Housekeeping (Lecture 9 - 9/25/2013)

- **Warmup #2 due at 11:45pm on Friday, 10/4/2013**
  - if you have code from a previous semester, be very careful and *not copy any code from it*
    - it's best if you just get rid of it
  - get started soon
- *Grading guidelines* is the *ONLY* way we will grade and we will grade on `nunki.usc.edu` in our grading account (which you don't have access to)
  - it's a good idea to run your code against the grading guidelines
- After submission, make sure you *Verify Your Submission*
- Have you installed *Ubuntu 11.10* on your laptop/desktop?
- Do you have partners for kernel assignments?
  - work with your potential partners for warmup 2
    - again, work at high level and must *not* share code

Context Switch

- **The big idea here is that in order to perform a context switch, you must first save your context**
  - **therefore, you must know what constitutes the context**
  - **then you save all of it**
    - **what's the minimum amount of context to save?**
    - **context can be stored in several places**
      - **stack**
      - **thread control block (e.g., in a system call, the TCB contains a pointer to *both* the corresponding user stack frame and the kernel stack frame)**
  - **when switching back, you must restore the context**

3.1 Context Switching

- **Procedures**
- **Threads & Coroutines**
- **Systems Calls**
- ***Interrupts***

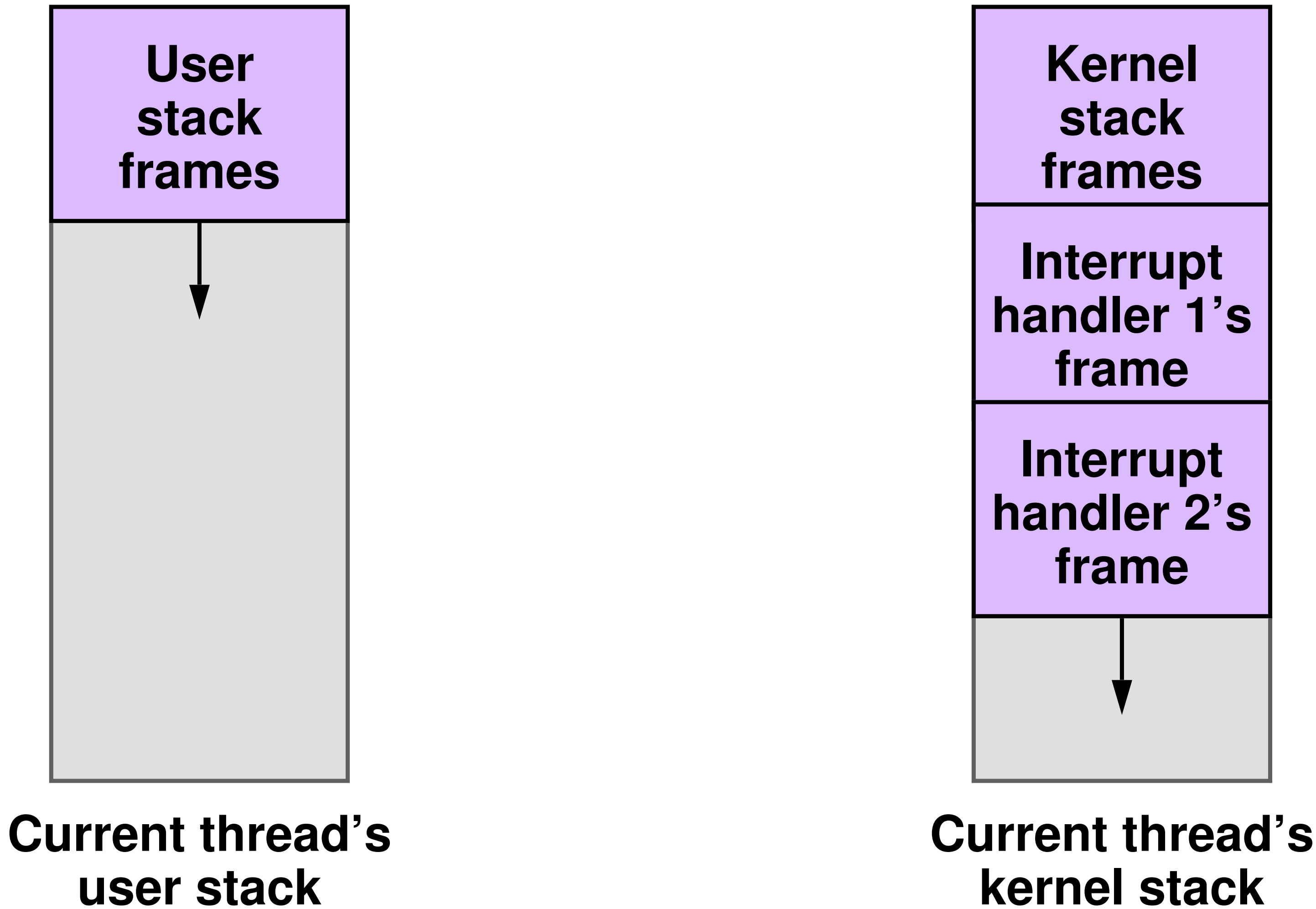
Interrupts

- Recall that *signals* are *generated by the kernel*
  - they are delivered to the *user process*
  - signals are "software interrupts"
- *Interrupts* are *generated by the hardware*
  - they are delivered to the *kernel*

