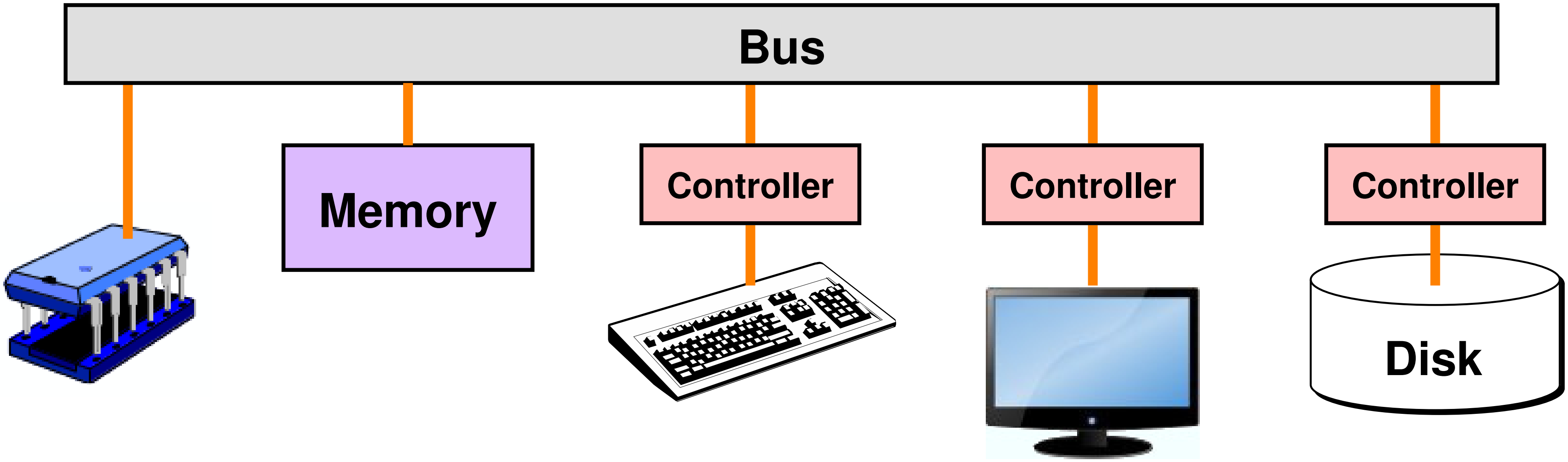
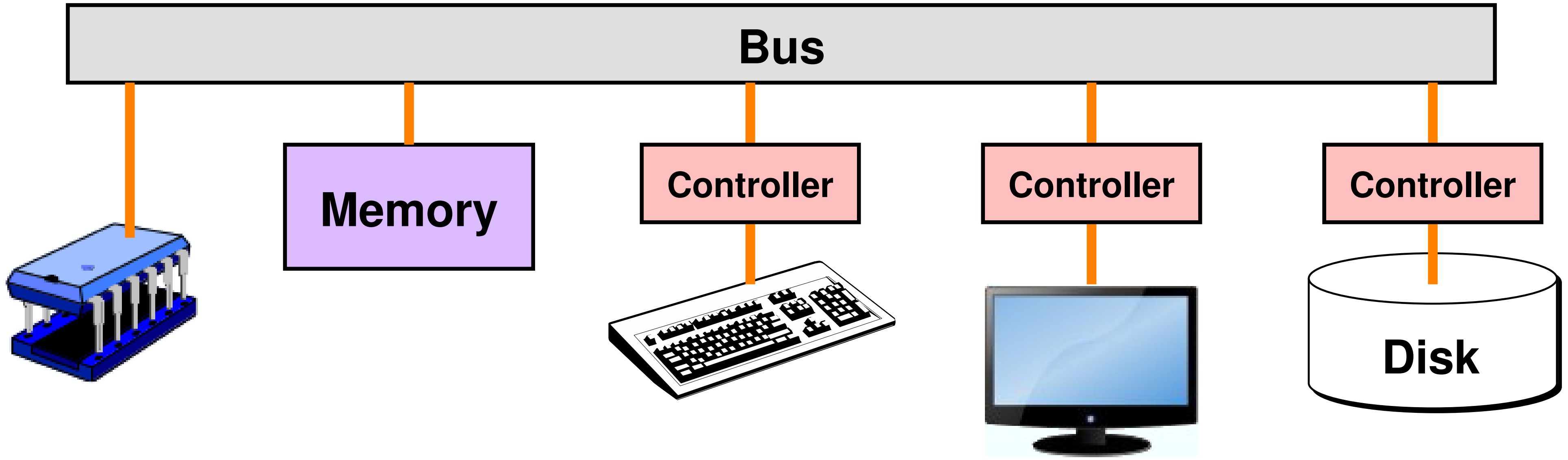
Simple I/O Architecture



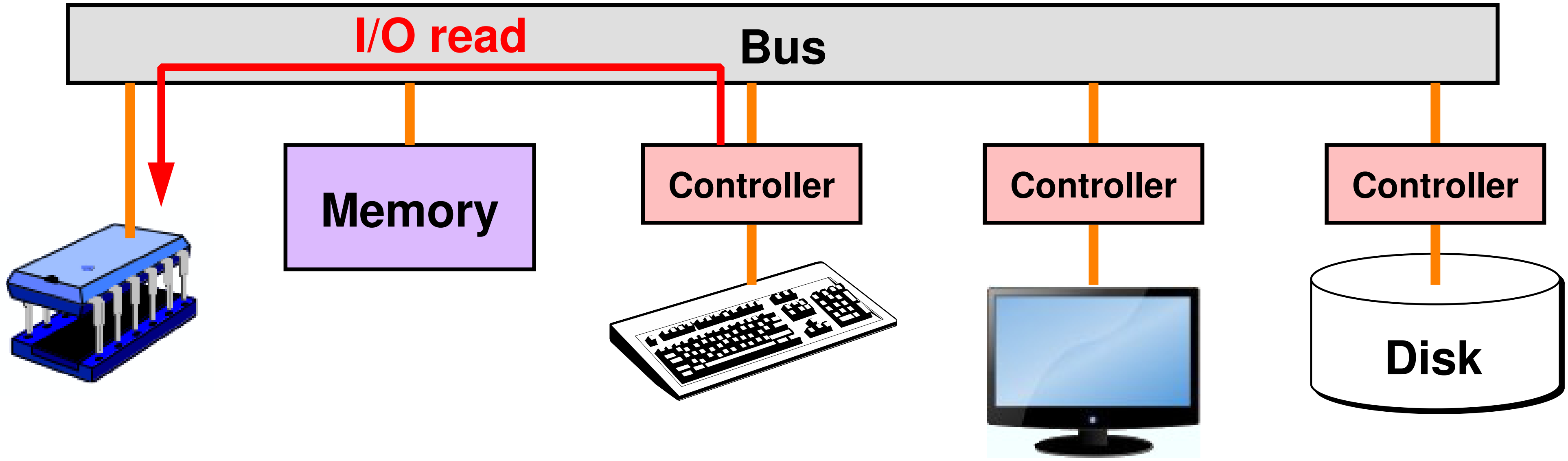
- memory-mapped I/O
  - all controllers listen on the bus to determine if a request is for itself or not
  - I/O controllers "process" the bus request
    - and respond to relatively few addresses
    - if no one responds, you get a "bus error"
  - memory is not really a "device"

