CPSC 213
Introduction to Computer Systems

Unit 2a
I/O Devices, Interrupts, and DMA

All slides adapted from materials by Mike Feeley, Jonatan Schroeder, Robert Xiao, and Jordon Johnson

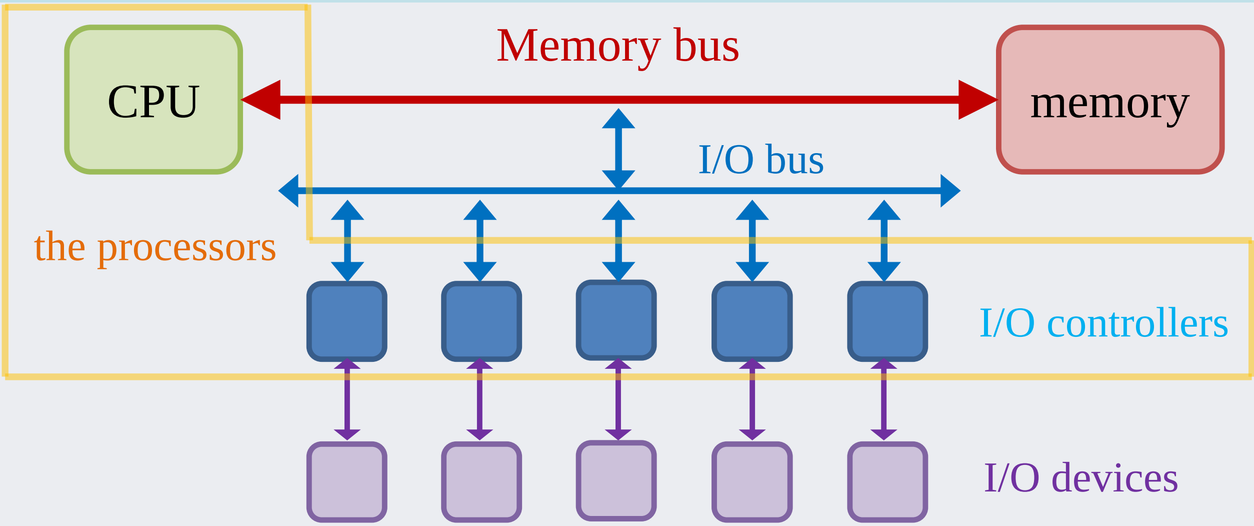
Announcements

- Google doc for lecture questions
  - See Piazza for link, section 102
  - https://docs.google.com/document/d/1G6hkekQS7mT9lFpP8AVftYao8vLRujIrRLAvOuX_07w/edit
  - Add your question anonymously (at the top)
  - Help answer questions too!

Overview

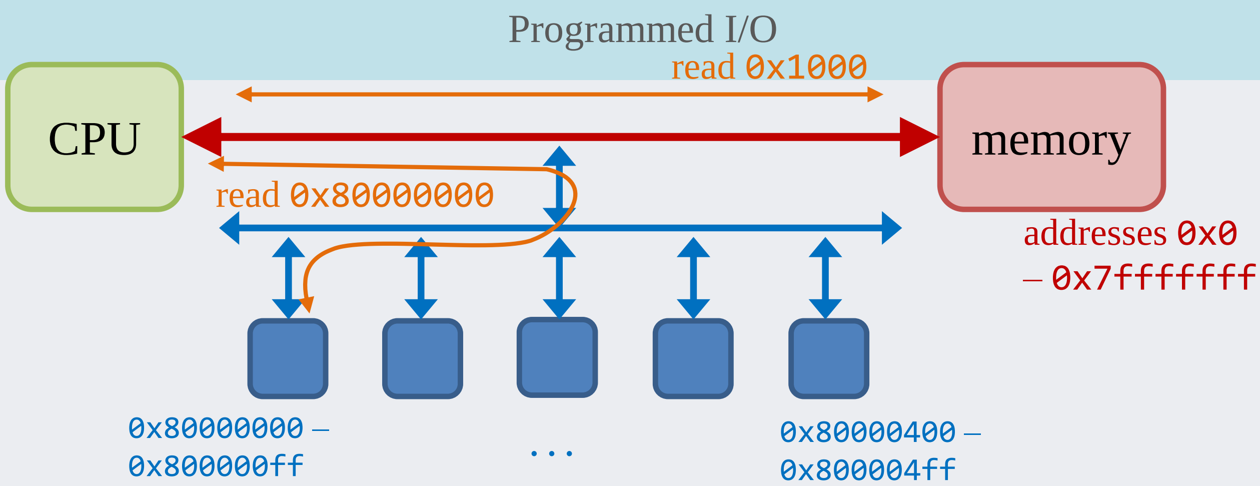
- Reading in Text
  - 8.1, 8.2.1, 8.5.1-8.5.3
- Learning Goals
  - Explain what PIO is, why it exists, what processor originates it, and how it is used
  - Explain what DMA is, why it exists, what it does, and what processor originates it
  - Explain what an interrupt is, why it exists, what it does, and what processor originates it
  - Compare PIO and DMA by identifying when each *can* be used and when each *should* be used
  - Compare PIO and interrupts by identify 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

