Multithreading

Thierry Sans

Disambiguation

- The textbook talks about managing **processes**
- Pintos does not have processes at all but **"kernel threads"**
- In your system programming class, you could create multiple **"user threads"** under a process

➡ Let's simplify things just for this week :

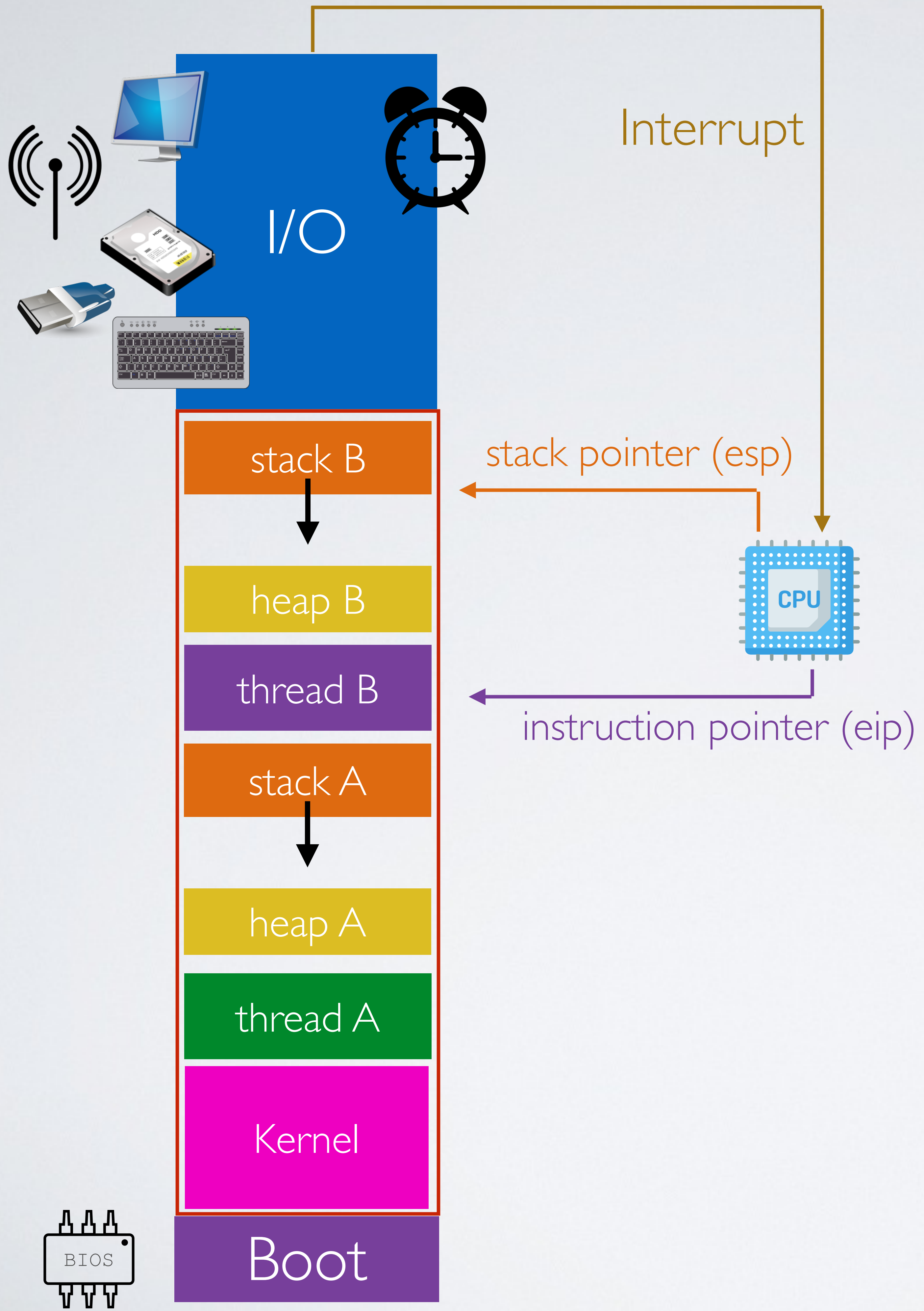
process ~ **thread**

Program vs Thread

- **Program** : static data on some storage
- **Thread** : instance of a program execution