Simple I/O Architecture



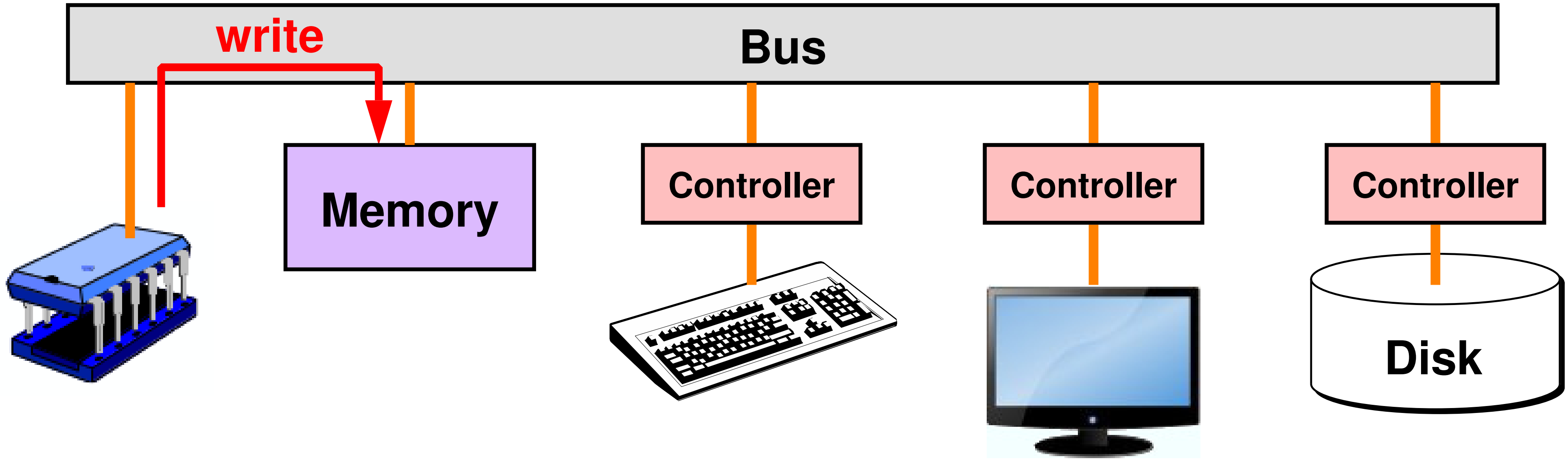
- memory-mapped I/O
- two categories of devices
  - PIO (programmed I/O)
    - perform I/O operations by reading or writing data in the controller registers one byte or word at a time over the bus

Simple I/O Architecture



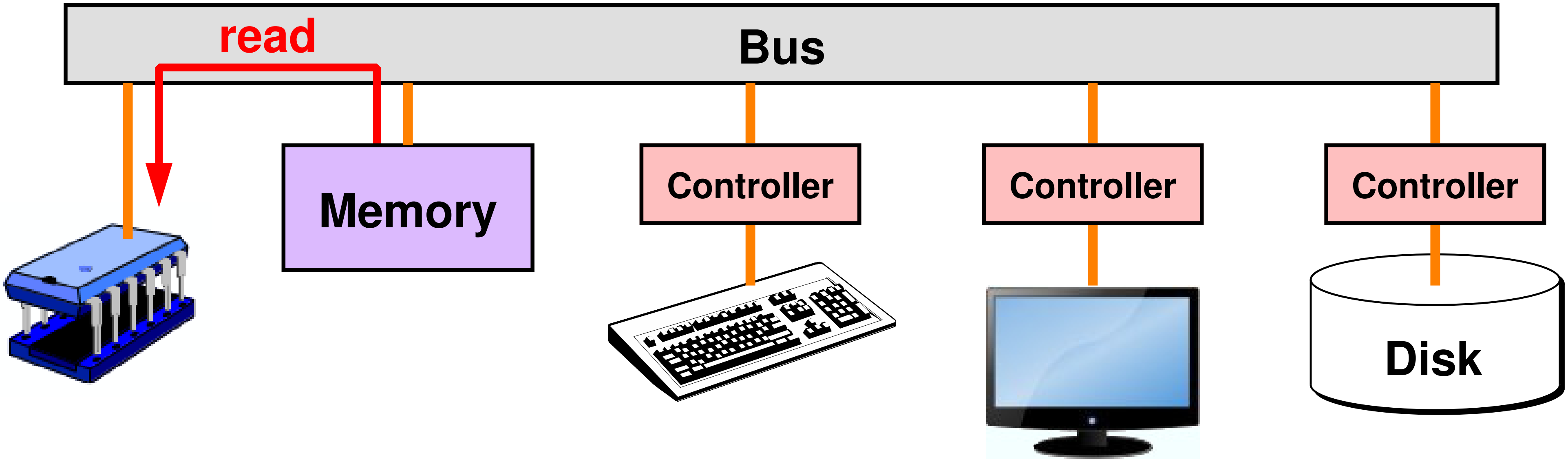
- memory-mapped I/O
- two categories of devices
  - PIO (programmed I/O)
    - perform I/O operations by reading or writing data in the controller registers one byte or word at a time over the bus

Simple I/O Architecture



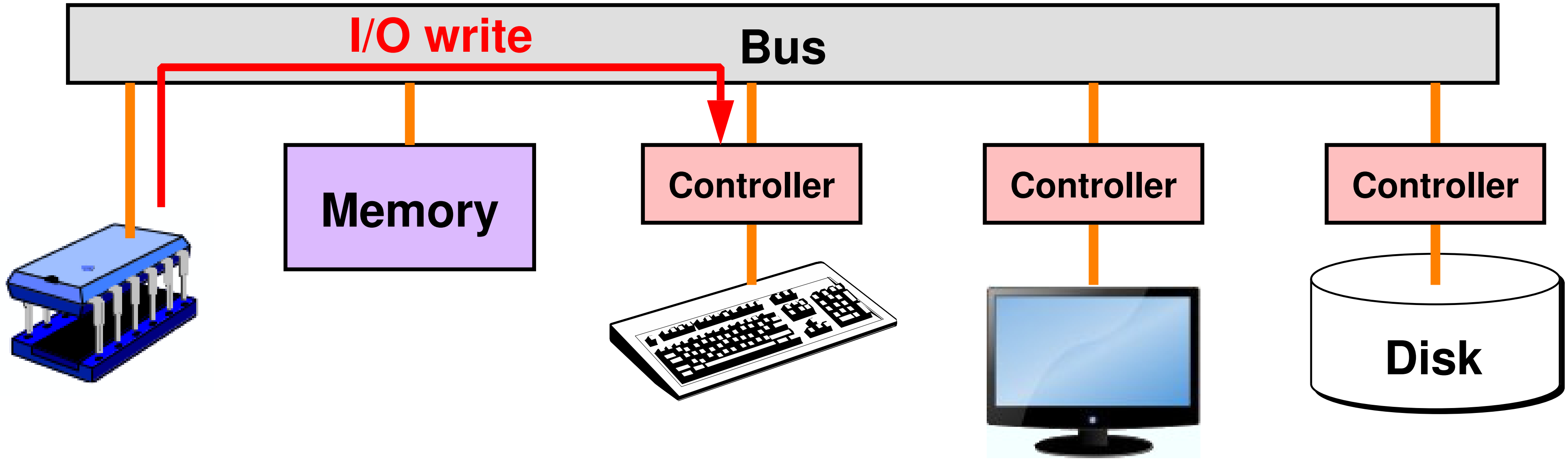
- memory-mapped I/O
- two categories of devices
  - PIO (programmed I/O)
    - perform I/O operations by reading or writing data in the controller registers one byte or word at a time over the bus