Interrupts

- When an *interrupt* occurs, the processor puts aside the current context and switch to an *interrupt context*
  - the current context can be a *thread context* or *another interrupt context*
  - when the interrupt handler is finishes, the processor generally resumes the original context
- Interrupt context needs a stack
  - which stack should it use?
  - there are several possibilities
    1) allocate a new stack each time an interrupt occurs
       - too slow
    2) have one stack shared by all interrupt handlers
       - not often done
    3) interrupt handler could borrow a stack from the thread it is interrupting
       - most common

Borrowing Stack From Current Thread



Interrupts

- For approaches (2) and (3), there is no way to suspend one interrupt handler and *resume* the execution of another
  - since there is only one stack for all the interrupt handlers
  - therefore, the handler of the most recent interrupt must *run to completion*
    - when it's done, the stack frame is removed, and the next-most-recent interrupt now must run to completion
  - this is a big deal!
    - once you have interrupt handlers running, a normal thread (no matter how important it is) cannot run until *all* interrupt handlers complete
    - if we have approach (1), then we won't have this problem

Interrupts

- There is another approach
  - interrupt handler places a description of the work that must be done on a queue of some sort, then arranges for it to be done in some other context at a later time
  - this approach is used in many systems, including Windows and Linux
    - will discuss further in Ch 5

Interrupt Mask

- Interrupt can be *masked*, i.e., temporarily blocked
  - if an interrupt occurs while it is masked, the interrupt indication remains pending
  - once it is unmasked, the processor is interrupted

- How interrupts are masked is architecture-dependent
  - common approaches
    1) hardware register implements a bit vector
       - if a particular bit is set, the corresponding interrupt class is masked
       - the kernel masks interrupts by setting bits in the register
       - when an interrupt does occur, the corresponding mask bit is set in the register (block other interrupts of the same class)
       - cleared when the handler returns
    2) hierarchical interrupt levels

Interrupt Mask

- Interrupt can be ***masked***, i.e., temporarily blocked
  - if an interrupt occurs while it is masked, the interrupt indication remains pending
  - once it is unmasked, the processor is interrupted

- How interrupts are masked is architecture-dependent
  - common approaches
    1) hardware register implements a bit vector
    2) hierarchical interrupt levels (more common)
       - the processor masks interrupts by setting an interrupt priority level (IPL) in a hardware register
       - all interrupts with the current or lower levels are masked
       - the kernel masks a class of interrupts by setting the IPL to a particular value
       - when an interrupt does occur, the current IPL is set to that of the level the interrupt belongs
       - restores to previous value on handler return

3.2 Input/Output Architectures

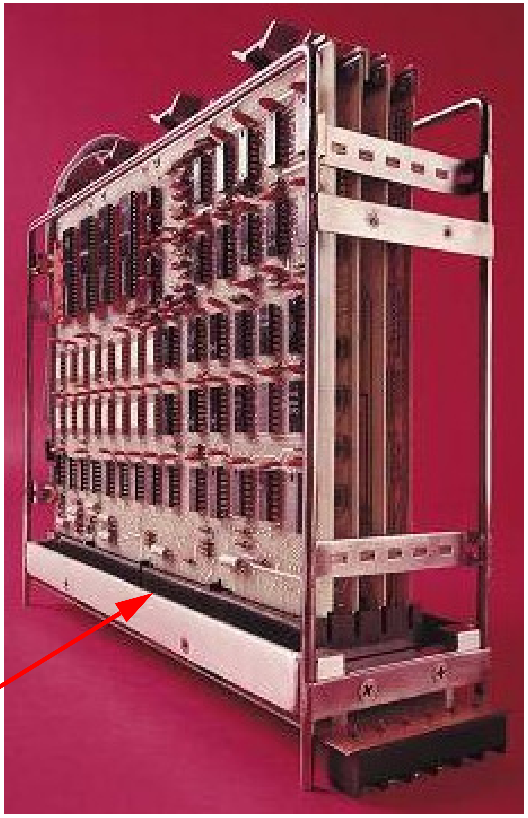
Input/Output

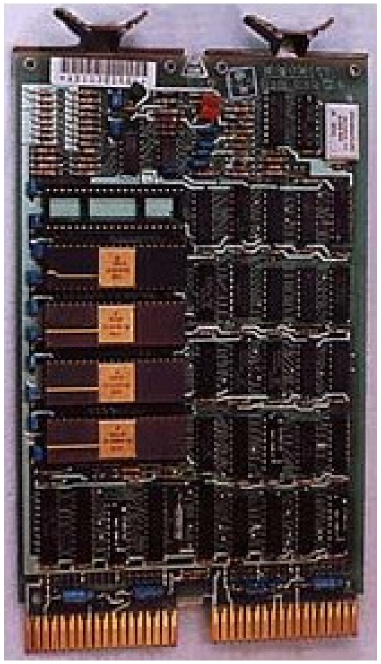
- **Architectural concerns**
  - memory-mapped I/O
    - programmed I/O (PIO)
    - direct memory access (DMA)
  - I/O processors (channels)
- **Software concerns**
  - device drivers
  - concurrency of I/O and computation

What Does A Computer Look Like?