➡ Different threads executing the same program can run concurrently

The architecture



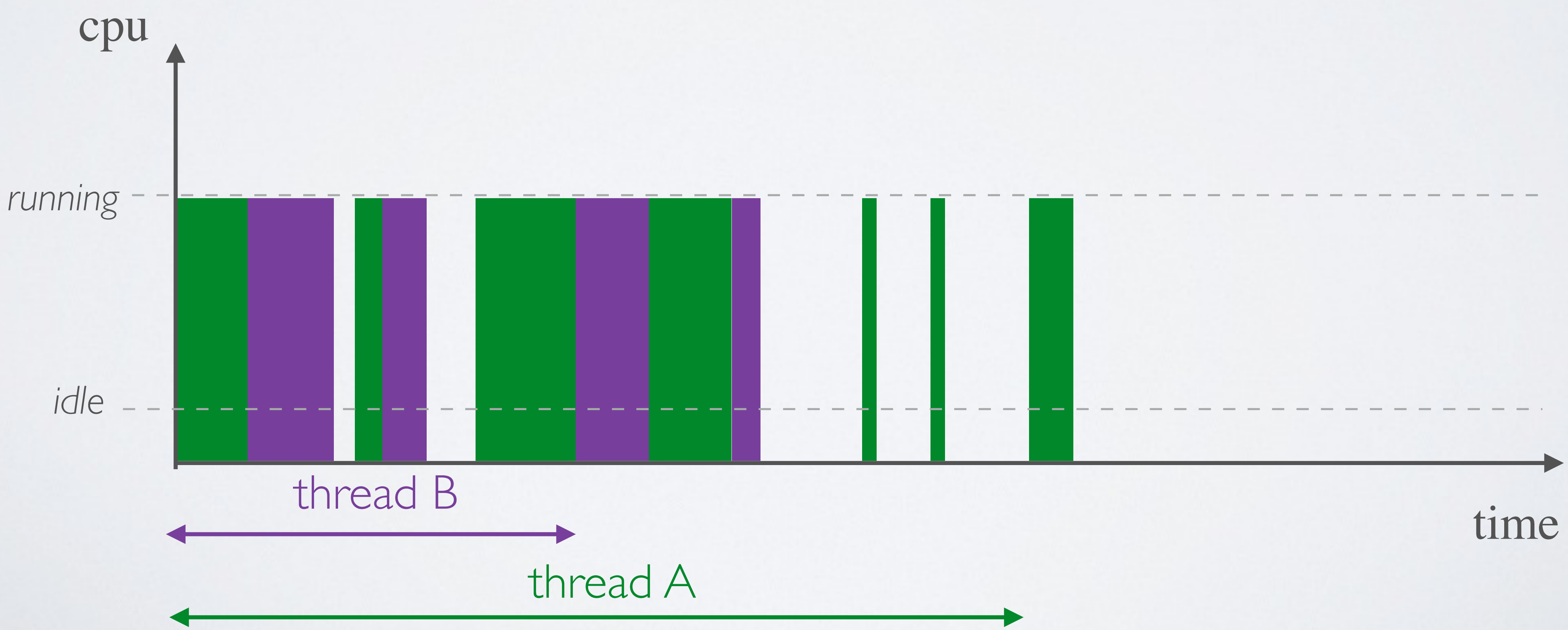
Running threads concurrently

A CPU core will run multiple thread concurrently by running each thread for a little amount of time before switching to another one

➡ **Limited Direct Execution**

The CPU will switch to another thread when either

- the running thread yields the CPU (non-blocking IO for instance)
- or the CPU stops the running thread (system clock interrupt)



The advantages of concurrency

✓ **From the system perspective**
better CPU usage resulting in a faster execution overall (but not individually)

✓ **From the user perspective**
programs seem to be executed in parallel

➡ It requires **some mechanisms to manage and schedule** these concurrent threads

Today's lecture

1. **Interrupts**
2. **Context Switching**
3. **Synchronization**

1. Interrupts

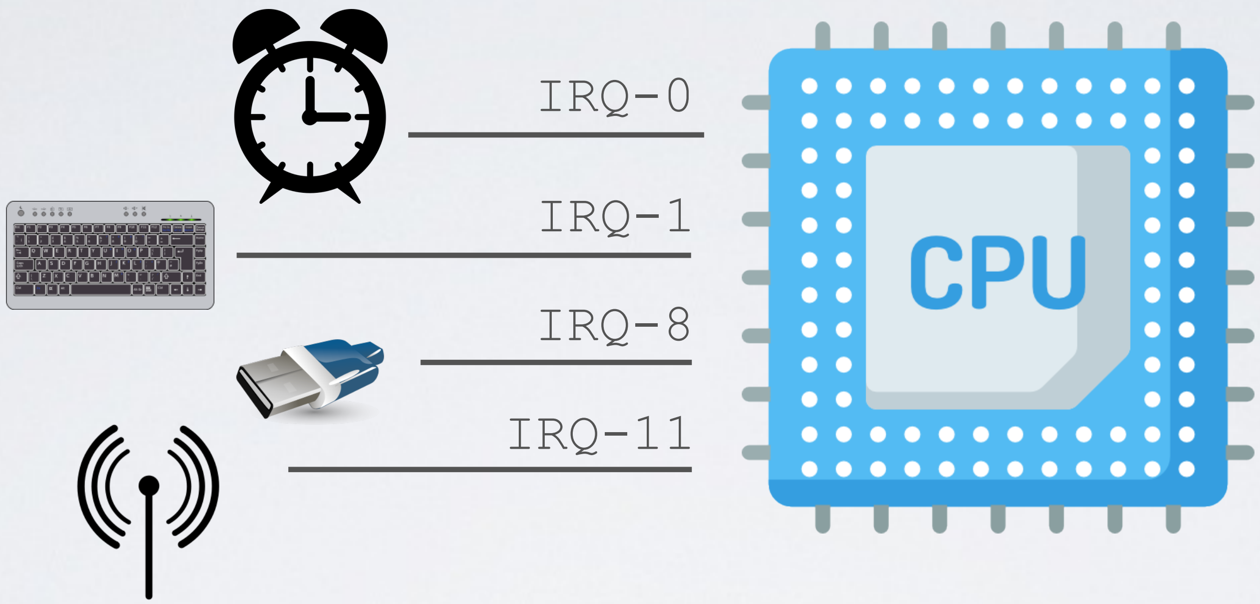
Two kinds of interrupts

External Interrupts a.k.a hardware interrupts
caused by an I/O device that needs some attention (asynchronous)

Internal Interrupts a.k.a system calls, exceptions and faults caused by executing instructions (synchronous)

- fault
  e.g divide by zero
  e.g page fault (coming later with memory management)
- trap - x86 `int` instruction (intended by the programmer)
  e.g `int $0x80` for Linux system call trap
  e.g `int $0x30` for Pintos system call trap

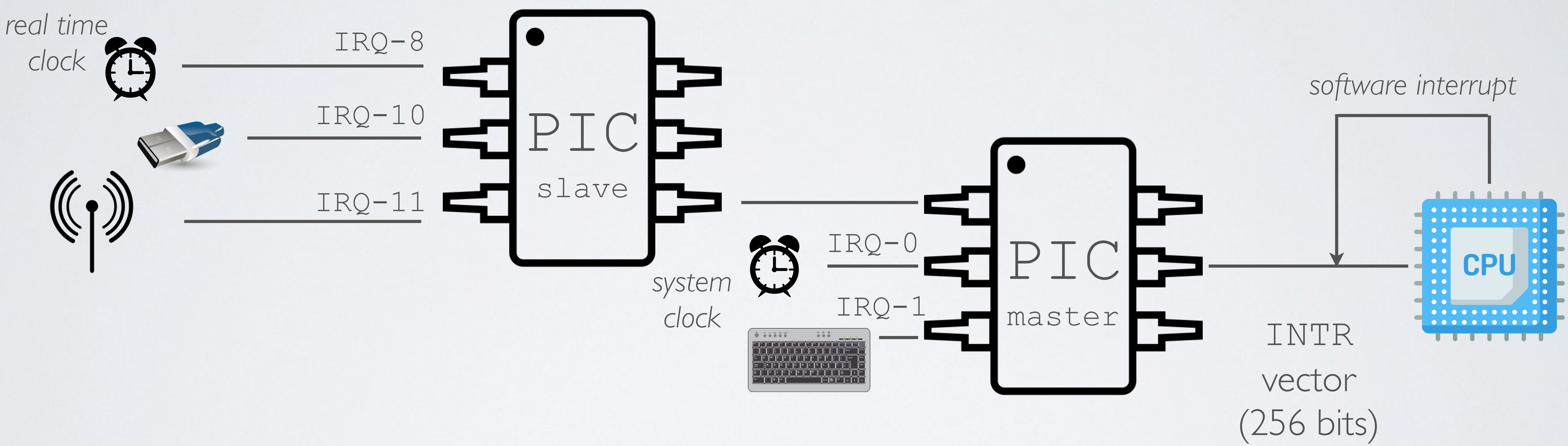
External Interrupt - the naive implementation



