CPSC 213

Introduction to Computer Systems

Summer Session 2019, Term 2

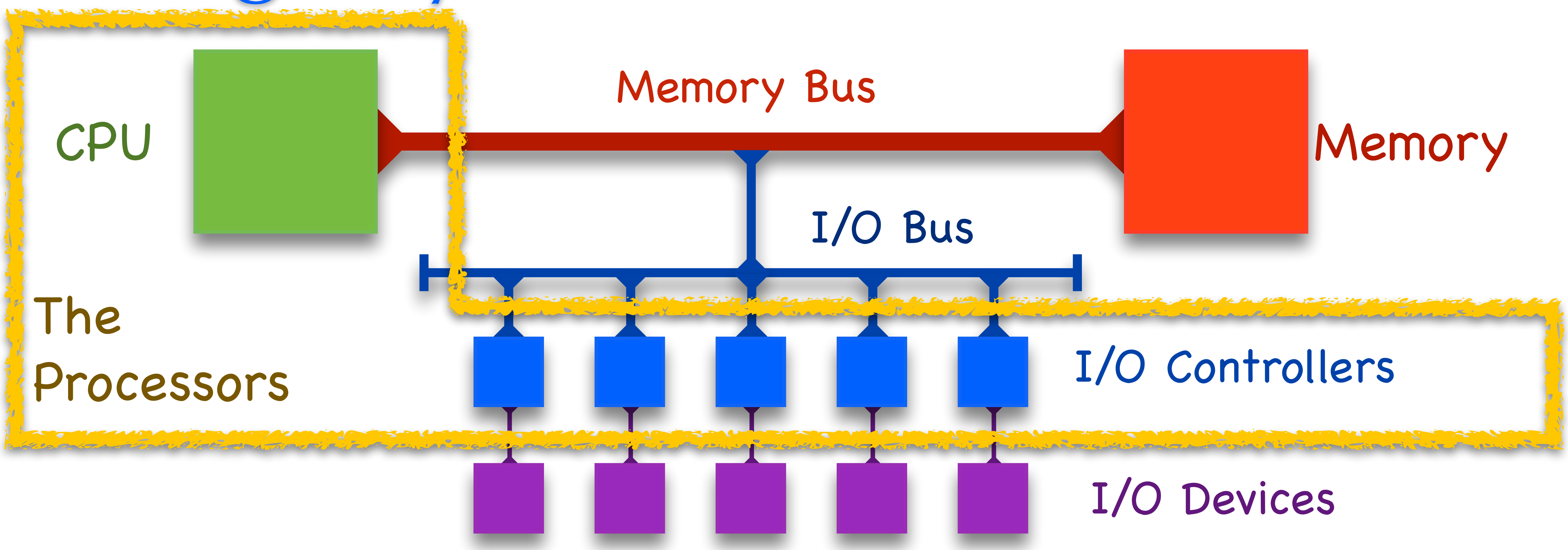
Unit 2a - Jul 30

I/O Devices, Interrupts and DMA

Overview

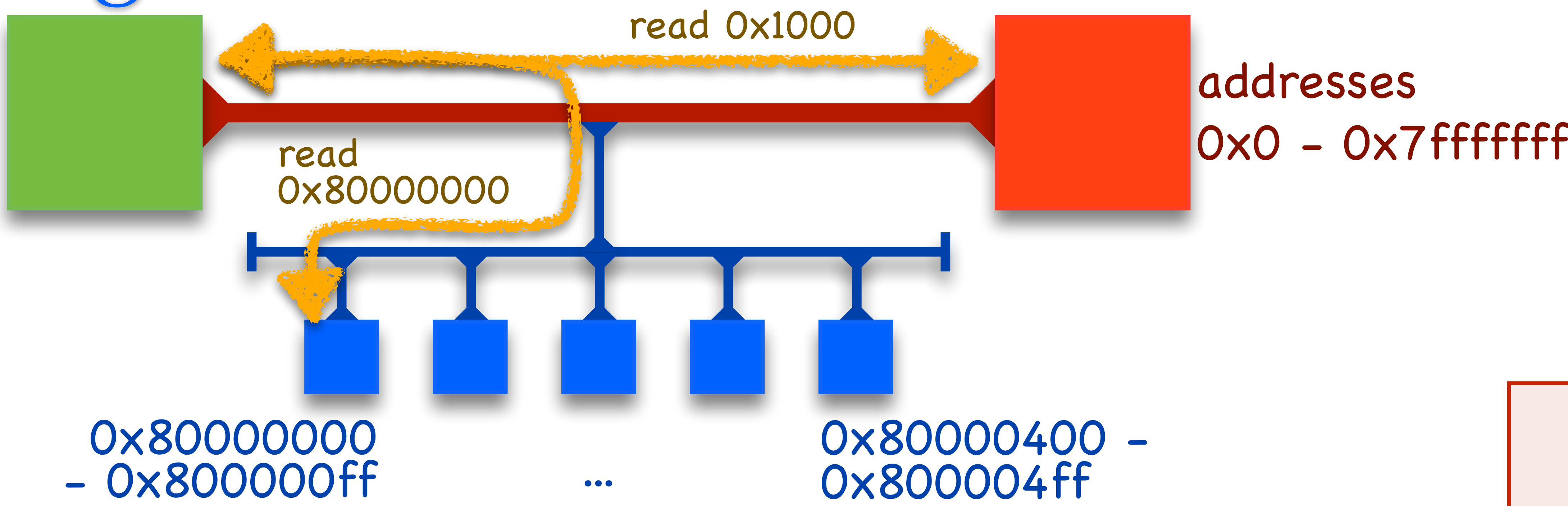
- Reading in Text
  - 8.1, 8.2.1, 8.5.1-8.5.3
- Learning Goals
  - Explain what PIO is, why it exists, what processor originates it, and how it is used
  - Explain what DMA is, why it exists, what it does, and what processor originates it
  - Explain what an interrupt is, why it exists, what it does, and what processor originates it
  - Compare PIO and DMA by identifying when each *can* be used and when each *should* be used
  - Explain the relationship between polling, PIO and interrupts
  - Explain the conditions that make polling acceptable or not
  - Explain what happens when an interrupt occurs at the hardware level
  - Explain what is means for an operation such as a disk read to be asynchronous and give examples of code that works when an operation is synchronous but does not work with it is asynchronous
  - Write event-driven programs in C using function pointers
  - Describe why event-driven programs may be harder to write, read, and debug

