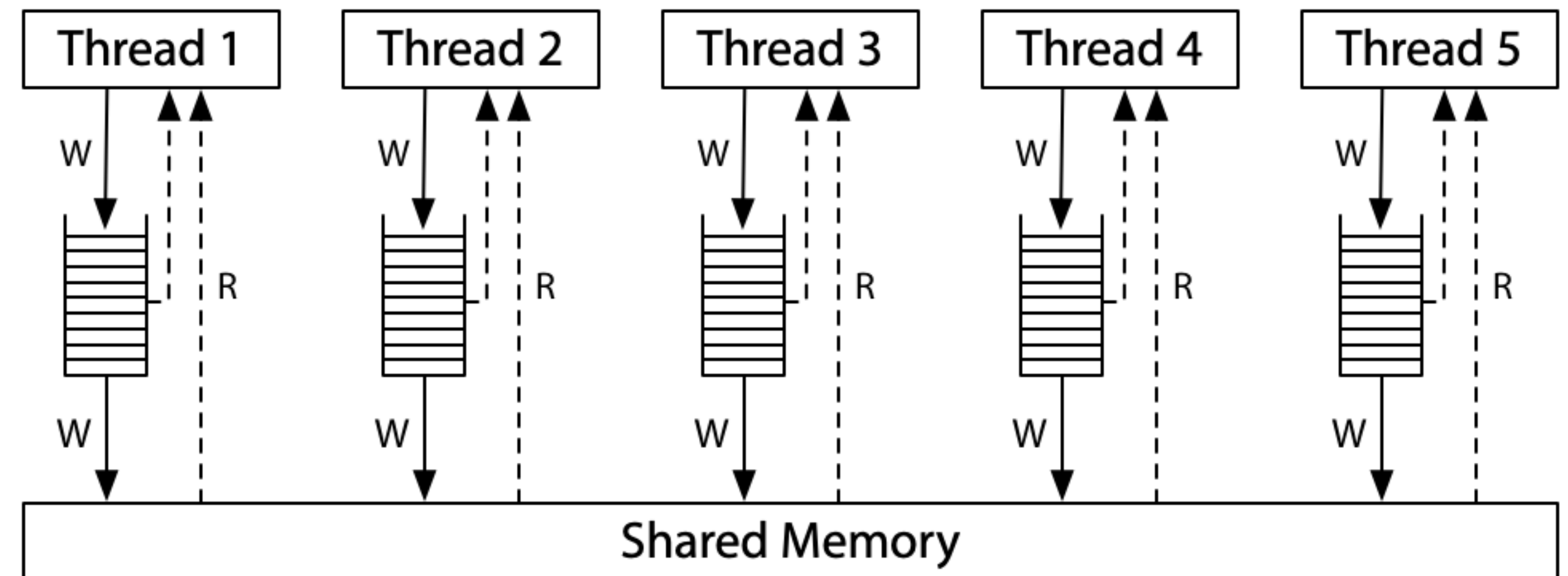
- Correctness analysis assumed “sequential consistency”:
  - CPU is executing one instruction after another
  - All reads and writes are served by shared memory one-at-a-time. No caches!
  - Parallel executions are just interleavings of sequential executions