➡ I/O devices are wired to **Interrupt Request lines** (IRQs)

- Not flexible (hardwired)
- CPU might get interrupted all the time
- How to handle interrupt priority

Internal Interrupt and External Interrupt - the real implementation

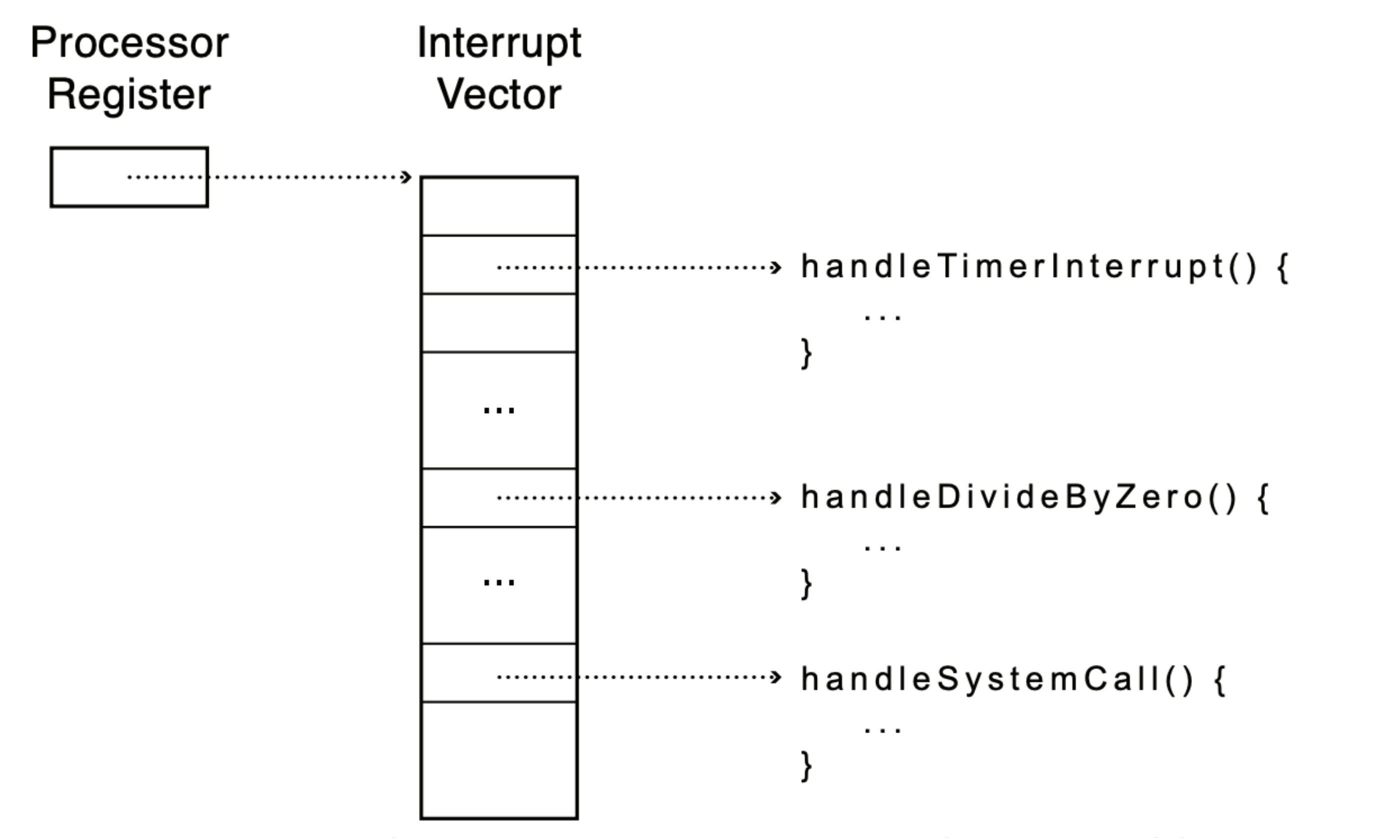


➡ I/O devices have unique or shared IRQs that are managed by two **Programmable Interrupt Controllers** (PIC)

Programmable Interrupt Controllers (PIC)

- ➡ Responsible to tell CPU when and which devices wishes to interrupt through the INTR vector
- ✓ 16 lines of interrupt (IRQ0 - IRQ15)
- ✓ Interrupts have different priority
- ✓ Interrupts can be masked

Handling an interrupt



1. The CPU receives an interrupt on the INTR vector
2. The CPU stops the running program and transfer control to the corresponding handler in the Interrupt Descriptor Table (IDT)
3. The handler saves the current running program state
4. The handler executes the functionality
5. The handler restores (or halt) the running program

Where are these interrupt handlers defined

- **Linux**
  `cat /proc/interrupt`
- **Windows**
  `msinfo32.exe`
- **Pintos**
  `see src/threads/interrupt.c`