Simple I/O Architecture



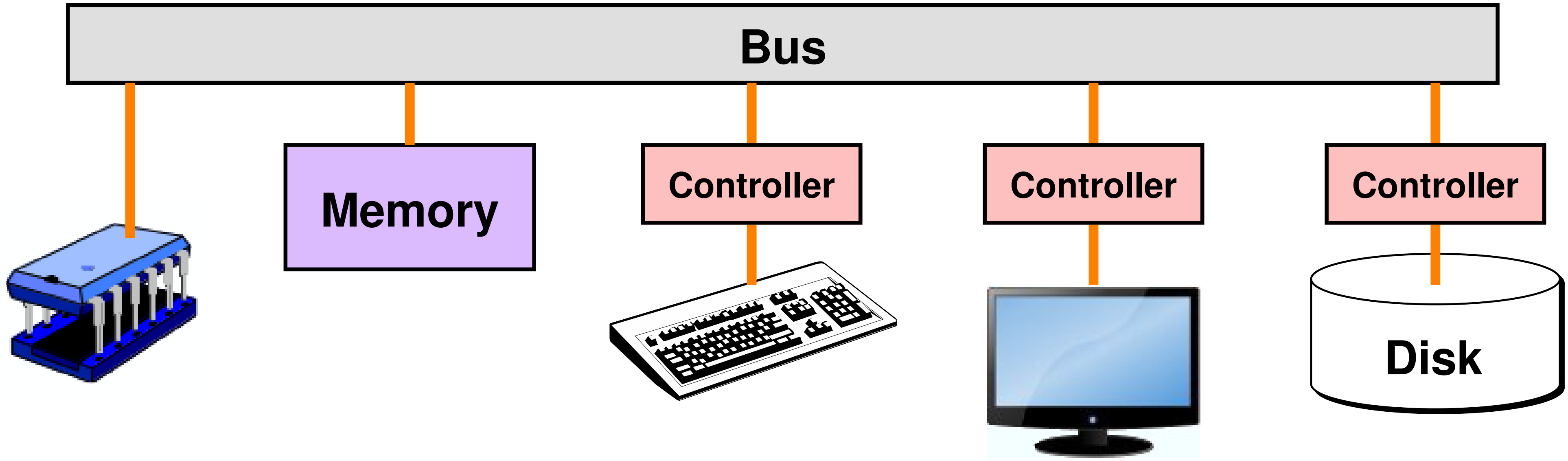
- memory-mapped I/O
- two categories of devices
  - PIO (programmed I/O)
    - perform I/O operations by reading or writing data in the controller registers one byte or word at a time over the bus

Simple I/O Architecture



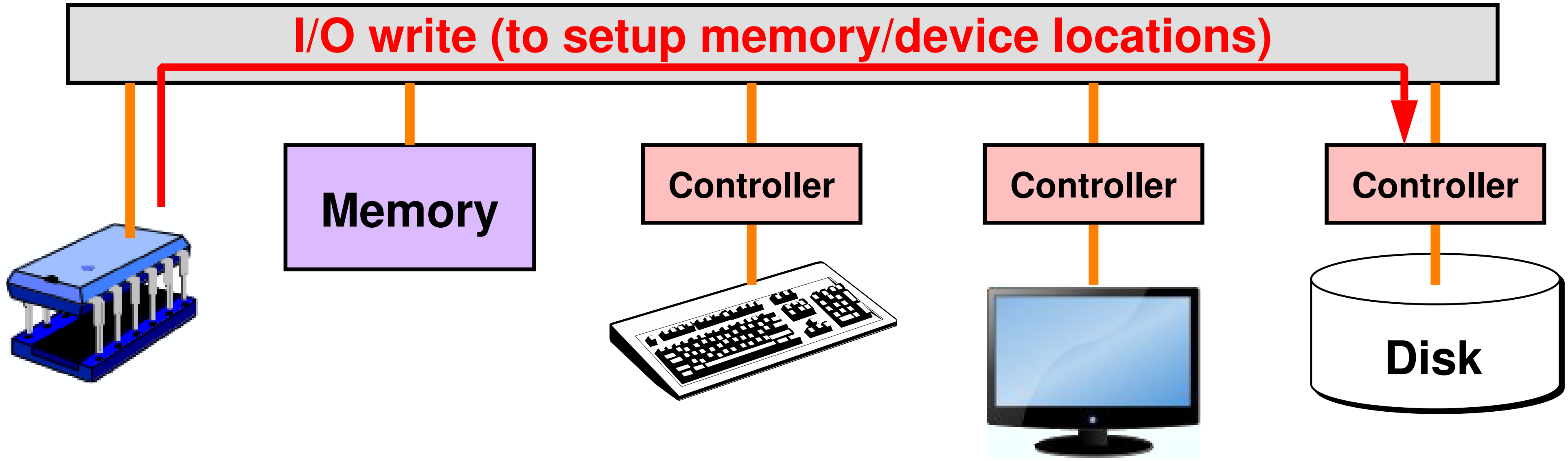
- memory-mapped I/O
- two categories of devices
  - PIO (programmed I/O)
    - perform I/O operations by reading or writing data in the controller registers one byte or word at a time over the bus

Simple I/O Architecture



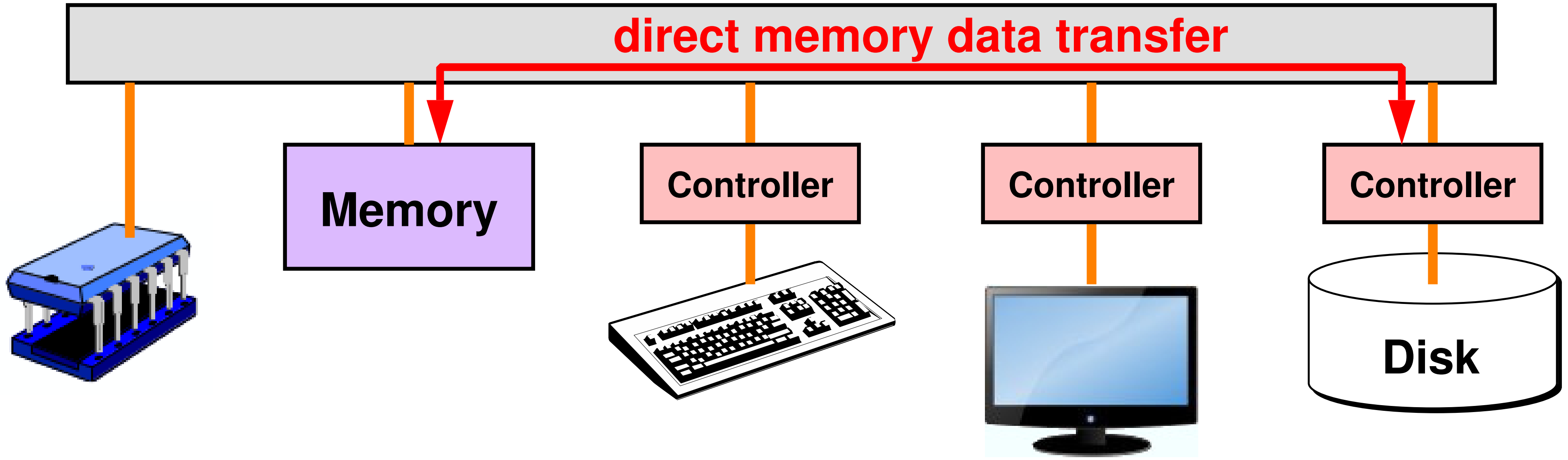
- memory-mapped I/O
- two categories of devices
  - PIO (programmed I/O)
  - DMA (direct memory access)
    - the controller performs the I/O itself
    - the processor writes to the controller to tell it where to transfer the results to
    - the controller takes over and transfers data between itself and primary memory