# CPUs do not follow sequential consistency!

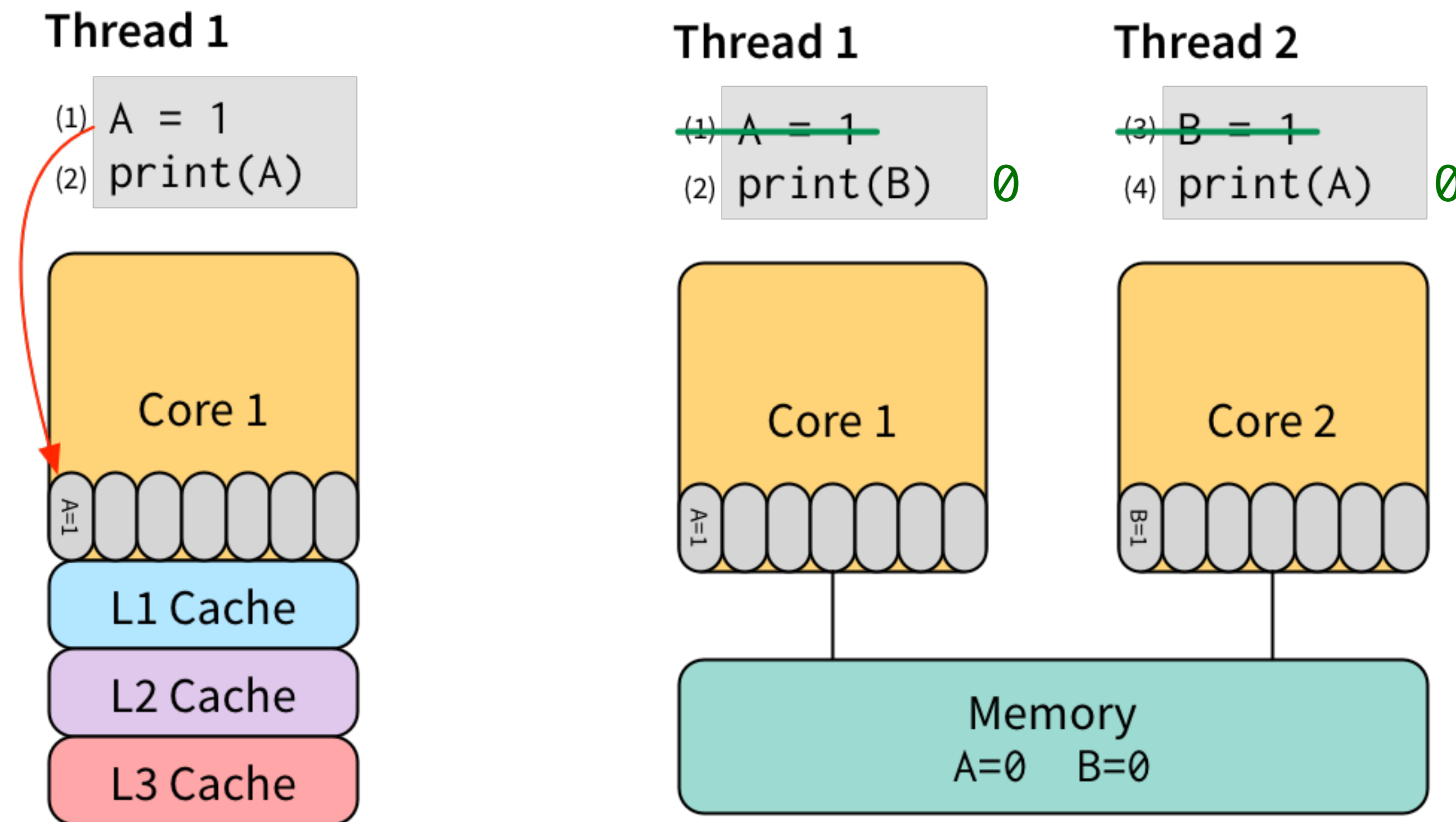
x86 follows “Total Store Order” (TSO)

- CPUs can run instructions out of order
- Writes go to CPU's FIFO store buffer
- Reads first check the local store buffer
- Writes are transmitted lazily to shared memory



# Example weird behaviours from weak memory models

## Memory models define observable behaviours



No interleaving of sequential executions can lead to this outcome!

# Peterson algorithm breaks on x86

peterson-breaks.c

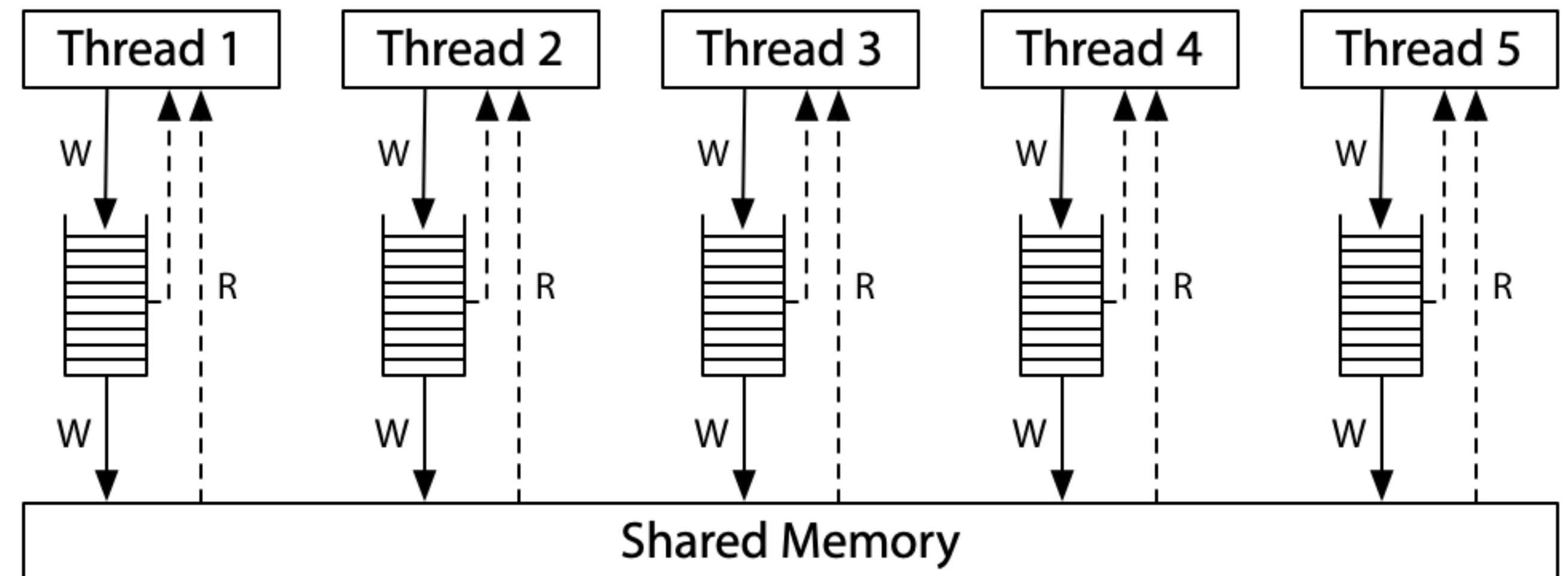
CPU 1	CPU 2
mov \$1, flag[0]	mov \$1, flag[1]
mov \$1, turn	mov \$0, turn
mov flag[1], %eax	mov flag[0], %ebx
mov turn, %ecx	mov turn, %ecx
flag[1] is 1, turn is 1	flag[0] is 1, turn is 1