Looking Beyond the CPU and Memory

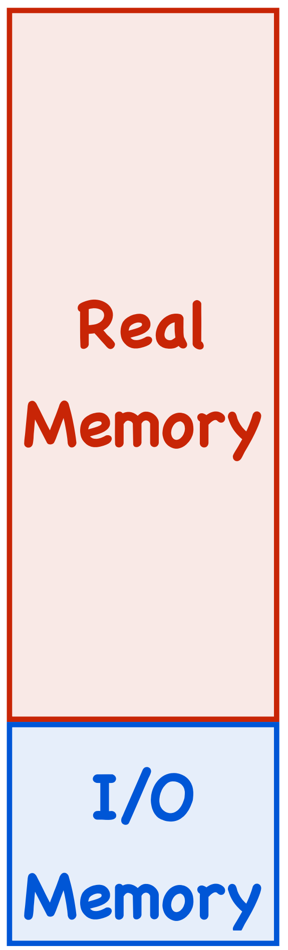


- Memory Bus
  - data/control path connecting CPU, Main Memory, and I/O Bus
  - also called the *Front Side Bus*
- I/O Bus
  - data/control path connecting Memory Bus and I/O Controllers
  - e.g., PCI
- I/O Controller
  - a processor running software (firmware)
  - connects I/O Device to I/O Bus
  - e.g. ,SCSI, SATA, Ethernet, ...
- I/O Device
  - I/O mechanism that generates or consumes data
  - e.g., disk, radio, keyboard, mouse, ...

Talking to an I/O Controller



- Programmed I/O (PIO)
  - CPU transfers a word at a time between CPU and I/O controller
  - typically use standard load/store instructions, but to I/O-mapped memory
- I/O-Mapped Memory
  - memory addresses outside of main memory
  - used to name I/O controllers (usually configured at boot time)
  - loads and stores are translated into I/O-bus messages to controller
- Example
  - to read/write to controller at address `0x80000000`




```
ld   $0x80000000, r0
st   r1 (r0)          # write the value of r1 to the device
ld   (r0), r1         # read a word from device into r1
```

Limitations of PIO

- Reading or writing large amounts of data slows CPU
  - requires CPU to transfer one word at a time
  - controller/device is much slower than CPU
  - and so, CPU runs at controller/device speed, mostly waiting for controller
- IO Controller cannot initiate communication
  - sometimes the CPU asks for data
  - but, sometimes controller receives data for the CPU, without CPU asking
    - e.g., mouse click or network packet reception (everything is like this really as we will see)
  - how does controller notify CPU that it has data the CPU should want?
- One idea...
  - what is it? ____
  - what are drawbacks? ________________________________
  - when is it okay? ___________________________________

Polling and I/O Memory


```
int pollDeviceForValue() {
  volatile int *ready = 0x80000100;
  volatile int *value = 0x80000104;
  int           v;

  while(*ready == 0) {};
  v      = *value;
  *ready = 0;

  return v;
}
```

```
void readyValueForCPUPoll(int v) {
  volatile int *ready = 0x80000100;
  volatile int *value = 0x80000104;

  while(*ready != 0) {};
  *value = v;
  *ready = 1;
}

readyValueForCPUPoll (7);
```