Simple I/O Architecture



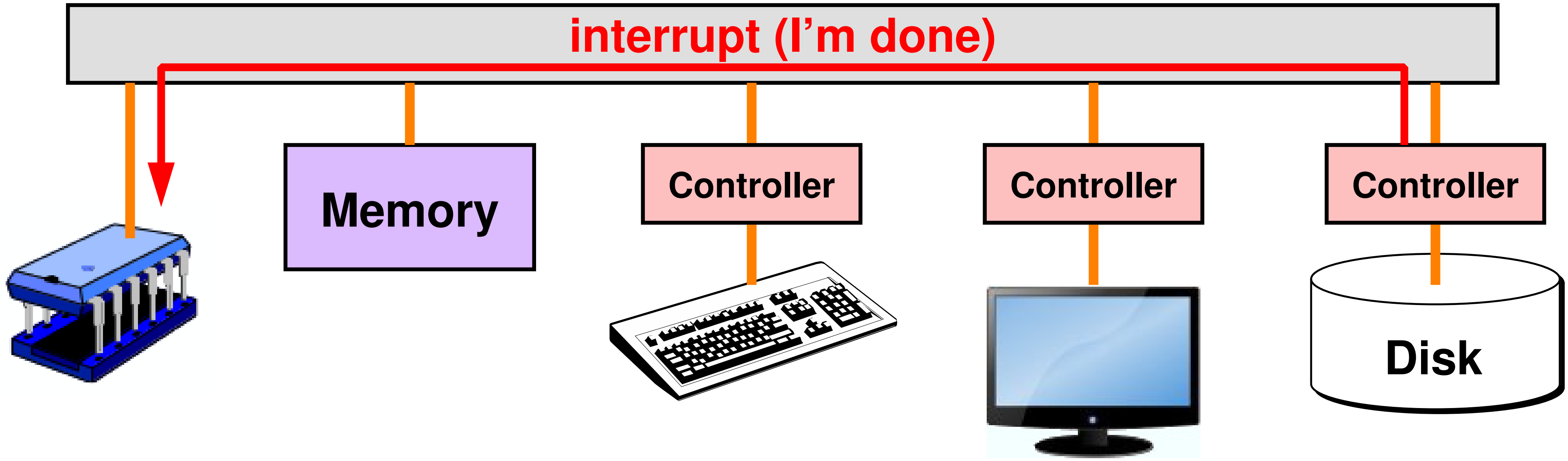
- memory-mapped I/O
- two categories of devices
  - PIO (programmed I/O)
  - DMA (direct memory access)
    - the controller performs the I/O itself
    - the processor writes to the controller to tell it where to transfer the results to
    - the controller takes over and transfers data between itself and primary memory

Simple I/O Architecture



- memory-mapped I/O
- two categories of devices
  - PIO (programmed I/O)
  - DMA (direct memory access)
    - the controller performs the I/O itself
    - the processor writes to the controller to tell it where to transfer the results to
    - the controller takes over and transfers data between itself and primary memory

Simple I/O Architecture



- memory-mapped I/O
- two categories of devices
  - PIO (programmed I/O)
  - DMA (direct memory access)
    - the controller performs the I/O itself
    - the processor writes to the controller to tell it where to transfer the results to
    - the controller takes over and transfers data between itself and primary memory

PIO Registers

- This is the abstraction of a PIO device
  - a *"register"* is just a *memory-mapped I/O address* on the bus

GoR	GoW	IER	IEW					Control register (1 byte)
RdyR	RdyW							Status register (1 byte)
								Read register (1 byte)
								Write register (1 byte)

Legend:

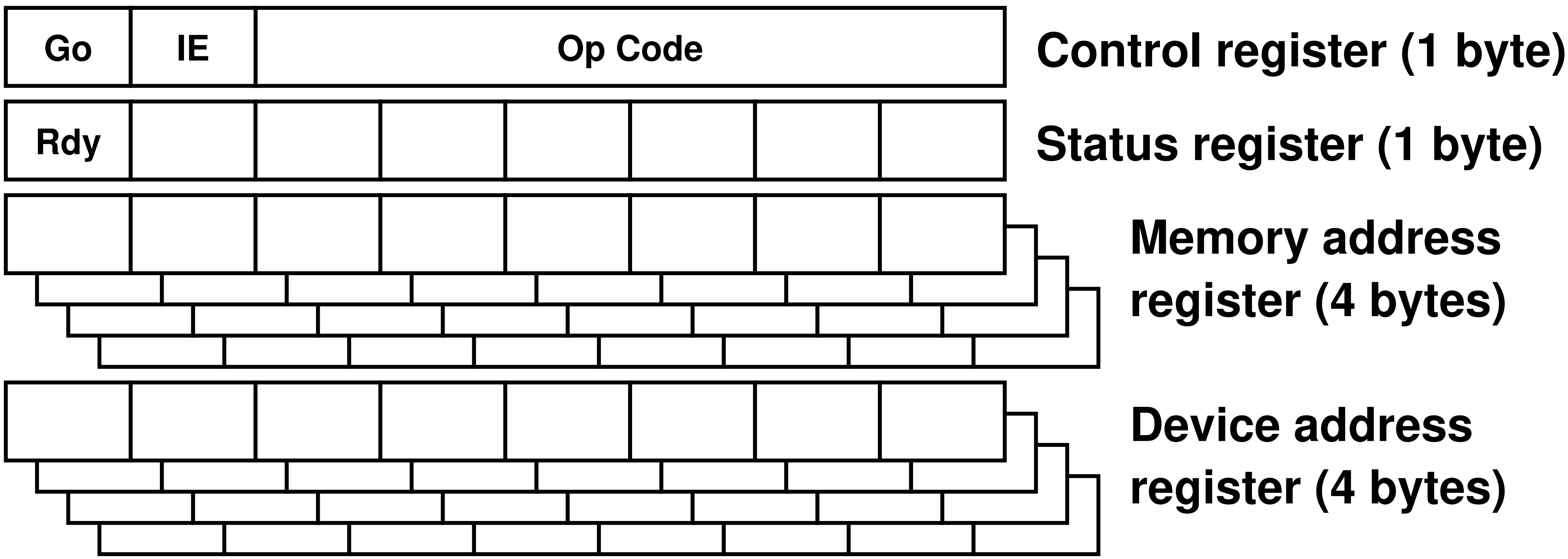
Legend:	GoR	Go read (start a read operation)
	GoW	Go write (start a write operation)
	IER	Enable read-completion interrupts
	IEW	Enable write-completion interrupts
	RdyR	Ready to read
	RdyW	Ready to write

Programmed I/O

- E.g.: Terminal controller
- Procedure (write)
  - write a byte into the write register
  - set the GoW bit (and optionally the IEW bit if you'd like to be notified via an interrupt) in the control register
  - poll and wait for RdyW bit (in status register) to be set (if interrupts have been enabled, an interrupt occurs when this happens)

DMA Registers

- This is the abstraction of a DMA device
  - a *"register"* is just a *memory-mapped I/O address* on the bus



Legend:		
	Go	Start an operation
	Op Code	Operation code (identifies the operation)
	IE	Enable interrupts
	Rdy	Controller is ready