CPU 1	CPU 2
mov \$1, flag[0]	mov \$1, flag[1]
mov \$1, turn	mov \$0, turn
mov flag[1], %eax	mov flag[0], %ebx
mov turn, %ecx	mov turn, %ecx
flag[1] is 0, turn is 1	flag[0] is 0, turn is 1

flag[\*]=1 is in local store buffer at the time of read

# Fences

- MFENCE: finish all pending reads and writes before continuing. Flush the store buffer before continuing



# Fixing Peterson's algorithm

```
int flag[2];
int turn = 0; // whose turn? (thread 0 or 1?)

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

void lock() {
    flag[self] = 1; // self: thread ID of caller
    turn = 1 - self; // make it other thread's turn

    asm volatile("mfence");
    while ((flag[1 - self] == 1) && (turn == 1 - self)) ; // spin-wait
}

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



# Why does it work?

- mfence forces writes to flag[0] and flag[1] to main memory
- Reads of flag[0] and flag[1] are served from main memory


CPU 1	CPU 2
mov \$1, flag[0]	mov \$1, flag[1]
mov \$1, turn	mov \$0, turn
mfence	mfence
mov flag[1], %eax	mov flag[0], %ebx
mov turn, %ecx	mov turn, %ecx
%eax is 1, %ecx is 1	%ebx is 1, %ecx is 1

flag[\*]=1 **cannot** be in local store buffer at the time of read

# Compiler can reorder and remove instructions!

```
void do() {  
    flag[self] = 1;  
    turn = 1 - self;  
    asm volatile("mfence");  
    while ((flag[1 - self] == 1) && (turn == 1 - self));  
    counter++; // critical section  
    flag[self] = 0;  
}
```

Independent variables from  
compiler's point of view!

Two curved arrows originate from the text 'Independent variables from compiler's point of view!'. The top arrow points to the line 'turn = 1 - self;'. The bottom arrow points to the closing brace '}' of the function 'do()'. This indicates that these two elements are independent of the compiler's point of view.

# Software barriers

```
volatile int flag[2];
```

```
void do() {  
    flag[self] = 1;  
    turn = 1 - self;
```

```
    __sync_synchronize( ); // software + hardware barrier  
    while ((flag[1 - self] == 1) && (turn == 1 - self));
```

```
    asm volatile ("" ::: "memory"); // software barrier  
    counter ++; // critical section
```