- LSI-11
  - processor for PDP-11
- Boards are connected over a "bus"
  - on the "backplane"
  - various standards for PDP-11
    - Unibus, Q-Bus, etc.

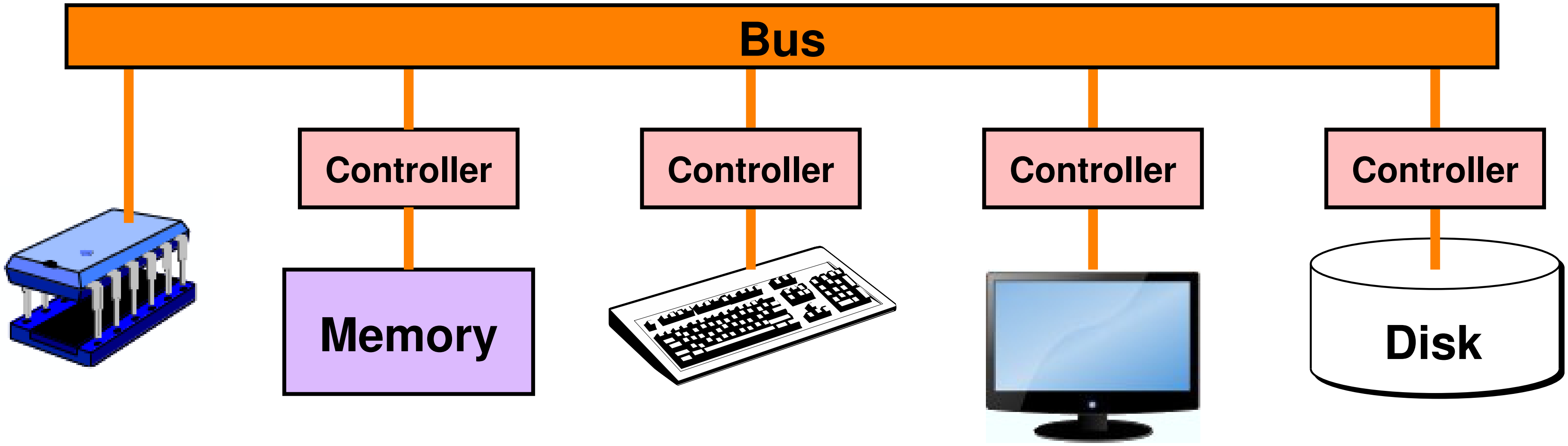
connect to backplane bus





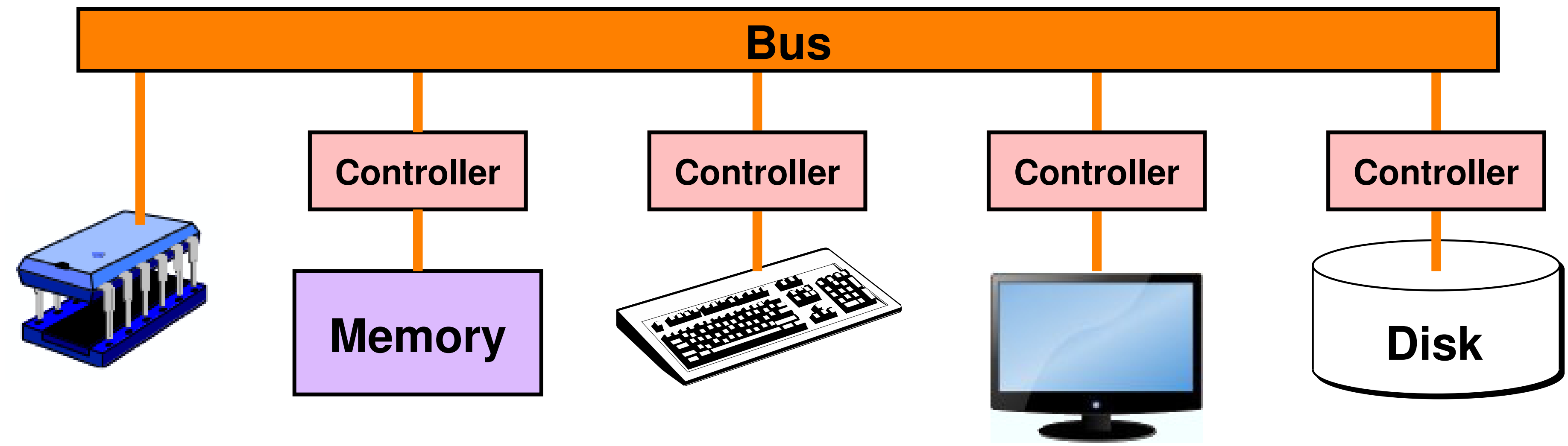
http://hampage.hu/pdp-11/lsi11.html

Simple I/O Architecture



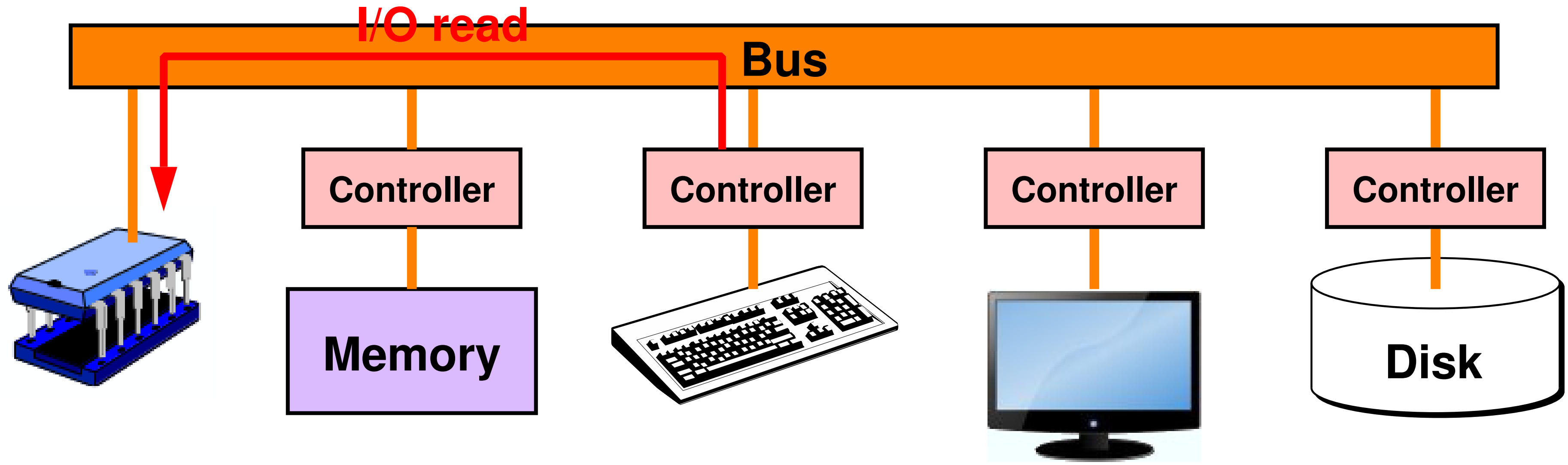
- memory-mapped I/O
  - all controllers listen on the bus to determine if a request is for itself or not