Example

When a key is pressed…

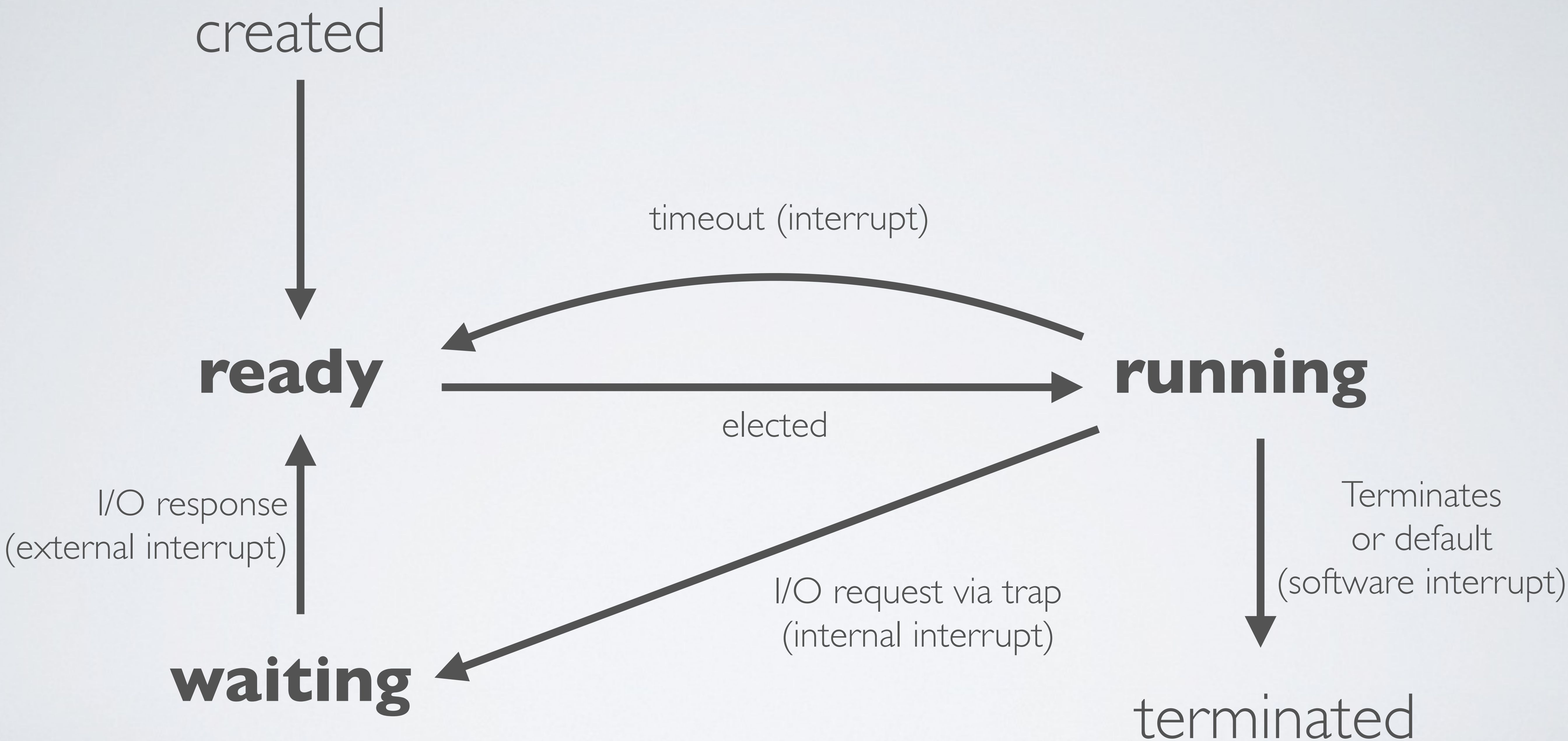
1. the keyboard controller tells PIC to cause an interrupt on IRQ #1
2. the PIC, which decides if CPU should be notified
3. If so, IRQ 1 is translated into a vector number to index into CPU's Interrupt Descriptor Table
4. The CPU stop the current running program
5. The CPU invoke the current handler
6. The handler talks to the keyboard controller via IN and OUT instructions to ask what key was pressed
7. The handler does something with the result (e.g write to a file in Linux)
8. The handler restores the running program

2. Context Switching

When the CPU runs threads concurrently

- Only one thread at a time is **running** (on one core)
- Several threads might be **ready** to be executed
- Several threads might be **waiting** for an I/O response

The different states of a thread



Context switching when

When the OS receives a fault

1. suspends the execution of the running thread
2. terminate the thread

When the OS receives a System Clock Interrupt or a System Call Trap (I/O request)

3. suspends the execution of the running thread
4. saves its execution context
5. changes the thread's state to ready (timeout) or waiting (I/O request)
6. elects a new thread from the ones in the ready state
7. changes its state to running
8. restores its execution context
9. resumes its execution

When the OS receives any other I/O interrupt

1. executes the I/O operation
2. switches the thread, that was waiting for that I/O operation, into the ready state
3. resumes the execution of the current program

➡ **For each thread, the OS needs to keep track of its state (ready, running, waiting) and its execution context (registers, stack, heap and so on)**

Process Control Block

TCB (Thread Control Block)
data structure to record thread information

- Tid (thread id)
- State (as either running, ready, waiting)
- Registers (including eip and esp)
- User (forthcoming lecture on user space)
- Pointer to a Process Control Block (coming next week)

State Queues

- ➡ The OS maintains a collection of queues with the TCBs of all threads
    - One queue for the threads in the ready state
    - Multiple queues for the threads in the waiting state (one queue for each type of I/O requests)

3. Synchronization

Now threads can collaborate but ...

What are these two threads printing?

Ping thread

```
while(1){
  printf("ping\n");
};
```

Pong thread

```
while(1){
  printf("pong\n");
};
```

Too much milk

	Alice	Bob
12:30	Look in the fridge. Out of milk.	
12:35	Leave for store	
12:40	Arrive at store	Look in the fridge. Out of milk.
12:45	Buy milk	Leave for store
12:50	Arrive home, put milk away	Arrive at store
12:55		Buy milk
1:00		Arrive home, put milk away ... oh no!

Beyond milk

X is a global variable initialized to 0

thread 1

```
void foo(){
    x++;
};
```

thread 2

```
void bar(){
    x--;
};
```

What is the value of x after thread 1 and 2?