Direct Memory Access

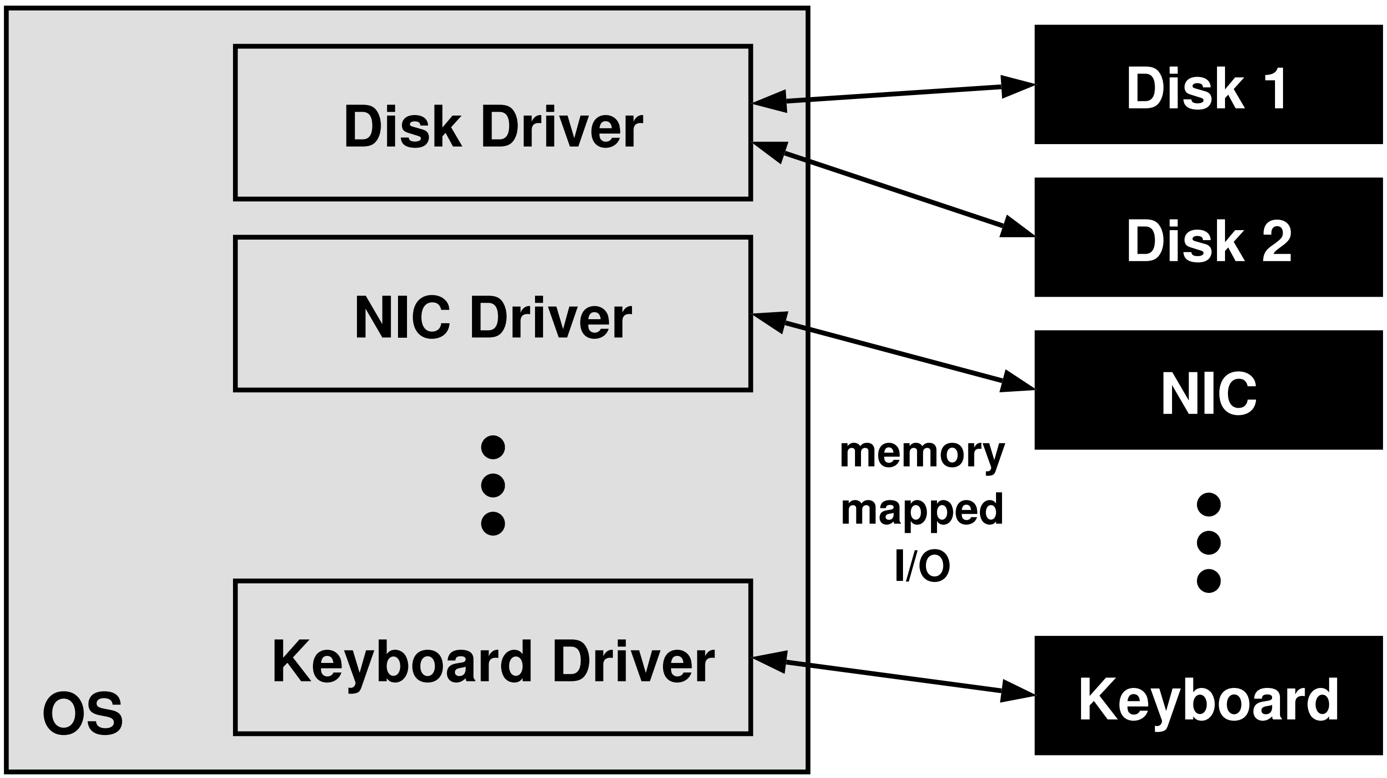
- E.g.: Disk controller
- Procedure
  - set the disk address in the device address register (only relevant for a seek request)
  - set the buffer address in the memory address register
  - set the op code (SEEK, READ or WRITE), the Go bit and, if desired, the IE bit in the control register
  - wait for interrupt or for Rdy bit to be set

Device Drivers

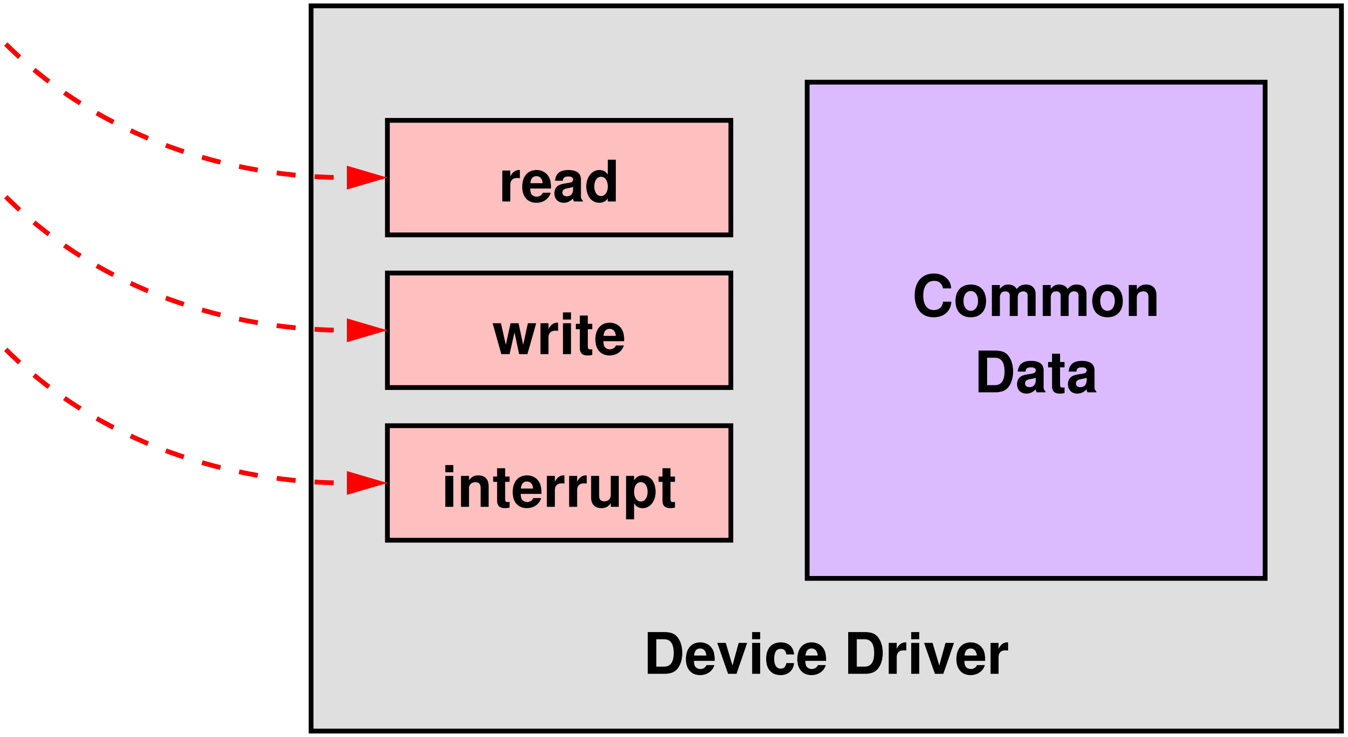
- Who knows how to use memory-mapped I/O to talk to devices?
  - *not* the kernel developers
  - device manufacturers do
  - it's desirable if kernel is *device-independent*
    - but how?