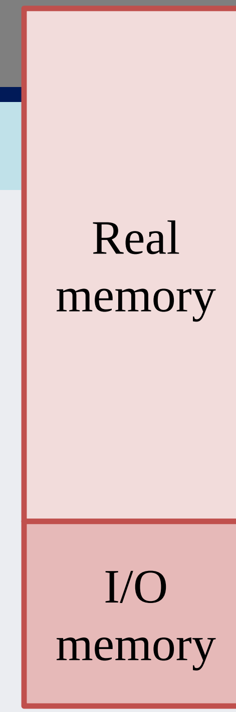
Looking beyond the CPU and memory



- 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** (unit of data) at a time between CPU and I/O controller
  - typically use standard load/store instructions but to I/O-mapped memory address
- I/O-mapped memory
  - memory addresses beyond the end 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 the device into r1
```

Limitations of PIO

- Reading or writing large amounts of data slows CPU
  - CPU must transfer one word at a time
  - controller/device is much slower than CPU
  - so, CPU runs at controller/device speed, mostly waiting for controller
- I/O controller cannot initiate communication
  - sometimes the CPU asks for data
  - but, sometimes the controller receives data for the CPU, without CPU asking
    - e.g. mouse click or network packet reception
  - how does controller notify CPU that it has data the CPU should want?
- One not-so-good idea:
  - what is it? Polling
  - what are the drawbacks? still done using PIO
  - when is it OK?

Polling and I/O memory

CPU

ready	80000100
value	80000104

I/O controller !

80000100	~~0~~
80000104	7

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

  while (*ready == 0) {};

  return *value;
}
```

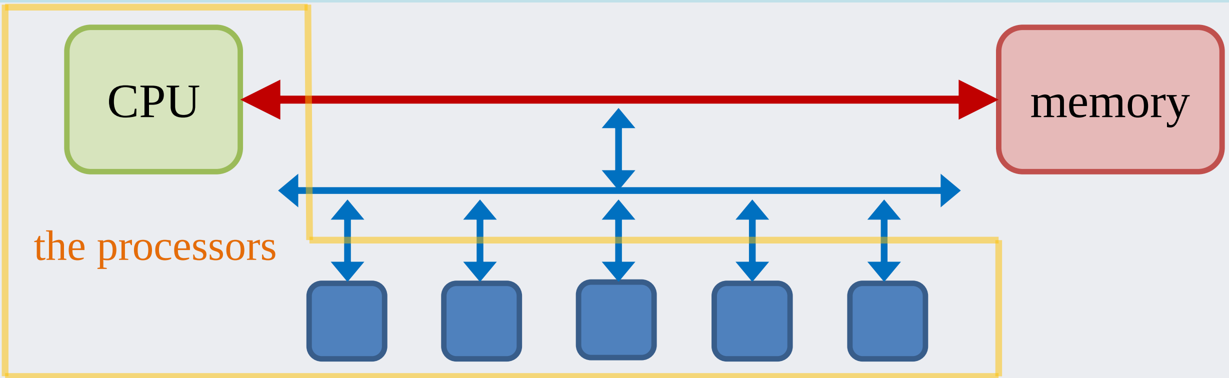
```
void send(int v) {
  volatile int* ready = 0x80000100;
  volatile int* value = 0x80000104;

  *value = v;
  *ready = 1;
}

send(7);
```