CPU instruction level

Incrementing (or decrementing) `x` is **not** an atomic operation

thread 1 (`foo` function)

```
LOAD X
INCR
STORE X
```

thread 2 (`bar` function)

```
LOAD X
DECR
STORE X
```

Non-deterministic execution

Execution scenario #1

```
LOAD X
INCR
STORE X
LOAD X
DECR
STORE X
```

➡ X is equal to 0

Execution scenario #2

```
LOAD X
LOAD X
INCR
DECR
STORE X
STORE X
```

➡ X is equal to −1

Execution scenario #3

```
LOAD X
LOAD X
INCR
DECR
STORE X
STORE X
```

➡ X is equal to 1

... and many other possible scenarios with the outcome of x being equal to either 0, −1 or 1

Race-condition problem

The system behaviours depends on the sequence or timing of events that is non-deterministic

- Not desirable in most cases (hard to catch bug)

Mutual Exclusion

We want to use **mutual exclusion** to synchronize access to to shared resources

Code that uses mutual exclusion to synchronize its execution is called a **critical section**

- Only one thread at a time can execute in the critical section
- All other threads are forced to wait on entry
- When a thread leaves a critical section, another can enter

A classical example - Producer Consumer

```
void producer () {
  while(1){
     item := produce()
     while(full(buffer)){
         /* do nothing */
     }
     write(buffer, item)
  }
}
```

```
void consumer () {
  while(1){
     while(emtpy(buffer)){
         /* do nothing */
     }
     item := read(buffer)
     consume(item)
  }
}
```

Critical Section

Requirements

1. **Mutual exclusion**
   If one thread is in the critical section, then no other is

➡ Mutual exclusion ensures ***safety property*** (nothing bad happen)

2. **Progress**
   If some thread T is not in the critical section, then T cannot prevent some other thread S from entering the critical section. A thread in the critical section will eventually leave it.

3. **Bounded waiting** (no starvation)
   If some thread T is waiting on the critical section, then T will eventually enter the critical section

➡ Progress and bounded waiting ensures the ***liveness property*** (something good happen)

4. **Performance**
   The overhead of entering and exiting the critical section is small with respect to the work being done within it

The concept of **lock** (a.k.a mutex)

- The **lock** supports three operations:
  - **`init()`**
    creates an unlocked mutex
  - **`acquire()`**
    waits until the mutex is unlocked, then locks it to enter the C.S
  - **`release()`**
    unlocks the mutex to leave the C.S, waking up anyone waiting for it

(Bad) Producer Consumer **using a lock**

```
lock := init()
```

```
void producer () {
  while(1){
      item := produce()
      acquire(lock)
      write(buffer, item)
      release(lock)
  }
}
```

```
void consumer () {
  while(1){
      acquire(lock)
      item := read(buffer)
      release(lock)
      consume(item)
  }
}
```

- The producer might write into a full buffer
- The consumer might read from an empty buffer

(Good) Producer consumer **using a lock**

```
lock := init()
```

```
void producer () {
  while(1){
     item := produce()
     acquire(lock)
     while(full(buffer)){
         release(lock)
         yield();
         acquire(lock)
     }
     write(buffer, item)
     release(lock)
  }
}
```

```
void consumer () {
  while(1){
     acquire(lock)
     while(emtpy(buffer)){
         release(lock)
         yield();
         acquire(lock)
     }
     item := read(buffer)
     release(lock)
     consume(item)
  }
}
```

Another Synchronization Construct
Condition Variable

A condition variable supports three operations

- **`cond_wait(cond, lock)`**
  unlock the lock and sleep until `cond` is signaled
  then re-acquire `lock` before resuming execution
- **`cond_signal(cond)`**
  signal the condition `cond` by waking up the next thread
- **`cond_broadcast(cond)`**
  signal the condition `cond` by waking up all threads

Producers Consumers **using a condition variable**

```
cond_init(not_full)
cond_init(not_empty)
```

```
void producer () {
 while(1){
  item := produce()
  acquire(mutex)
  while(full(buffer))
     cond_wait(not_full, mutex)
  write(buffer, item)
  cond_signal(not_empty)
  release(mutex)
 }
}
```

```
void consumer () {
 while(1){
  acquire(mutex)
  while(empty(buffer))
     cond_wait(not_empty, mutex)
  item := read(buffer)
  cond_signal(not_full)
  release(mutex)
  consume(item)
 }
}
```

Another Synchronization Construct
Semaphore

An abstract data type to provide mutual exclusion
described by *Dijkstra* in the "*THE multiprogramming system*" in 1968

➡ Semaphores are "integers" that support two operations:

- `Semaphore::P()` decrement, block until semaphore is open
  a.k.a `wait()`, or `sem_wait()`, or `sema_down()`
- `Semaphore::V()` increment, allow another thread to enter
  a.k.a `signal()`, or `sem_post()`, or `sema_up()`

✓ Semaphore safety property
the semaphore value is always greater than or equal to 0

Blocking mechanism

Associated with each semaphore is a queue of waiting threads

➡ When `P()` is called by a thread:

- If semaphore is open, thread continue
- If semaphore is closed, thread blocks on queue

➡ Then `V()` opens the semaphore

- If a thread is waiting on the queue, the thread is unblocked
- If no threads are waiting on the queue, the signal is remembered for the next thread

(Bad) Producer Consumer **using a semaphore**

```
sem_init(not_full, n)
sem_init(not_empty, 0)
```

```
void producer () {
  while(1){
     item := produce()
     sem_wait(not_full)
     write(buffer, item)
     sem_signal(not_empty)
  }
}
```

```
void consumer () {
  while(1){
     sem_wait(not_empty)
     item := read(buffer)
     sem_signal(not_full)
     consume(item)
  }
}
```

- Producer and consumer can be in the critical section at the same time

(Bad) Producer consumer **using a semaphore**

```
sem_init(not_full, n)
sem_init(not_empty, 0)
sem_init(mutex, 1)
```

```
void producer () {
  while(1){
      item := produce()
      sem_wait(mutex)
      sem_wait(not_full)
      write(buffer, item)
      sem_signal(not_empty)
      sem_signal(mutex)
  }
}
```

```
void consumer () {
  while(1){
      sem_wait(mutex)
      sem_wait(not_empty)
      item := read(buffer)
      sem_signal(not_full)
      sem_signal(mutex)
      consume(item)
  }
}
```

- **Deadlock** : the producer waits for the consumer to release `mutex` while the consumer waits for producer to release `not_empty` (or vice versa)

Deadlock

Deadlock when one thread tries to access a resource that a second holds, and vice-versa