Device Drivers

- Who knows how to use memory-mapped I/O to talk to devices?
  - *not* the kernel developers
  - device manufacturers do
  - it's desirable if kernel is *device-independent*
    - but how?
- Device manufacturers package their knowledge into *device drivers*



Device Drivers



- *Device drivers* provide a *standard interface* to the rest of the OS
  - *code in device drivers* knows how to talk to devices (the rest of the OS really doesn't know the details)
  - OS can treat I/O in a *device-indepdendent* manner by calling functions in the standard interface
    - "interface" = *array of function pointers*

C++ Interface = Array of Function Pointers

```
class disk {
  public:
    virtual status_t read(request_t) = 0;
    virtual status_t write(request_t) = 0;
    virtual status_t interrupt( ) = 0;
};
```

- C++ *polymorphism* achieved using *virtual base class* or *interface*
  - each type of disk driver is a *subclass* of the disk class and has its own implementation of these functions
    - each disk driver looks like a generic disk to the OS
  - this gets compiled into an array of function pointers (which is what C++ code gets compiled into)
    - in reality, there are no object classes and no polymorphism
      - the CPU doesn't even know about data structures
      - the CPU only knows about memory addresses and how to execute machine instructions

C Impmlementation of C++ Polymorphism

```
struct disk {
  void **disk_ops;
  ...
}
```

```
(void *)wd123_disk_ops[] = {
  read_handler_t wd123_read;
  write_handler_t wd123_write;
  intr_handler_t wd123_intr;
  ...
}

(void *)sg76_disk_ops[] = {
  read_handler_t sg76_read;
  write_handler_t sg76_write;
  intr_handler_t sg76_intr;
  ...
}
```

```
struct disk *d = ...;
d->disk_ops = wd123_disk_ops; /* known as "binding" */
/* or d->disk_ops = sg76_disk_ops; */
```

- to read from any disk, call the first function indirectly

```
(*((read_handler_t)d->disk_ops[0]))(...);
```

... in C++

```
class disk {
  public:
    virtual status_t read(request_t) = 0;
    virtual status_t write(request_t) = 0;
    virtual status_t interrupt( ) = 0;
};
```

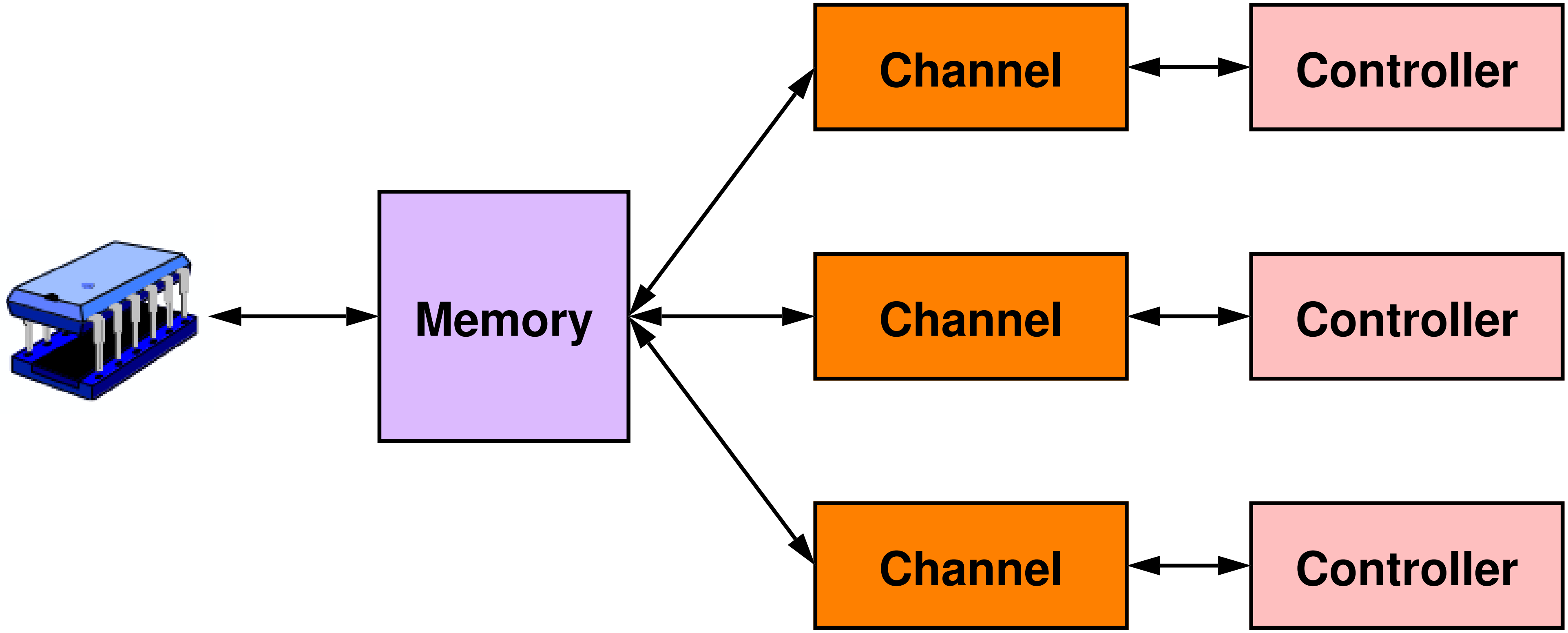
- This is a synchronous interface
  - a user thread would call `read/write()` system call
  - these functions are called in the kernel which starts the device
  - the device driver's interrupt method is called in the interrupt context
    - if I/O is completed, the thread is unblocked and return from the `read/write()` system call

A Bit More Realistic

```
class disk {
  public:
    virtual handle_t start_read(request_t) = 0;
    virtual handle_t start_write(request_t) = 0;
    virtual status_t wait(handle_t) = 0;
    virtual status_t interrupt( ) = 0;
};
```

- Even in Sixth-Edition Unix, the internal driver interface is often *asynchronous*
  - `start_read/start_write()` returns a handle identifying the operation that has started
  - a thread can call the `wait()` method to *synchronously* wait for *I/O completion*

I/O Processors: Channels



- when I/O costs dominate computation costs
  - use I/O processors (a.k.a. channels) to handle much of the I/O work
  - important in large data-processing applications
- can even download program into a channel processor

3.3 Dynamic Storage Allocation

- *Best-fit & First-fit Algorithms*
- Buddy System
- Slab Allocation

Dynamic Storage Allocation

- Where in the kernel do you need to do memory allocation?
  - stack space
  - `malloc()`
  - `fork()`
  - various OS data structures
    - process control block
    - thread control block
    - mutex (it's a queue)
  - etc.
- Memory allocator
  - computer science people like to personify things
    - just like we say that a variable "lives" in an address space
  - a "memory allocator" is simply a collection of functions
    - it's not a thread
    - it's not a process

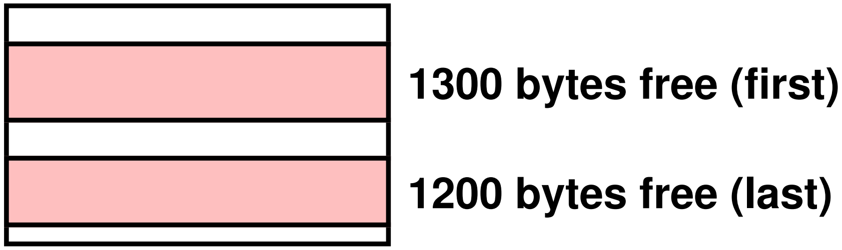
Dynamic Storage Allocation

- Goal: allow dynamic creation and destruction of data structures
  - use "memory allocator" to manage a large block of memory (i.e., a collection of contiguous memory addresses)
    - *contract* with application:
      - application will not touch anything owned by the memory allocator (memory allocator can do anything with them)
      - application will only touch "allocated memory blocks" (memory allocator promises not to touch these)
    - very bad things can happen if contract is violated
      - assuming that there are no bugs in a memory allocator