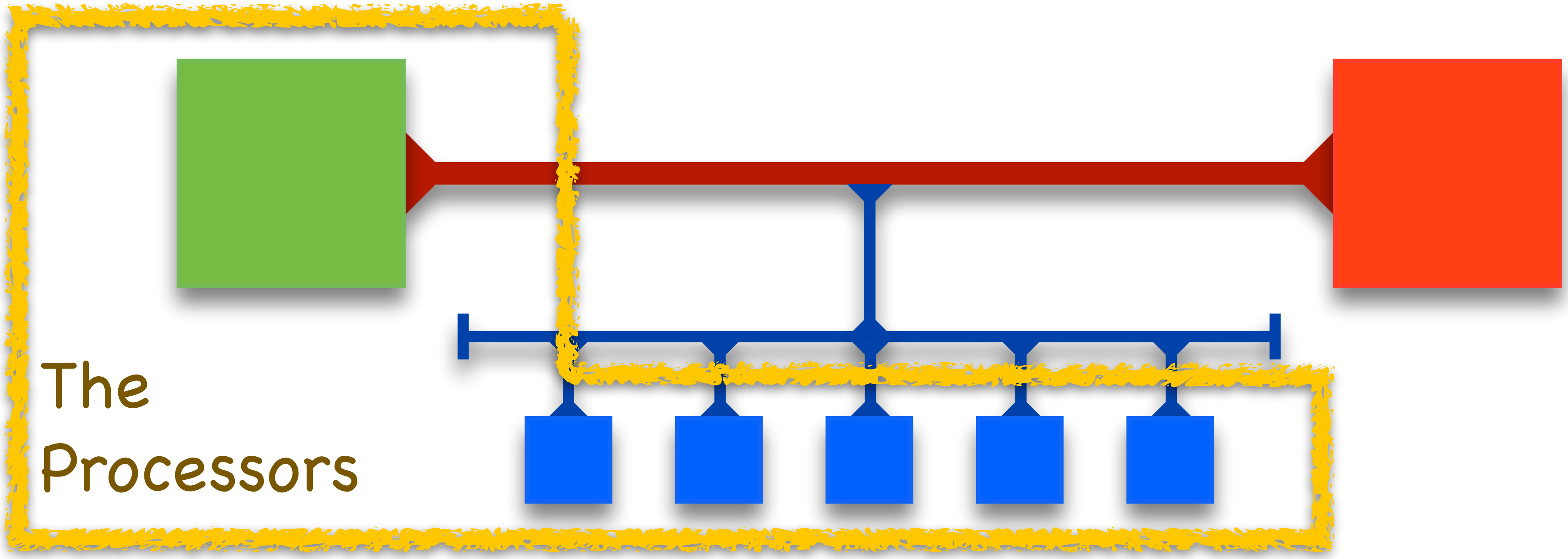
Reading (or writing) I/O Memory IS SLOW

CPU -> I/O Device: give me value at address X

I/O Device -> CPU: here is value

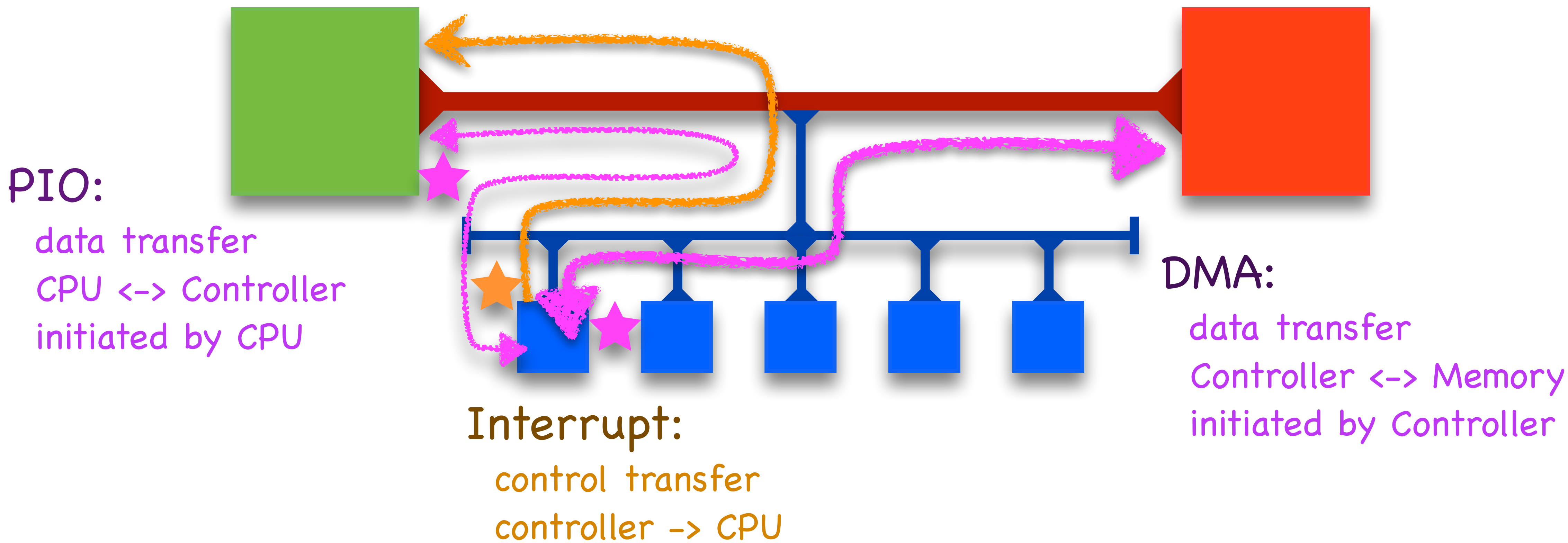
Polling is Okay If poll has low overhead or if high-probability of "hit".