- They can never make progress

```
void thread1 (){
      ...
      sem_wait(sem1)
      sem_wait(sem2)
      /* critical section */
      sem_signal(sem2)
      sem_ignal(sem1)
      ...
}
```

```
void thread2(){
      ...
      sem_wait(sem2)
      sem_wait(sem1)
      /* critical section */
     sem_signal(sem1)
     sem_signal(sem2)
...
}
```

(Good) Producers Consumer **using semaphores**

```
sem_init(not_full, n)
sem_init(not_empty, 0)
sem_init(mutex, 1)
```

```
void producer () {
  while(1){
     item := produce()
     sem_wait(not_full)
     sem_wait(mutex)
     write(buffer, item)
     sem_signal(mutex)
     sem_signal(not_empty)
  }
}
```

```
void consumer () {
  while(1){
     sem_wait(not_empty)
     sem_wait(mutex)
     item := read(buffer)
     sem_signal(mutex)
     sem_signal(not_full)
     consume(item)
  }
}
```

How to avoid deadlocks

Avoiding deadlock using primitive synchronization mechanisms (locks and semaphores) **is hard** (cf chapter 32)

Implementing synchronization constructs

Two approaches :

- Either implement locks first (Linux approach) and build semaphores and condition variable on the top
  - ➡ Linux has two versions
    - Spinlock (non-blocking)
    - Mutex (blocking)
- Or implement semaphores first (Pintos approach) and build locks and condition variable on top
  - ➡ Pintos approach

(bad) implementation of a spin lock

```
struct lock {
    int held = 0;
}

void acquire (lock) {
    while (lock->held);
    lock->held = 1;
}

void release (lock) {
    lock->held = 0;
}
```

What is the context switch happens in between?

➡ We have a race condition

The hardware to the rescue

* `test-and-set`(`TAS` x86 CPU instruction) atomically writes to the memory location and returns its old value in a **single indivisible step**

➡ the caller is responsible for testing if the operation has succeeded or not

```
bool test_and_set(bool *flag) {
  bool old = *flag;
  *flag = True;
  return old;
}
```

This is pseudo-code!
The hardware execute this atomically

(good) implementation of a spin lock

```
struct lock {
    int held = 0;
}

void acquire (lock) {
    while test-and-set(&lock->held);
}

void release (lock) {
    lock->held = 0;
}
```

Busy wait (a.k.a spin)

- Waste of CPU time
- Unfair access to lock

(bad) implementation of a sleeping lock

```
struct lock {
}

void acquire (lock) {
    disable_interrupts();
}

void release (lock) {
    enable_interrupts();
}
```

- ➡ Disabling interrupts blocks notification of external events that could trigger a context switch
- ◉ Can miss or delay important events
- ◉ The thread is no longer preemptive

(good) implementation of a sleeping lock

```
struct lock {
    int held = 0;
    queue Q;
}

void acquire (lock) {
     disable_interrupts();
     while (lock->held){
         enqueue(lock->Q, current_thread);
         thread_block(current_thread);
     }
     lock->held = 1;
     enable_interrupts();
}

void release (lock) {
    disable_interrupts();
    if (!isEmpty(lock->Q)){
        thread_unblock(dequeue(lock->Q));
    }
    lock->held = 0;
    enable_interrupts();
}
```

Semaphore Implementation

```
struct semaphore {
    int value;
    queue Q;
}

void init(sema, value){
    sema->value = value;
}

void P (sema) {
    disable_interrupts();
    while (sema->value == 0){
        enqueue(sema->Q, current_thread);
        thread_block(current_thread);
    }
    sema->value--;
    enable_interrupts();
}

void V (sema) {
    disable_interrupts();
    if (!isEmpty(sema->Q)){
        thread_unblock(dequeue(sema->Q));
    }
    sema->value++;
    enable_interrupts();
```

Other interesting synchronization problems

Readers Writers

➡ allow multiple readers but only one writer in the critical section

```
void writer () {
   while(1){
       write(file, data);
   }
}
```

```
void reader () {
   while(1){
       data:= read(file);
   }
}
```

Solution

1. **`readcount`** (variable) to keep track of the number of readers currently reading

2. **`mutex`** (binary semaphore) to synchronize the access to `readcount`

3. **`writer_or_readers`** (binary semaphore) to provide exclusive access to each writer or all readers
   - writer should wait before writing and signal after
   - readers should wait when `readcount` goes from 0 to 1 and signal when `readcount` goes from 1 to 0

Readers Writers

```
readcount = 0
sem_init(mutex, 1)
sem_init(writer_or_readers, 1)
```

```
void writer () {
  while(1){
    sem_wait(writer_or_readers)
    write(file, data)
    sem_signal(writer_or_readers)
  }
}
```

```
void reader () {
 while(1){
  sem_wait(mutex)
  readcount += 1;
  if (readcount == 1)
    sem_wait(writer_or_readers)
  sem_signal(mutex)
  data:=read(file)
  sem_wait(mutex)
  readcount -= 1;
  if (readcount == 0)
    sem_signal(writer_or_readers)
  sem_signal(mutex)
 }
}
```