  - memory controller behaves differently from other controllers, i.e., it passes the bus request to primary memory
  - others "process" the bus request
    - and respond to relatively few addresses
  - memory is not really a "device"

Simple I/O Architecture



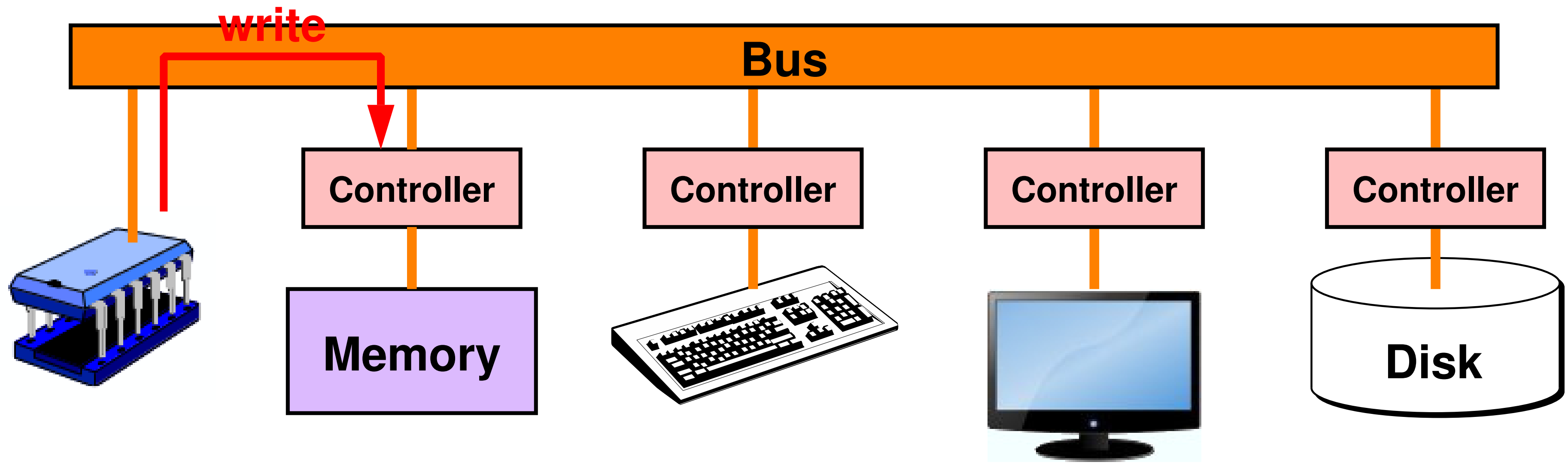
- memory-mapped I/O
- two categories of devices
  - PIO (programmed I/O)
    - perform I/O operations by reading or writing data in the controller registers one byte or word at a time over the bus