Key Observation



- CPU and I/O Controller are independent processors
  - they should be permitted to work in parallel
  - either should be able to initiate data transfer to/from memory
  - either should be able to signal the other to get the other's attention

Autonomous Controller Operation



- Direct Memory Access (DMA)
  - controller can send/read data from/to any main memory address
  - the CPU is oblivious to these transfers
  - DMA addresses and sizes are *programmed* by CPU using PIO
- CPU Interrupts
  - controller can signal the CPU
  - CPU checks for interrupts on every cycle (its like a really fast, clock-speed poll)
  - CPU jumps to controller's *Interrupt Service Routine* if it is interrupting

Adding Interrupts to Simple CPU

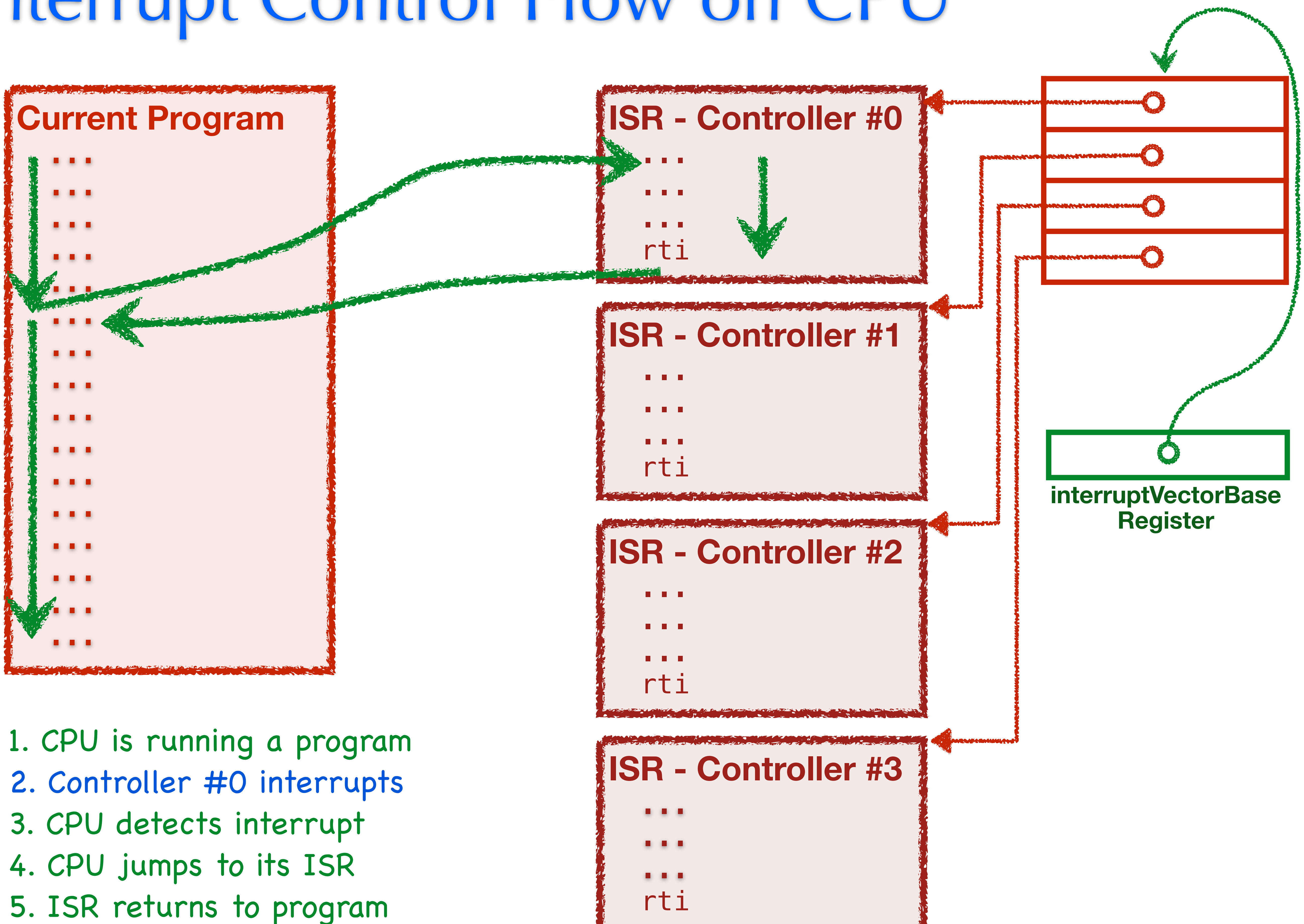
- New special-purpose CPU registers
  - **isDeviceInterrupting** set by I/O Controller to signal interrupt
  - **interruptControllerID** set by I/O Controller to identify interrupting device
  - **interruptVectorBase** interrupt-handler jump table, initialized a boot time
- Modified fetch-execute cycle

```
while(true) {
  if(isDeviceInterrupting) {
    r[5] ← r[5] – 4;  m[r[5]] ← r[6];
    r[6] ← pc;
    pc   ← interruptVectorBase[interruptControllerID];
  }
  fetch();
  execute();
}
```