- **Writers starvation!**

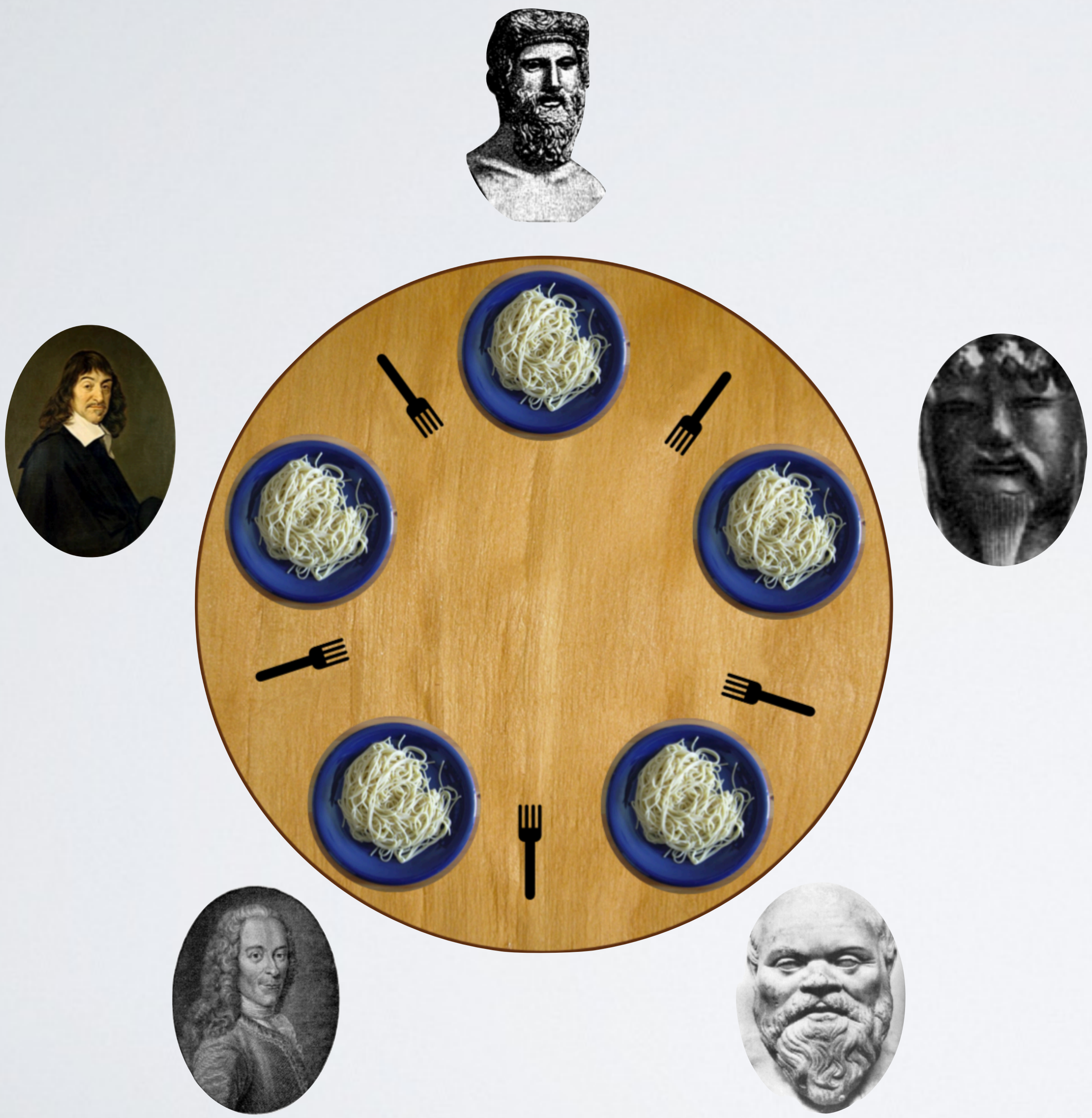
Readers Writers

```
readcount = 0
sem_init(mutex, 1)
sem_init(writer_or_readers, 1)
sem_init(service, 1)
```

```
void writer () {
  while(1){
    sem_wait(service)
    sem_wait(writer_or_readers)
    sem_signal(service)
    write(file, data)
    sem_signal(writer_or_readers)
  }
}
```

```
void reader () {
 while(1){
  sem_wait(service)
  sem_wait(mutex)
  readcount += 1;
  if (readcount == 1)
    sem_wait(writer_or_readers)
  sem_signal(service)
  sem_signal(mutex)
  data:=read(file)
  sem_wait(mutex)
  readcount -= 1;
  if (readcount == 0)
    sem_signal(writer_or_readers)
  sem_signal(mutex)
```

Dining Philosophers



```
void philosopher (i, n) {
  while(1){
    grab_fork(i)
    grab_fork((i + 1)% n)
    eat & think
    drop_fork(i)
    drop_fork((i + 1)% n)
  }
}
```