Simple I/O Architecture



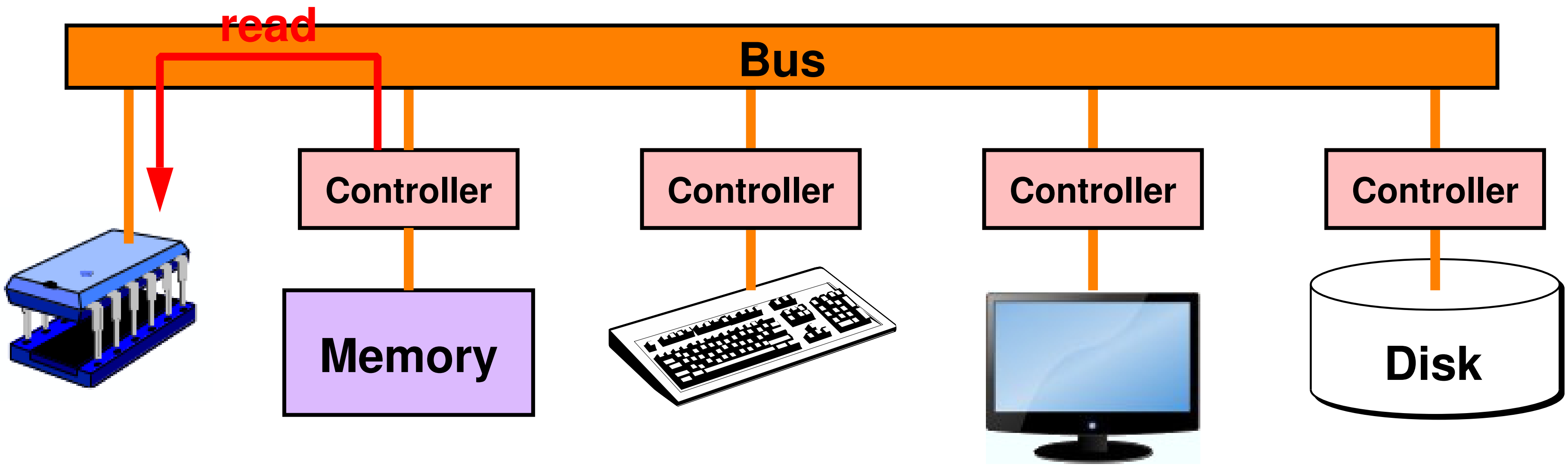
- memory-mapped I/O
- two categories of devices
  - PIO (programmed I/O)
    - perform I/O operations by reading or writing data in the controller registers one byte or word at a time over the bus

Simple I/O Architecture



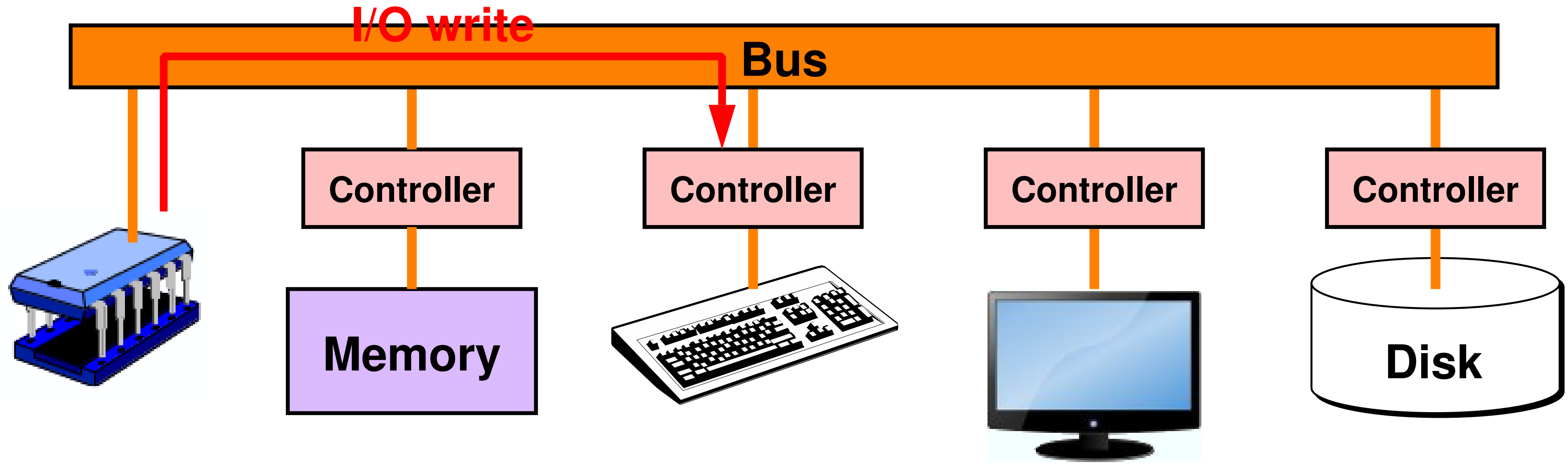
- memory-mapped I/O
- two categories of devices
  - PIO (programmed I/O)
    - perform I/O operations by reading or writing data in the controller registers one byte or word at a time over the bus