RTI: Return from Interrupt Instruction

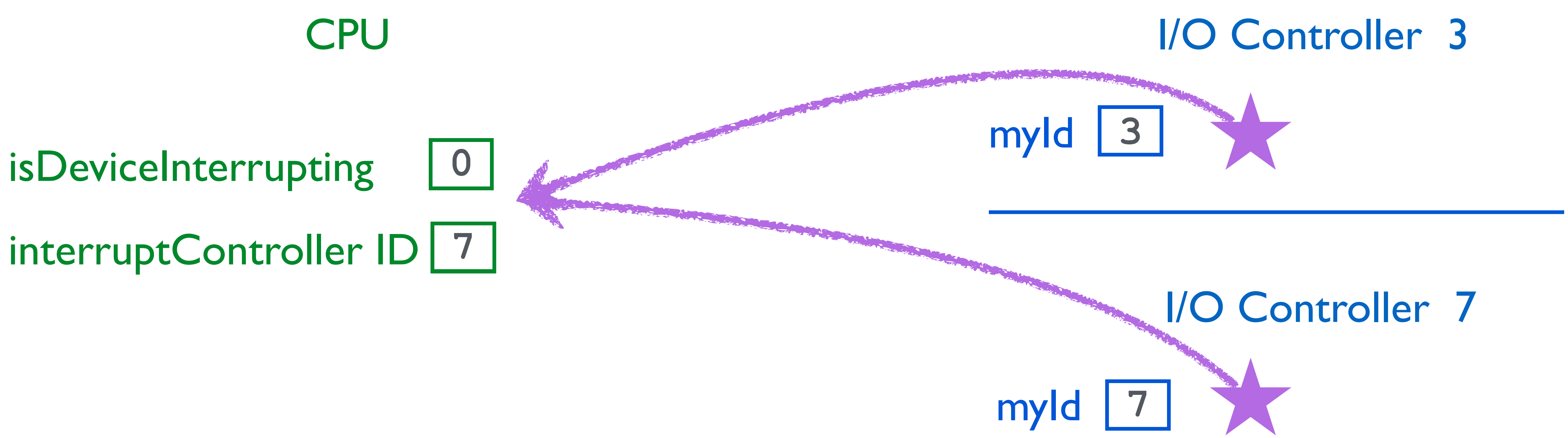
```
t ← r[6];
r[6] ← m[r[5]]; r[5] ← r[5] + 4;
isDeviceInterrupting ← 0;
pc ← t;
```

Interrupt Control Flow on CPU



1. CPU is running a program
2. Controller #0 interrupts
3. CPU detects interrupt
4. CPU jumps to its ISR
5. ISR returns to program

Architectural View of Interrupts




```
while(true) {
  if(isDeviceInterrupting) {
    r[5] ← r[5] – 4;  m[r[5]] ← r[6];
    r[6] ← pc;
    pc   ← interruptVectorBase[interruptControllerID];
    isDeviceInterrupting ← 0;
  }
  fetch();
  execute();
}
```

Polling on CPU register instead of I/O Memory: turning bad polling into good polling

Programming with I/O

Disk Read Timeline

CPU

1. PIO to request read

...
do other things
...