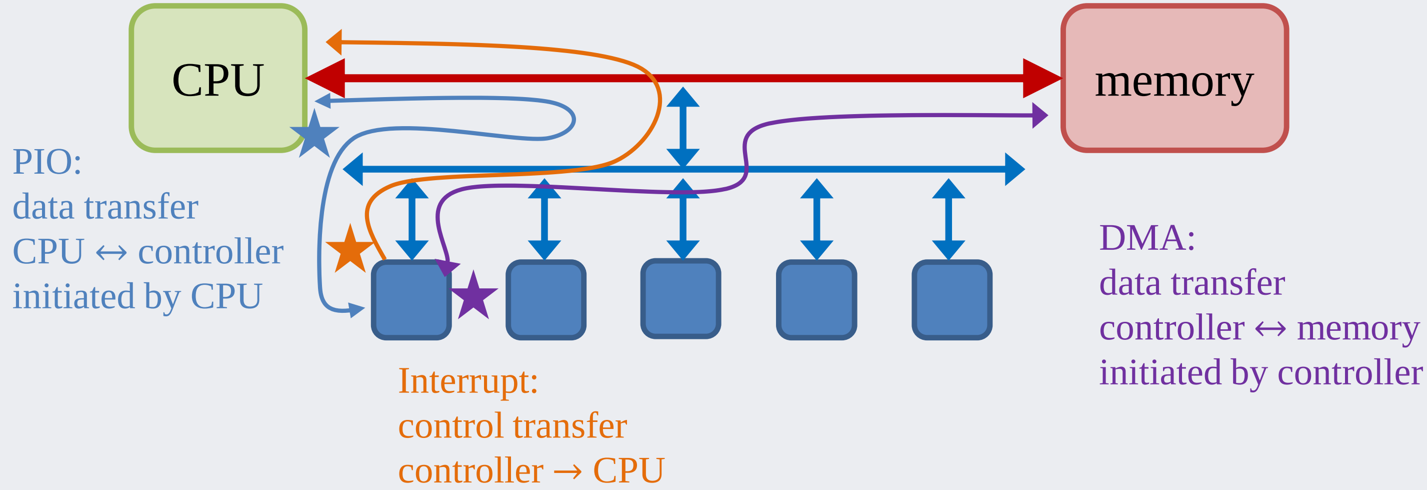
- Reading (or writing) I/O memory is SLOW
  - CPU -> I/O device: "give me value at address x"
  - I/O device -> CPU: "OK, here is the value"
- Polling is OK if poll has low overhead, or has high probability of "hit"