- Concerns:
  - efficient use of storage
  - efficient use of processor time

- Example:
  - *first-fit* vs. *best-fit* allocation

Allocation Example

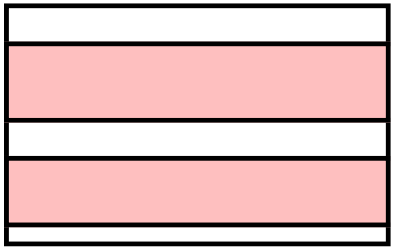


Allocate 1000 bytes: **First Fit** **Best Fit**

Allocate 1100 bytes:

Allocate 250 bytes:

Allocation Example

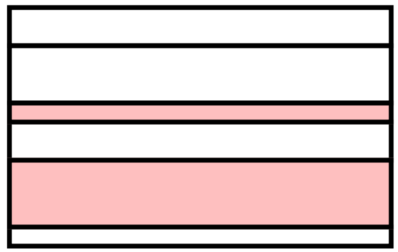


1300 bytes free (first)

1200 bytes free (last)

Allocate 1000 bytes:

First Fit



300

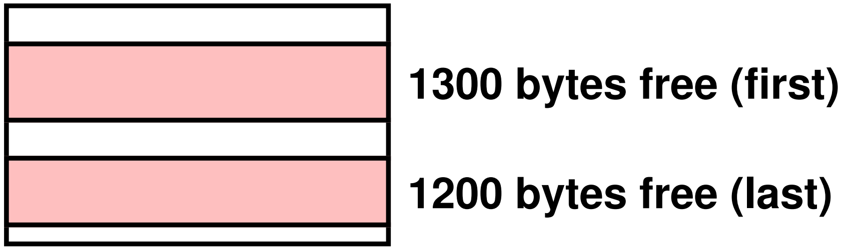
1200

Best Fit

Allocate 1100 bytes:

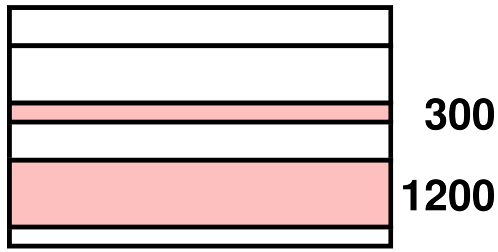
Allocate 250 bytes:

Allocation Example

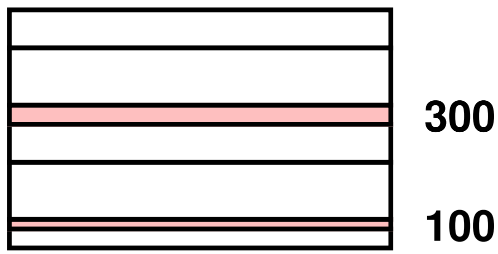


Allocate 1000 bytes:

First Fit | **Best Fit**

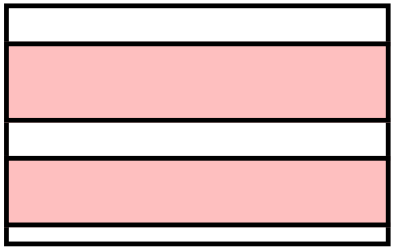


Allocate 1100 bytes:



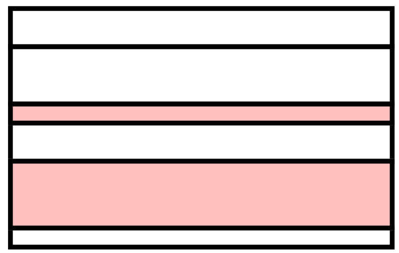
Allocate 250 bytes:

Allocation Example

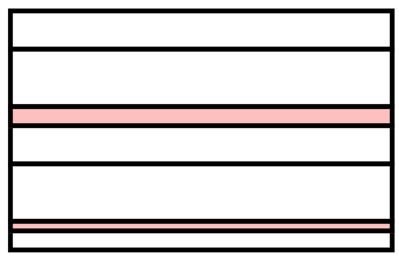


Allocate 1000 bytes:

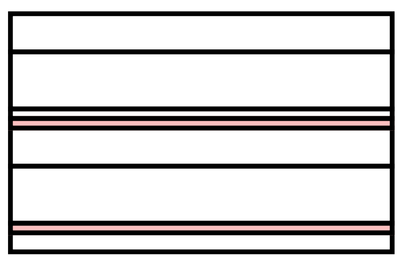
First Fit **Best Fit**



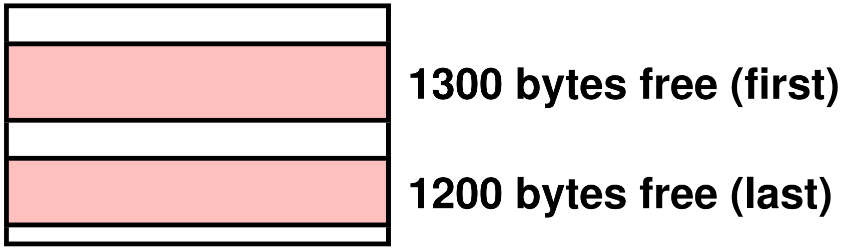
Allocate 1100 bytes:



Allocate 250 bytes:

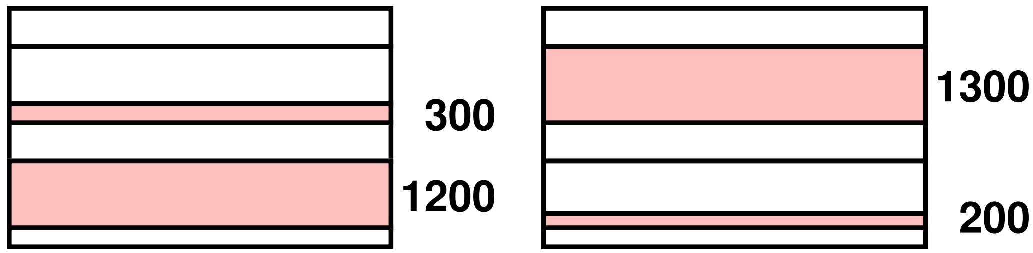


Allocation Example



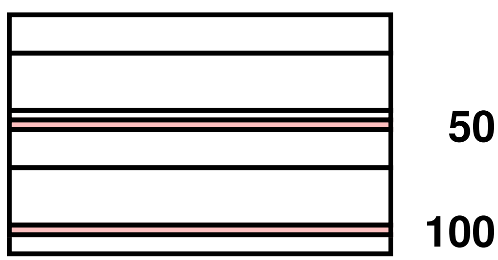
Allocate 1000 bytes:

First Fit | **Best Fit**

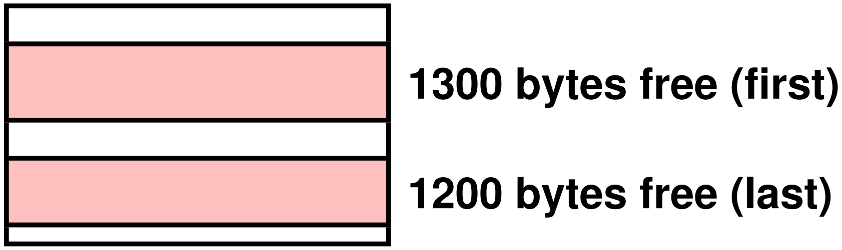


Allocate 1100 bytes:

Allocate 250 bytes:



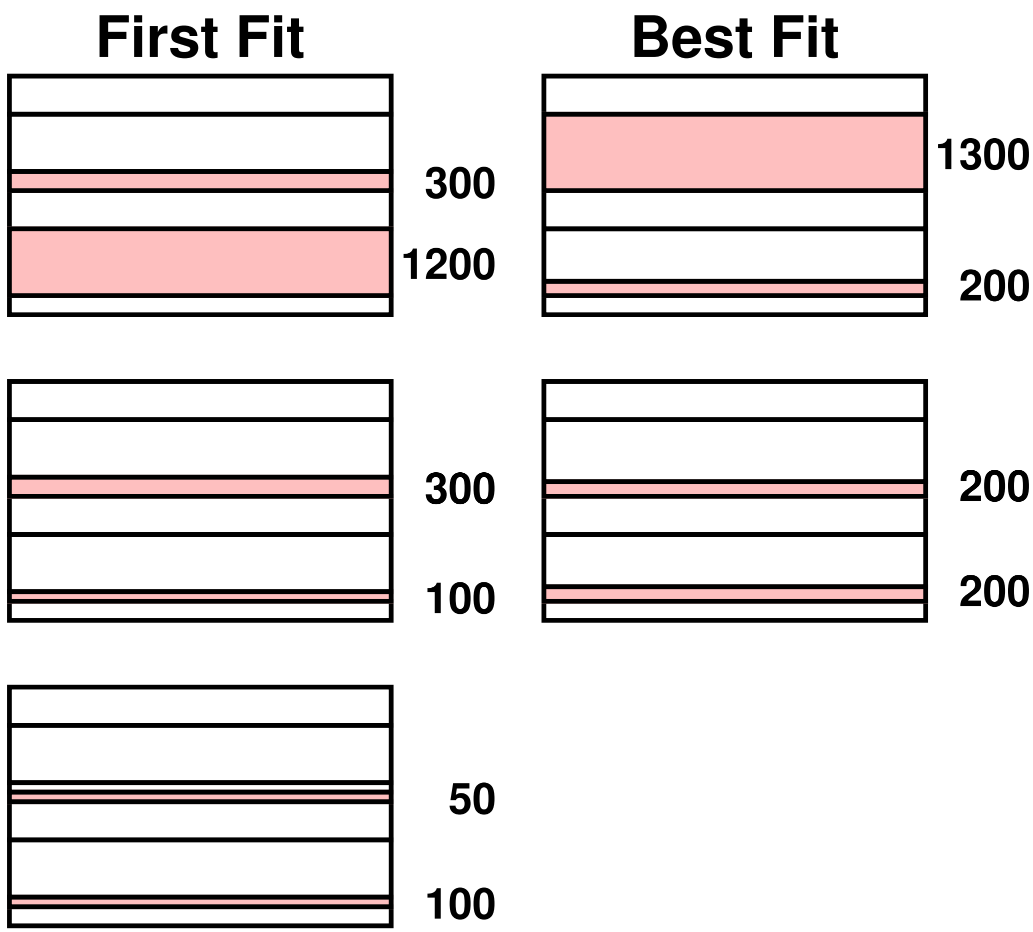
Allocation Example



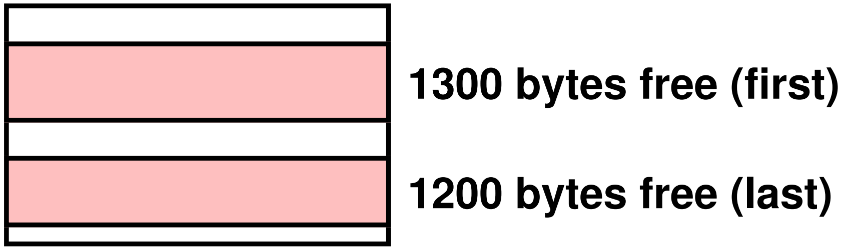
Allocate 1000 bytes:

Allocate 1100 bytes:

Allocate 250 bytes:



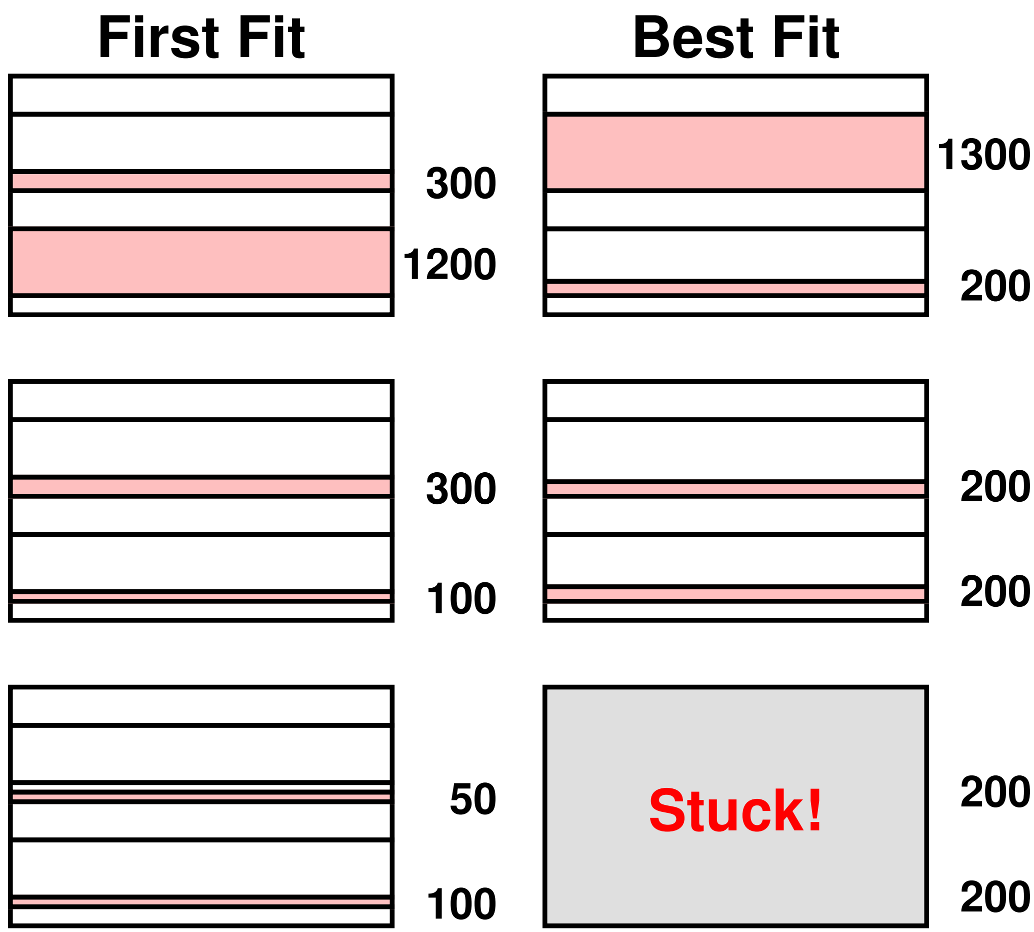
Allocation Example



Allocate 1000 bytes:

Allocate 1100 bytes:

Allocate 250 bytes:



Fragmentation

- ***First-fit*** **vs.** ***best-fit*** **allocation**
  - **studies have shown that first-fit works better**
  - **best-fit tends to leave behind a large number of regions of memory that are too small to be useful**
    - **best-fit tends to create smallest left-over blocks!**
  - **this is the general problem of** ***fragmentation***
    - ***internal fragmentation:*** **unusable memory is contained within an allocated region (e.g., buddy system)**
    - ***external fragmentation:*** **unusable memory is separated into small blocks and is interspersed by allocated memory (e.g., best-fit)**