6. Interrupt Received
   Call readComplete

I/O Controller

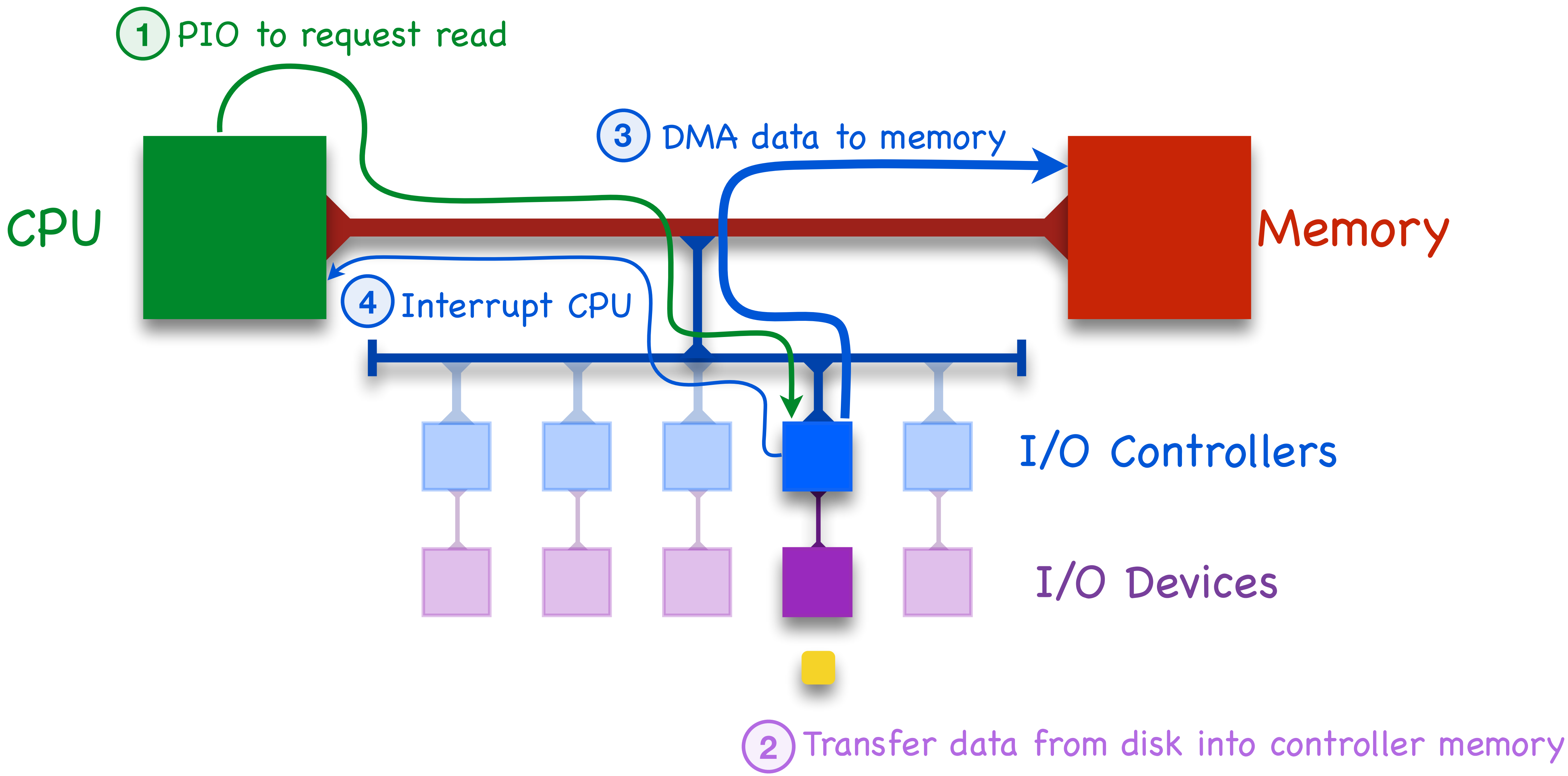
2. PIO Received, start read

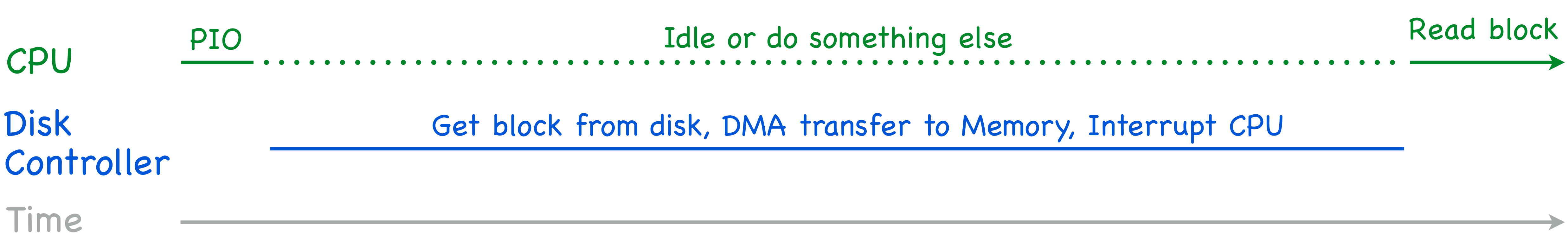
...
wait for read to complete
...

3. Read completes
4. Transfer data to memory (DMA)
5. Interrupt CPU

A disk stores blocks of data. Each block has a number. CPU can read or write.

Disk Read Timeline





The Power of the Sequence

- Consider a program that reads data from a disk
  - you can sort of think of the read has having three parts:
    1. figure out what data you want
    2. get the data
    3. do something with the data you got
  - these steps must happen in this order
  - we think of these steps as a sequence
- For Example …

```
int sumDiskData(blockNum, numBytes) {
  char buf[numBytes];
  int sum = 0;

  diskRead(buf, blockNum, numBytes);
  for(int i=0; i<numBytes; i++)
    sum += buf[i];

  return sum;
}
```