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 SM213

- 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 at 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();
}
```

'jump table

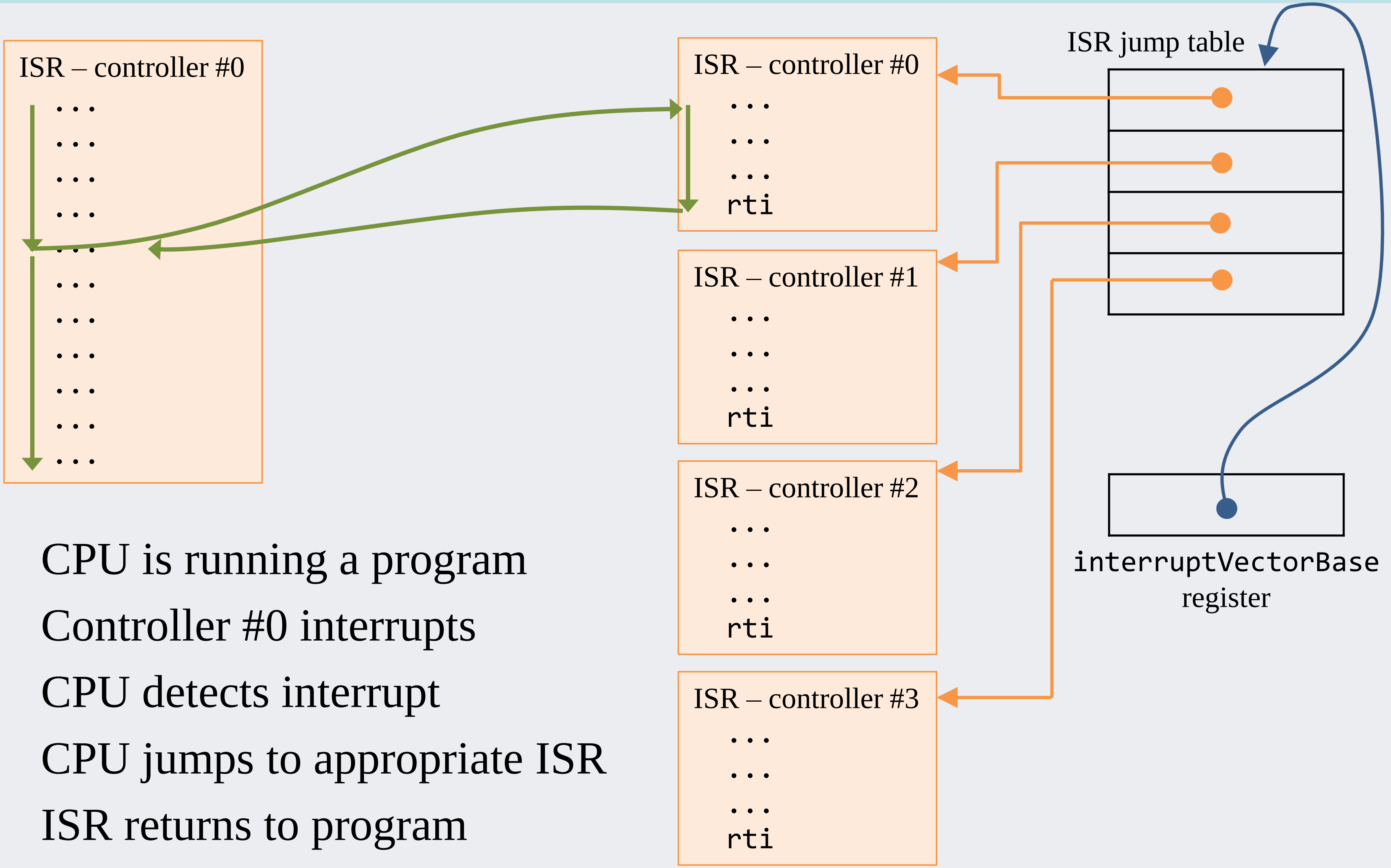
index

"polls" on CPU register instead of I/O memory

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 appropriate ISR
5. ISR returns to program

Architectural view of interrupts

CPU

isDeviceInterrupting	~~0~~ 1
interruptControllerID	7

I/O controller #3

myID 3

I/O controller #7

myID 7

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

Programming with I/O

Disk read timeline

CPU

1. PIO to request read

...
~~idle waiting~~