Simple I/O Architecture



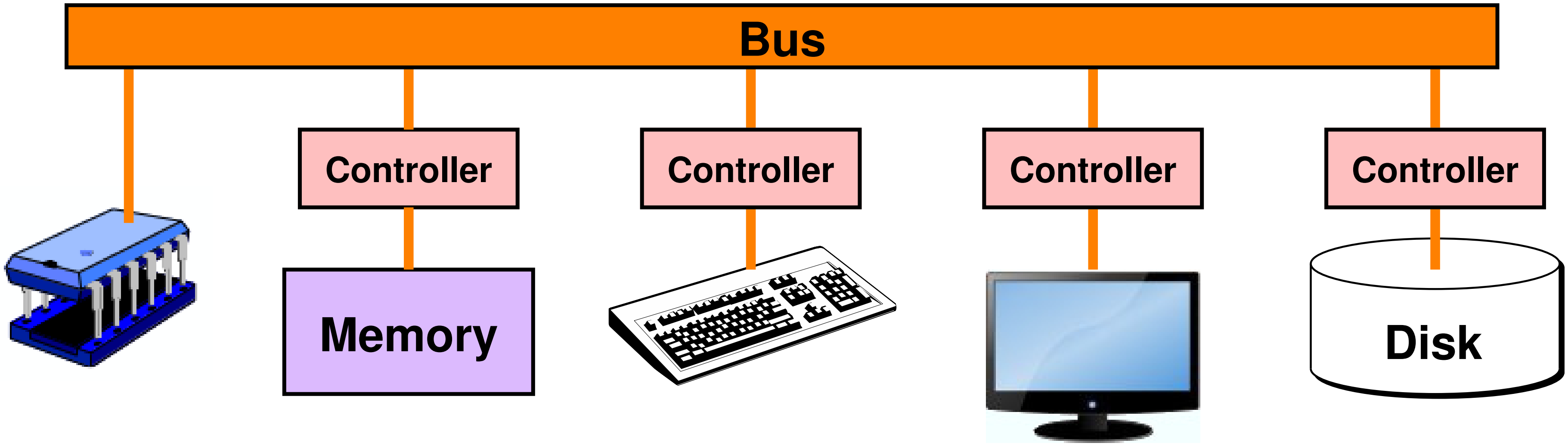
- memory-mapped I/O
- two categories of devices
  - PIO (programmed I/O)
    - perform I/O operations by reading or writing data in the controller registers one byte or word at a time over the bus