```
    asm volatile ("" ::: "memory"); // software barrier  
    flag[self] = 0;
```

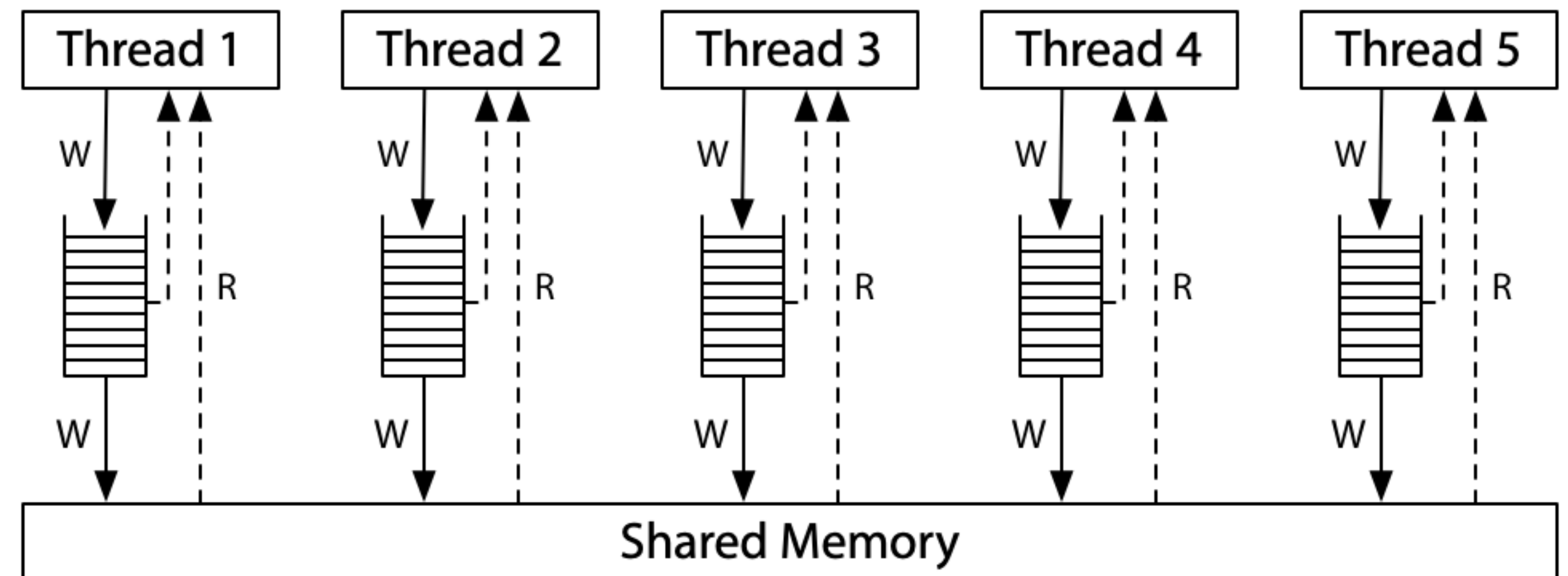
```
}
```

# Why do we NOT need a hardware barrier when releasing?

```
counter++; // critical section
```

```
asm volatile ("" ::: "memory"); // software barrier  
flag[self] = 0;
```

- Store buffer is FIFO!
- If “flag[self] = 0” is visible, then write to counter is also visible!
- Point of long debate in Linux kernel mailing list (1999). Improved some benchmark performance by 4%!



# x86 synchronising instructions

- LOCK <instruction>:
  - MFENCE +
  - Only one CPU can run a locked instruction at a time
  - Example: atomics.c

# Spin locks in xv6

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

    // The xchg is atomic.
    while(xchg(&lk->locked, 1) != 0);
    __sync_synchronize();
    asm volatile ("" ::: "memory");

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

```
static inline uint xchg(volatile uint *addr, uint newval) {
    uint result;

    // The + in "+m" denotes a read-modify-write operand
    asm volatile("lock; xchgl %0, %1" :
        "+m" (*addr), "=a" (result) :
        "1" (newval) :
        "cc");
    return result;
}
```

```
void release(struct spinlock *lk)
{
    if(!holding(lk))
        panic("release");

    lk->pcs[0] = 0;
    lk->cpu = 0;

    __sync_synchronize();
    asm volatile ("" ::: "memory");

    // Release the lock, equivalent to lk->locked = 0.
    asm volatile("movl $0, %0" : "+m" (lk->locked) : );

    popcli();
}
```

# Other metrics

- Safety: Bad things never happen
  - Two threads shall never *simultaneously* acquire the lock
- Liveness: Good things eventually happen
  - Some thread (trying to lock) eventually gets to acquire the lock
- Fairness
  - Do some threads (CPUs) get unfair advantage over others?
- Starvation freedom
  - Is it guaranteed that all threads will eventually get the lock?



# Spin locks in xv6 are not starvation free

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

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

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

```
static inline uint xchg(volatile uint *addr, uint newval) {
    uint result;

    // The + in "+m" denotes a read-modify-write operand
    asm volatile("lock; xchgl %0, %1" :
                 "+m" (*addr), "=a" (result) :
                 "1" (newval) :
                 "cc");
    return result;
}
```