image from *wikipedia*

(Bad) Dining Philosophers

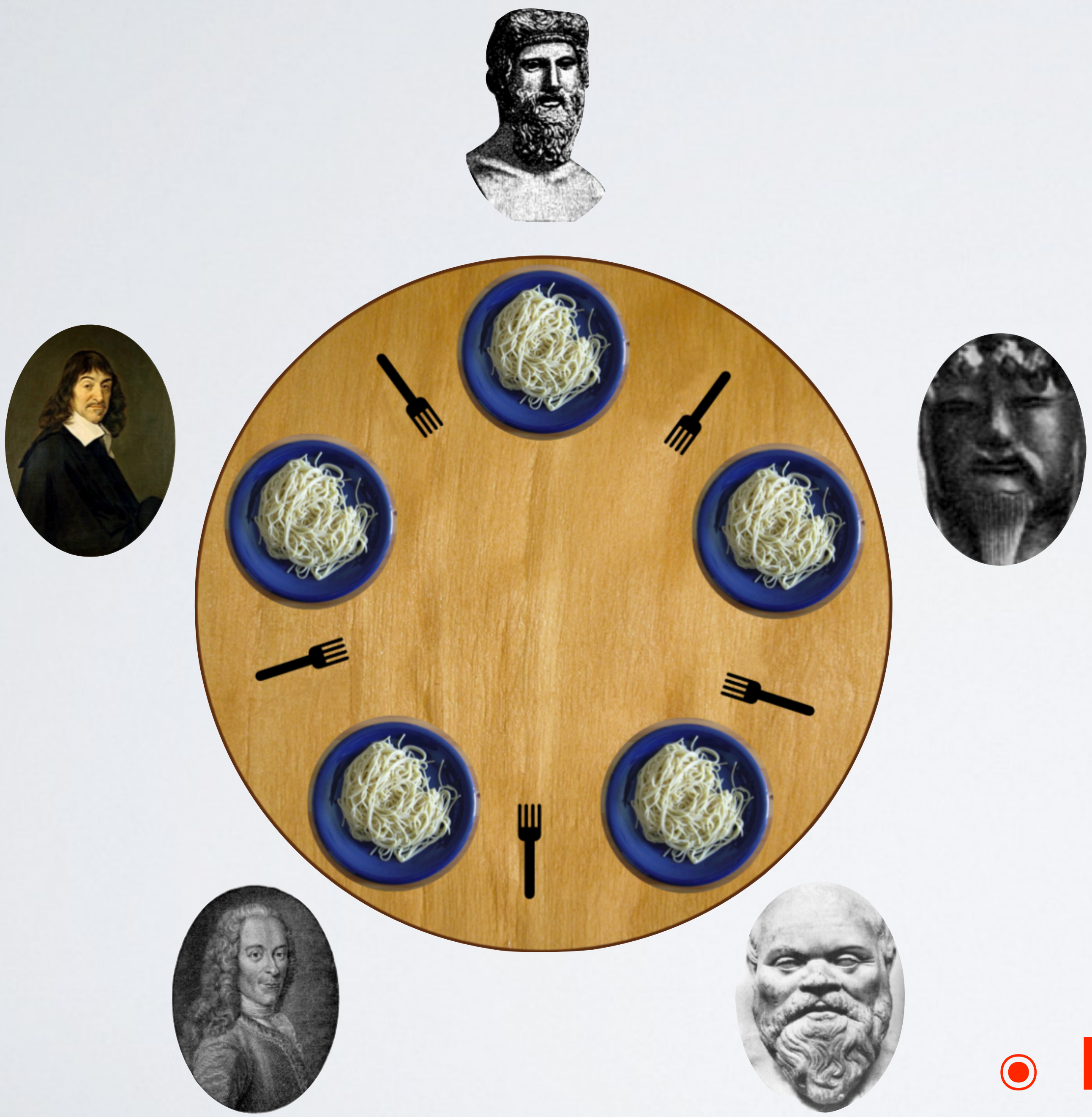


image from *wikipedia*

```
for(i=0, i<n, i++){
   sem_init(fork[i], 1)
}
```

```
void philosopher (i, n) {
   while(1){
      sem_wait(fork[i])
      sem_wait(fork[(i + 1)% n])
      eat & think
      sem_signal(fork[i])
      sem_signal(fork[(i + 1)% n])
   }
}
```

- **Deadlock** when each philosopher take the first fork "at the same time"

(Good) Dining Philosophers

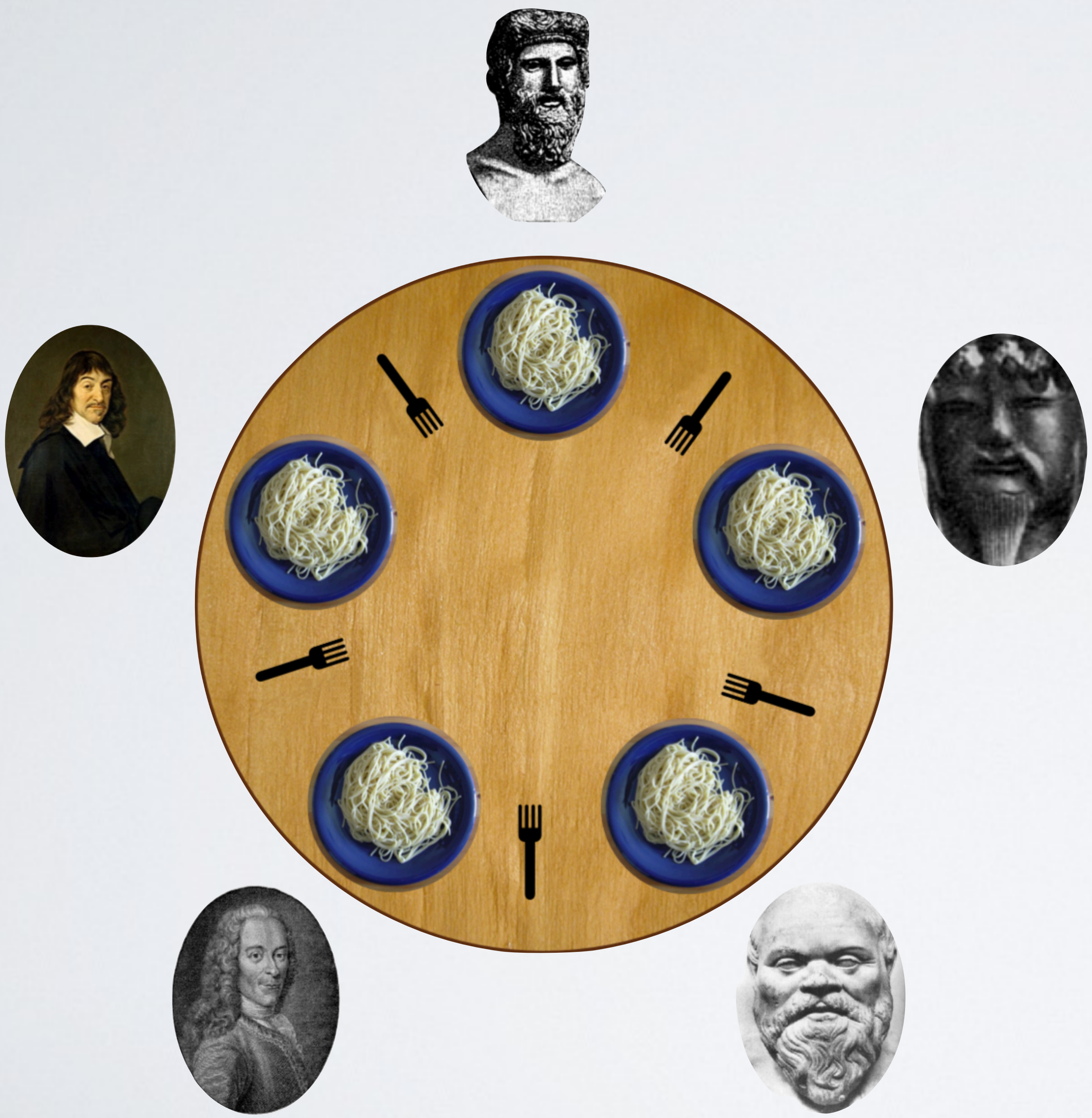


image from *wikipedia*

```
for(i=0, i<n, i++){
   init(fork[i], 1)
}
```

```
void philosopher (i, n) {
  while(1){
    if ((i+1) == n){
      sem_wait(fork[(i + 1)% n])
      sem_wait(fork[i])
    }else{
      sem_wait(fork[i])
      sem_wait(fork[(i + 1)% n])
    }
    eat & think
    sem_signal(fork[i])
    sem_signal(fork[(i + 1)% n])
  }
}
```

Acknowledgments

Some of the course materials and projects are from

- Ryan Huang - teaching CS 318 at *John Hopkins University*
- David Mazière - teaching CS 140 at *Stanford*