Simple I/O Architecture



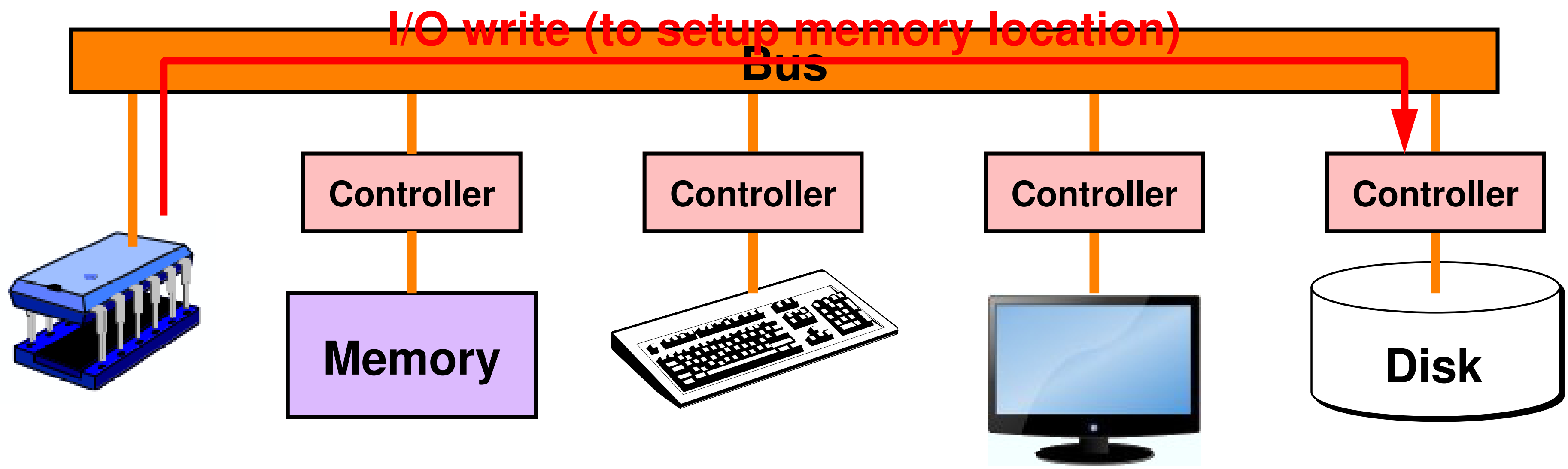
- memory-mapped I/O
- two categories of devices
  - PIO (programmed I/O)
    - perform I/O operations by reading or writing data in the controller registers one byte or word at a time over the bus

Simple I/O Architecture



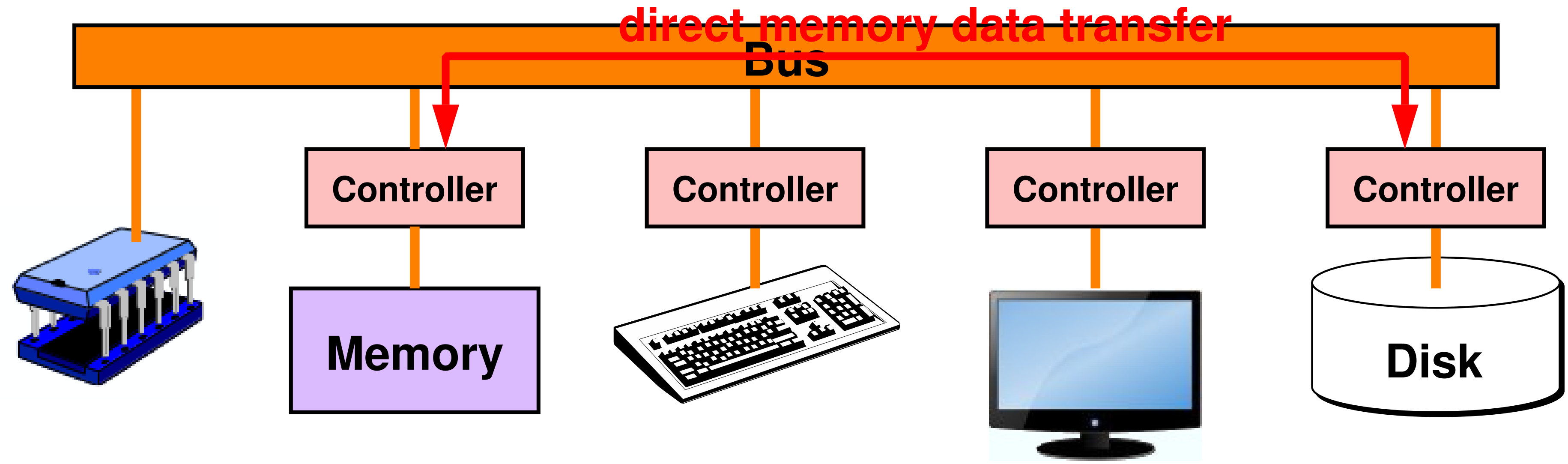
- memory-mapped I/O
- two categories of devices
  - PIO (programmed I/O)
  - DMA (direct memory access)
    - the controller performs the I/O itself
    - the processor writes to the controller to tell it where to transfer the results to
    - the controller takes over and transfers data between itself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