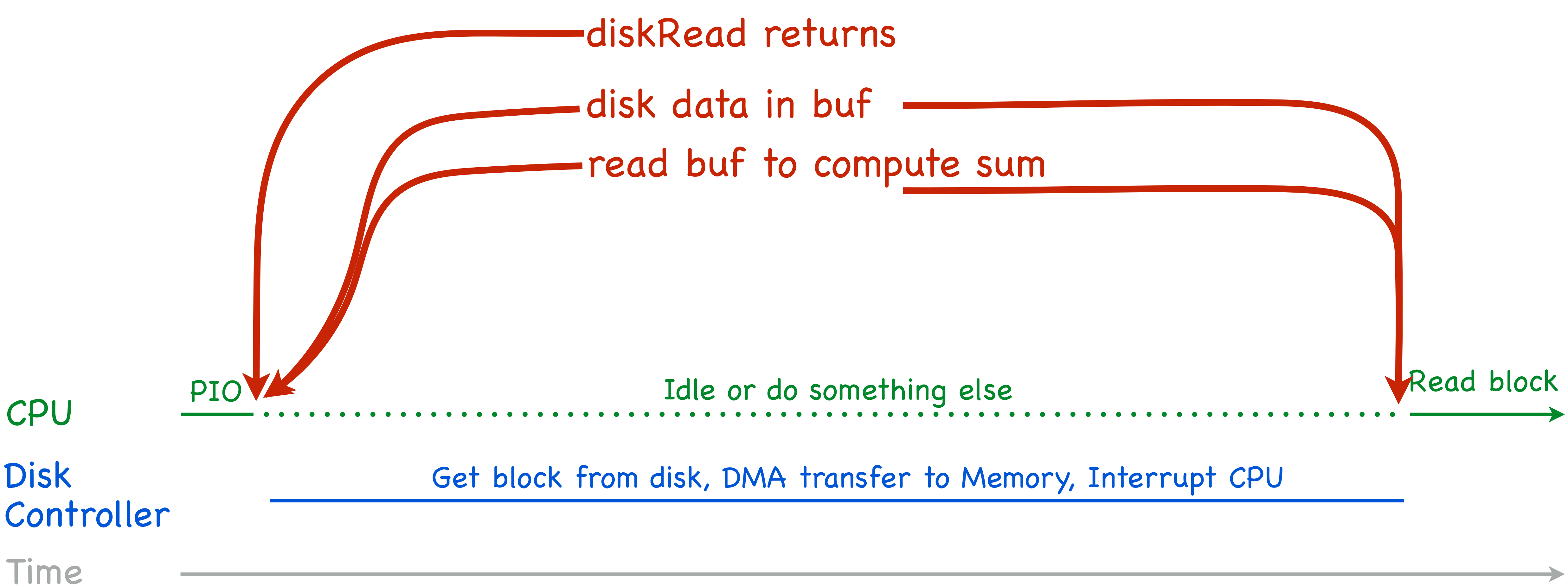
… Meets Reality

```
int sumDiskData(blockNum, numBytes) {
  char buf[numBytes];
  int sum = 0;

  diskRead(buf, blockNum, numBytes);
  for(int i=0; i<numBytes; i++)
    sum += buf[i];

  return sum;
}
```

must wait for read to complete



Making Code Asynchronous

This code requests something

```
char buf[numBytes]
diskRead (buf, blockNum, numBytes);
```

2

These two pieces of code are not listed sequentially … they are not synchronous

```
int sum = 0;
for (int i=0; i<numBytes; i++)
    sum += buf [i];
```

This code runs when that thing completes

1 PIO

Idle or do something else

3 Read block

CPU

Disk Controller

Get block from disk, DMA transfer to Memory, Interrupt CPU

Time

Writing Asynchronous Code in C

Events and Event Handlers

- the things that causes asynchronous code to run are called ***events***
  - e.g., disk-read completion
- the code that runs when an event occurs is called an event ***handler*** *(or callback)*
  - e.g., the code that computes the checksum
- handlers are ***registered*** to a specific event
- events are ***fired*** to trigger the execution of the handler

In Code

- request and registration of event handler are listed together in the code
- this code continues does not wait for event
- handler runs asynchronously when event occurs

```
void computeSum(char *buf, int blockNum, int numBytes) {
  int sum = 0;
  for(int i=0; i<numBytes; i++)
    sum += buf [i];
}

int sum(blockNum, numBytes) {
  char buf[numBytes];

  diskRead(buf, blockNum, numBytes, computeSum);
}
```

Asynchronous Execution

```
char buf[numBytes]
diskRead(buf, blockNum, numBytes, computeSum);
```

Registration of Handler

2

Event Handler

```
void computeSum(char* buf, int blockNum, int numBytes) {
  int sum = 0;
  for(int i=0; i<numBytes; i++)
    sum += buf[i];
}
```

Event Fired

Implementing Disk Read (Simplified)

- Interrupt Vector, Device ID, and PIO Address
  - initialized before all this starts
  - by operating system when device connects

```
#define MAX_DEVICES
void (*interruptVector [MAX_DEVICES])();

int  diskID      = 4;
int* diskAddress = (int*) 0x80001000;

interruptVector [diskID] = diskISR;
```

- Disk Read
  - register event handler
  - request block (PIO)

```
void diskRead(char *buf, int blockNum, int numBytes,
              void (*whenComplete)(char*, int, int))
{
  enqueue_handler(whenComplete, buf, blockNum, numBytes);
  // Perform PIO … more in a moment
}
```

- Interrupt Handler
  - find event handler and fire event
  - firing means calling the handler procedure

```
void diskISR() {
  struct handler_dsc {
    void (*handler)(char*, int, int);
    char* buf;
    int   blockNum;
    int   numBytes;
  };
  struct handler_dsc hd;
  dequeue_handler (&hd);
  hd.handler (hd.buf, hd.blockNum, hd.numBytes);
}
```

Disk Read PIO

- Expanded Disk Read with PIO
  - the message sent to disk has several fields
  - each field had different device-memory address
  - access it like a struct
    - but, keep in mind that writes are to device-memory;
    - they are messages across bus to device controller they are not writes to main memory

```
void *diskAddress = (void*) 0x80001000;

#DEFINE DISK_OP_READ  1
#DEFINE DISK_OP_WRITE 2

struct disk_ctl {
  int   op;
  char *buf;
  int   blockNum;
  int   numBytes;
}
```

```
void diskRead(char *buf, int blockNum, int numBytes, ...) {
  struct disk_ctl *dc = diskAddress;
  enqueue_handler (whenComplete, buf, blockNum, numBytes);
  dc->op       = DISK_OP_READ;
  dc->buf      = buf;
  dc->blockNum = blockNum;
  dc->numBytes = numBytes;
}
```

Did We Really Solve The Problem?