```
void release(struct spinlock *lk)
{
    if(!holding(lk))
        panic("release");

    lk->pcs[0] = 0;
    lk->cpu = 0;

    __sync_synchronize();


    // Release the lock, equivalent to lk->locked = 0.
    asm volatile("movl $0, %0" : "+m" (lk->locked) : );

    popcli();
}
```

# Ticket locks

```
1  typedef struct __lock_t {
2      int ticket;
3      int turn;
4  } lock_t;
5
6  void lock_init(lock_t *lock) {
7      lock->ticket = 0;
8      lock->turn   = 0;
9  }
10
11 void lock(lock_t *lock) {
12     int myturn = FetchAndAdd(&lock->ticket);
13     while (lock->turn != myturn)
14         ; // spin
15 }
16
17 void unlock(lock_t *lock) {
18     lock->turn = lock->turn + 1;
19 }
```

**Atomic instruction**



# What if critical sections are long?

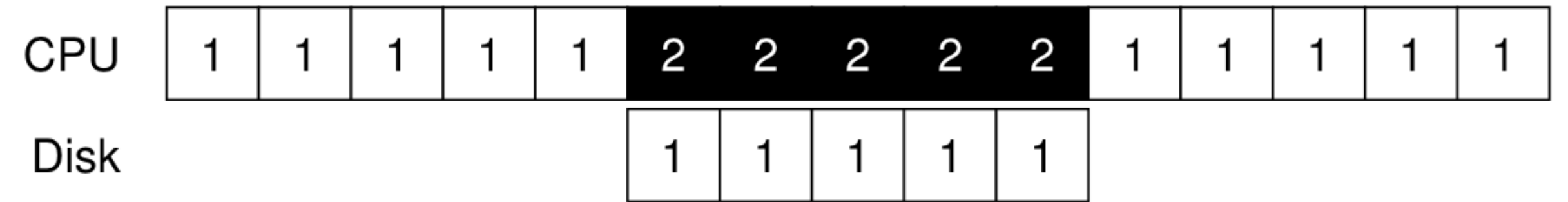
- Spin locks waste CPU time while unnecessary spinning
- Yield — let something else run while we are anyways waiting
- May not know the length of critical section while locking
  - Hybrid locks: spin for some time and then yield
  - Example: futex in linux (Figure 28.10 OSTEP book)

# Yielding: condition variables

## Case study: xv6 disk driver

```
void sleep(void *chan) {
    struct proc *p = myproc();
    // Go to sleep.
    p->chan = chan;
    p->state = SLEEPING;
    sched();
    // Tidy up.
    p->chan = 0;
}

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;
}
```



```
void ideintr(void) {
    struct buf *b = idequeue;
    if(!(b->flags & B_DIRTY) && idewait(1) >= 0)
        insl(0x1f0, b->data, BSIZE/4);
    b->flags |= B_VALID;
    b->flags &= ~B_DIRTY;
    wakeup(b);
}

void iderw(struct buf *b){
    if(idequeue == b)
        idestart(b);
    while((b->flags & (B_VALID|B_DIRTY)) != B_VALID){
        noop();
        sleep(b);
    }
}
```

# Case study: wait/exit

```
void exit(void) {
    struct proc *curproc = myproc();
    struct proc *p;

    wakeup(curproc->parent);
}

void sleep(void *chan) {
    struct proc *p = myproc();
    p->chan = chan;
    p->state = SLEEPING;
    sched();
}

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;
}
```

```
int wait(void) {
    struct proc *p;
    struct proc *curproc = myproc();

    for(;;){
        // Scan through table looking for exited children.
        for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
            if(p->parent != curproc)
                continue;
            if(p->state == ZOMBIE){
                // Found one.
                ...
                return pid;
            }
        }
        // Wait for children to exit
        sleep(curproc);
    }
}
```

# Lost wakeup problem

```
void exit(void) {  
    struct proc *curproc = myproc();  
    struct proc *p;
```

```
    wakeup(curproc->parent);  
}
```

```
void sleep(void *chan) {  
    struct proc *p = myproc();  
    p->chan = chan;  
    p->state = SLEEPING;  
    sched();  
}
```

```
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;  
}
```

eip →

```
int wait(void) {  
    struct proc *p;  
    struct proc *curproc = myproc();  
  
    for(;;){  
        // Scan through table looking for exited children.  
        for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){  
            if(p->parent != curproc)  
                continue;  
            if(p->state == ZOMBIE){  
                // Found one.  
                ...  
                return pid;  
            }  
        }  
        // Wait for children to exit  
        sleep(curproc);  
    }  
}
```