...

...
do other things
...

6. Interrupt received
   Call `readComplete`

I/O controller

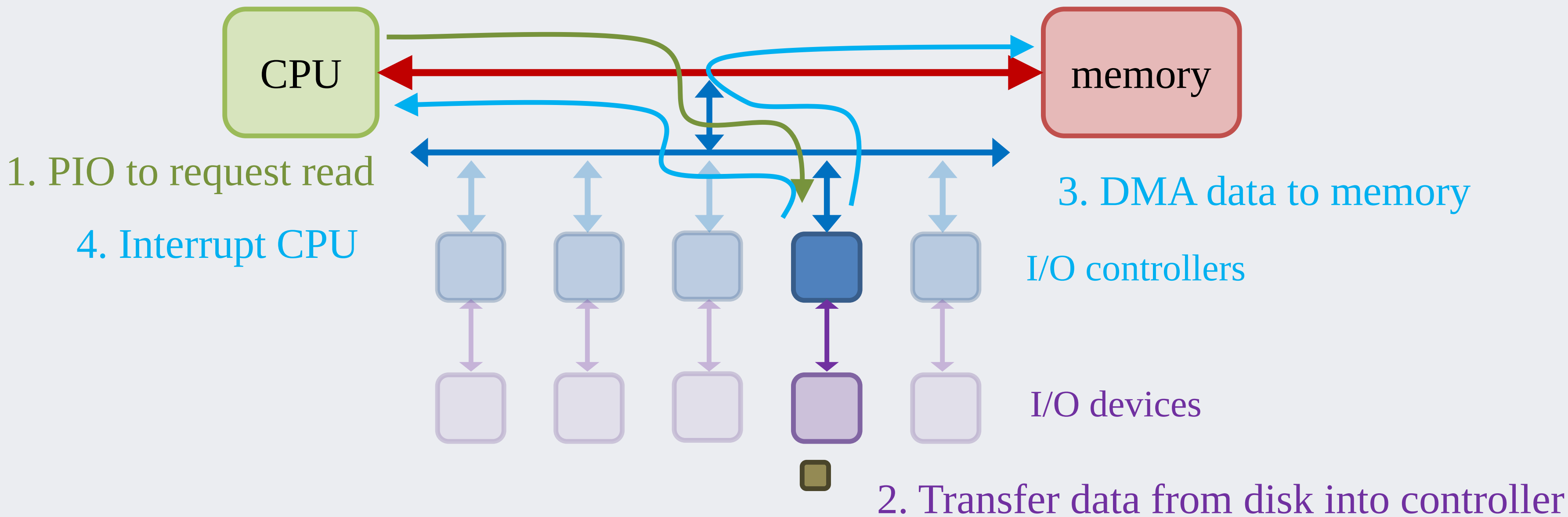
2. PIO received, start read

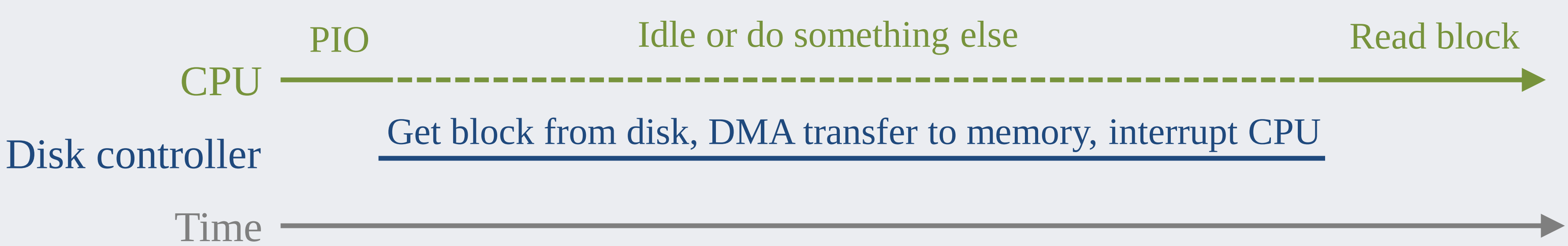
...
wait for read to complete
...

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 request read or write to a block

Disk read timeline





Sequential execution…

- Consider a program that reads data from a disk
  - you can think of the read as 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 executing in sequence

- Example:

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

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

  return sum;
}
```