- We wanted to do this

```
int sumDiskData(blockNum, numBytes) {
  char buf[numBytes];
  int sum = 0;

  diskRead(buf, blockNum, numBytes);
  for (int i=0; i<numBytes; i++)
    sum += buf[i];

  return sum;
}
```

- But, reality forced us to do this

```
void computeSum(char* buf, int blockNum, int numBytes) {
  int sum = 0;
  for (int i=0; i<numBytes; i++)
    sum += buf[i];
}

void sumDiskData(blockNum, numBytes) {
  char buf[numBytes];

  diskRead(buf, blockNum, numBytes, computeSum);
}
```

- What's wrong?

Connecting Asynchrony to Program

- How do we use the value computed from the disk block
  - lets say we want to print it

```
void something() {
  ...
  int s = sumDiskData(buf, blk, n);
  printf("%d\n", s);
}
```

  - but asynchronously?

```
void computeSum(char* buf, int blockNum, int numBytes) {
  int sum = 0;
  for (int i=0; i<numBytes; i++)
    sum += buf [i];
  free(buf);
}

void sumDiskData (blockNum, numBytes) {
  char *buf = malloc(numBytes);

  diskRead (buf, blockNum, numBytes, computeSum);
}
```

Ordering in Asynchronous Programs

- If something has to happen after event
  - it must be triggered by the event
  - often this means it must be part of (or called by) event's handler
- To print the checksum

```
void computeSumAndPrint(char *buf, int blockNum, int numBytes) {
  int sum = 0;
  for (int i=0; i<numBytes; i++)
    sum += buf[i];
  printf("%d\n", sum);
  free(buf);
}

void sum(blockNum, numBytes, whenComplete) {
  char *buf = malloc(numBytes);

  diskRead(0, blockNum, numBytes, whenComplete);
}

sum (1234, 4096, computeSumAndPrint);
```

- Huge problem
  - often there's a ton of stuff that depend on returned data
  - that is, that must come after a particular event

Improving the Code … But making it worse

```
void computeSumAndPrint(char *buf, int blockNum, int numBytes) {
  int sum = 0;
  for (int i=0; i<numBytes; i++)
    sum += buf[i];
  printf("%d\n", sum);
  free(buf);
}
```

```
void printInt (int i) {
  printf ("%d\n", i);
}

void computeSumAndCallback (…, void (*sumCallback) (int)) {
  int sum = 0;
  for (int i=0); i<numBytes; i++)
    sum += buf [i];
  sumCallback (sum);
  free (buf);
}

void sum (blockNum, numBytes, whenComplete, sumCallback) {
  char* buf = malloc (numBytes);

  diskRead (buf, blockNum, numBytes, whenComplete, sumCallback);
}

sum (1234, 4096, computeSumAndCallback, printInt);
```

What if you want to do something after printInt?

Welcome to "callback hell" …

Happy System, Sad Programmer

- Humans like synchrony
  - we expect each step of a program to complete before the next one starts
  - we use the result of previous steps as input to subsequent steps
  - with disks, for example,
    - we read from a file in one step and then usually use the data we've read in the next step
- Computer systems are asynchronous
  - the disk controller takes 10-20 milliseconds (10^{-3}s) to read a block
    - CPU can execute 60 million instructions while waiting for the disk to complete one read
    - we must allow the CPU to do other work while waiting for I/O completion
  - many devices send unsolicited data at unpredictable times
    - e.g., incoming network packets, mouse clicks, keyboard-key presses
    - we must allow programs to be interrupted many, many times a second to handle these things
- Asynchrony makes programmers sad
  - it makes programs more difficult to write and much more difficult to debug

Possible Solutions

- Accept the inevitable
  - use an event-driven programming model
    - event triggering and handling are de-coupled
  - a common idiom in many Java programs
    - GUI programming follows this model
  - *CSP* (communicating sequential processes) boosts this idea to first-class status
    - no procedures or procedure calls
    - program code is decomposed into a set of sequential/synchronous processes
    - processes can fire events, which can cause other processes to run in parallel
    - each process has a guard predicate that lists events that will cause it to run
  - Javascript in web browsers and Node.js embrace asynchrony, albeit awkwardly
- Invent a new abstraction
  - an abstraction that provides programs the illusion of synchrony
  - but, what happens when
    - a program does something asynchronous, like disk read?
    - an unanticipated device event occurs?
- What's the right solution?
  - we still don't know — this is one of the most pressing questions we currently face