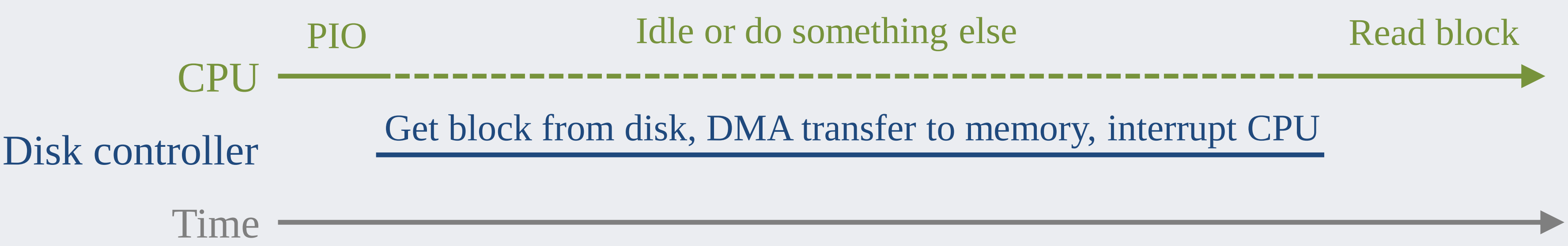
iClicker 2a.1

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

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

  return sum;
}
```

Place these events on the timeline below:
X: request `diskRead`
Y: requested data is in `buf`
Z: `for (int…)`



What is the actual order?

A: XYZ B: XZY C: Either XYZ or XZY, you can't tell D: Some other order E: 🤷‍♀️

Making code asynchronous

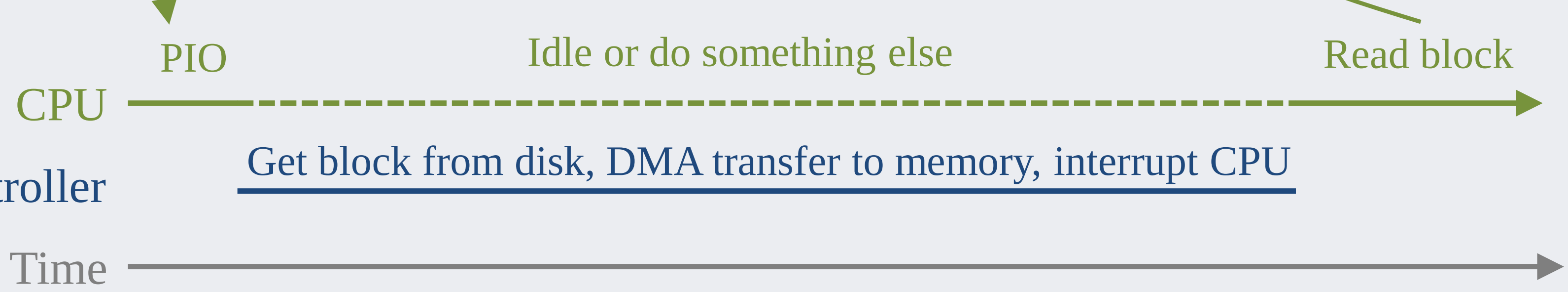
```
char buf[numBytes];
diskRead (buf, blockNum, numBytes);
```

This code requests something

The desired execution has something happening between these two pieces of code

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

This code should run when the request is complete



Writing asynchronous code in C

- Events and event handlers
  - things that cause 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. 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 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 sumDiskBlock (blockNum, numBytes) {
  char buf [numBytes];

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

Making code asynchronous

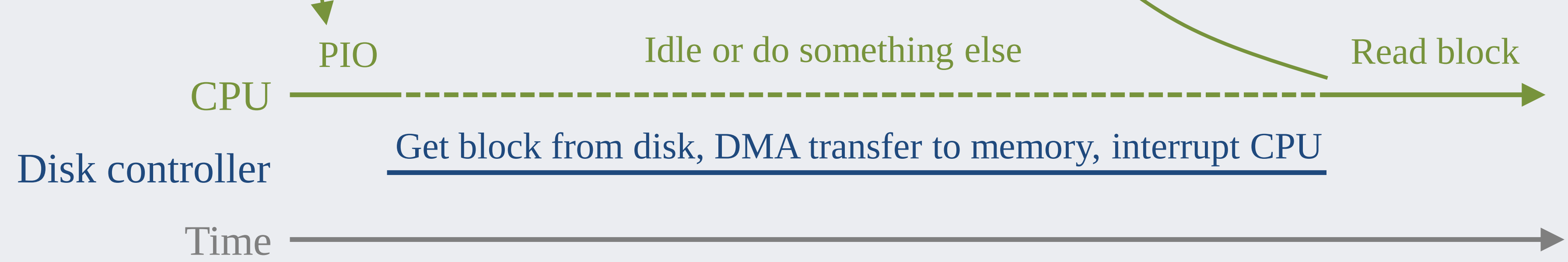
```
char buf[numBytes];
diskRead (buf, blockNum, numBytes, computeSum);
```

Registration of handler

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



Disk read PIO

- Expanded disk read with PIO
  - the message sent to disk has several fields
  - each field has 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;
}
```

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 about this later
}
```

- 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);
}
```

Did we really solve the problem?

iClicker 2a.2

- We wanted to do this:

```
int sumDiskBlock (int blockNum, int 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:

Which line is wrong?

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

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

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

What's the problem?

Why can't computeSum return a value?

iClicker 2a.3

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

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

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

What procedure calls `computeSum`?

A. `sumDiskBlock`
B. the procedure that calls `sumDiskBlock`
C. another procedure in this program
D. the disk interrupt service routine
E. something else / I don't know

Connecting asynchrony to program

- How do we use the value computed from the disk block?
  - suppose we want to print it

```
void something () {
  ...
  int s = sumDiskBlock (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 sumDiskBlock (blockNum, numBytes) {
  char* buf = malloc (numBytes);
  diskRead (buf, blockNum, numBytes, computeSum);
}
```

Ordering in asynchronous programs

- If something has to happen after an 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 sumDiskBlock (blockNum, numBytes, whenComplete) {
  char* buf = malloc (numBytes);
  diskRead (buf, blockNum, numBytes, whenComplete);
}

void something() {
  sumDiskBlock (1234, 4096, computeSumAndPrint);
}
```

Huge problem:

- often there is a ton of stuff that depend on the returned data
- these must happen after a particular event

Improving the code…

```
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);
}
```

too specific, make this a parameter (callback)

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

…but making it worse

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

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

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

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

void something() {
  sumDiskBlock(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* is a language 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