Simple I/O Architecture



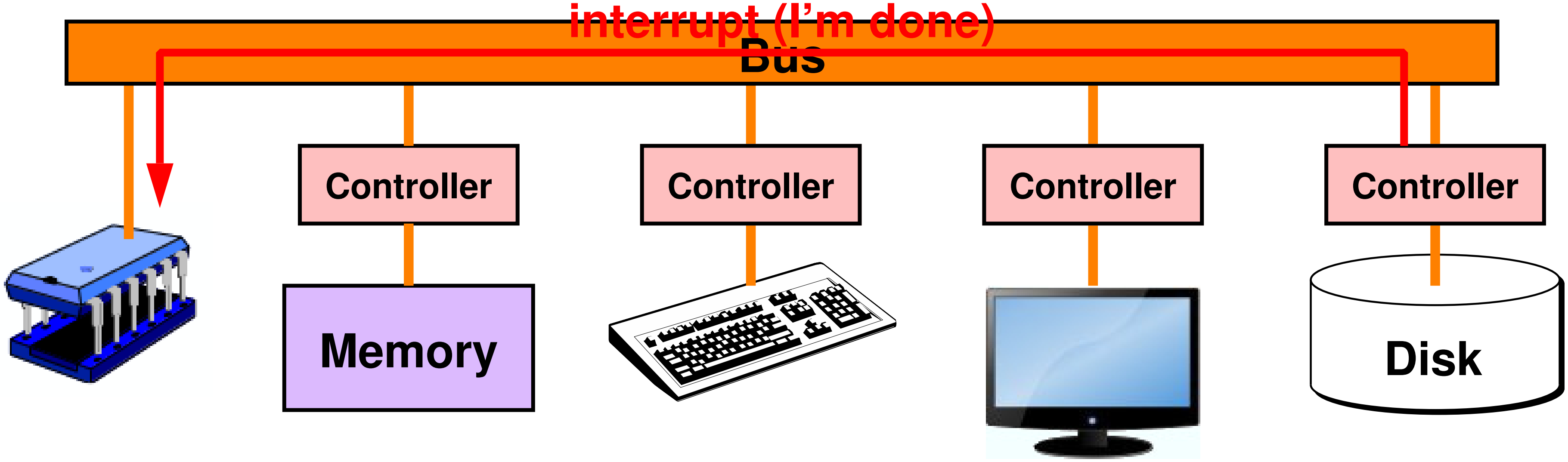
- memory-mapped I/O
- two categories of devices
  - PIO (programmed I/O)
  - DMA (direct memory access)
    - the controller performs the I/O itself
    - the processor writes to the controller to tell it where to transfer the results to
    - the controller takes over and transfers data between itself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

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

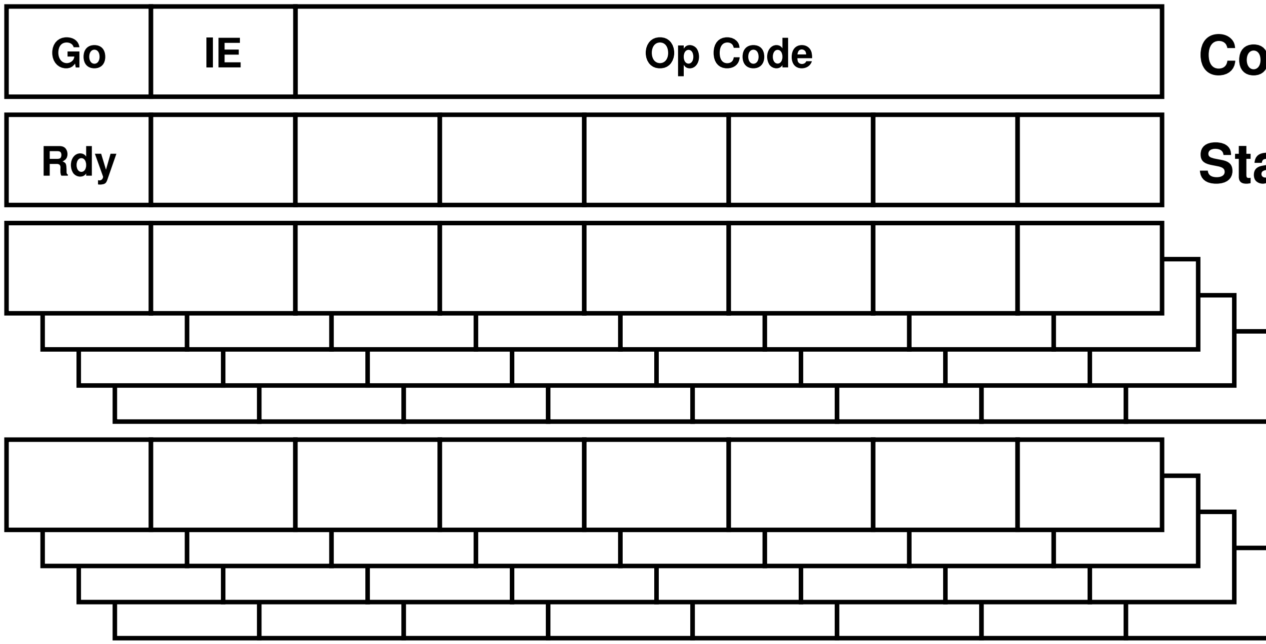
Legend:		
	GoR	Go read (start a read operation)
	GoW	Go write (start a write operation)
	IER	Enable read-completion interrupts
	IEW	Enable write-completion interrupts
	RdyR	Ready to read
	RdyW	Ready to write

- This is the abstraction of a PIO device
  - you need to know this well for your project

Programmed I/O

- E.g.: Terminal controller (in the simulator, i.e., class projects)
- 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



Control register (1 byte)

Status register (1 byte)

Memory address register (4 bytes)

Device address register (4 bytes)

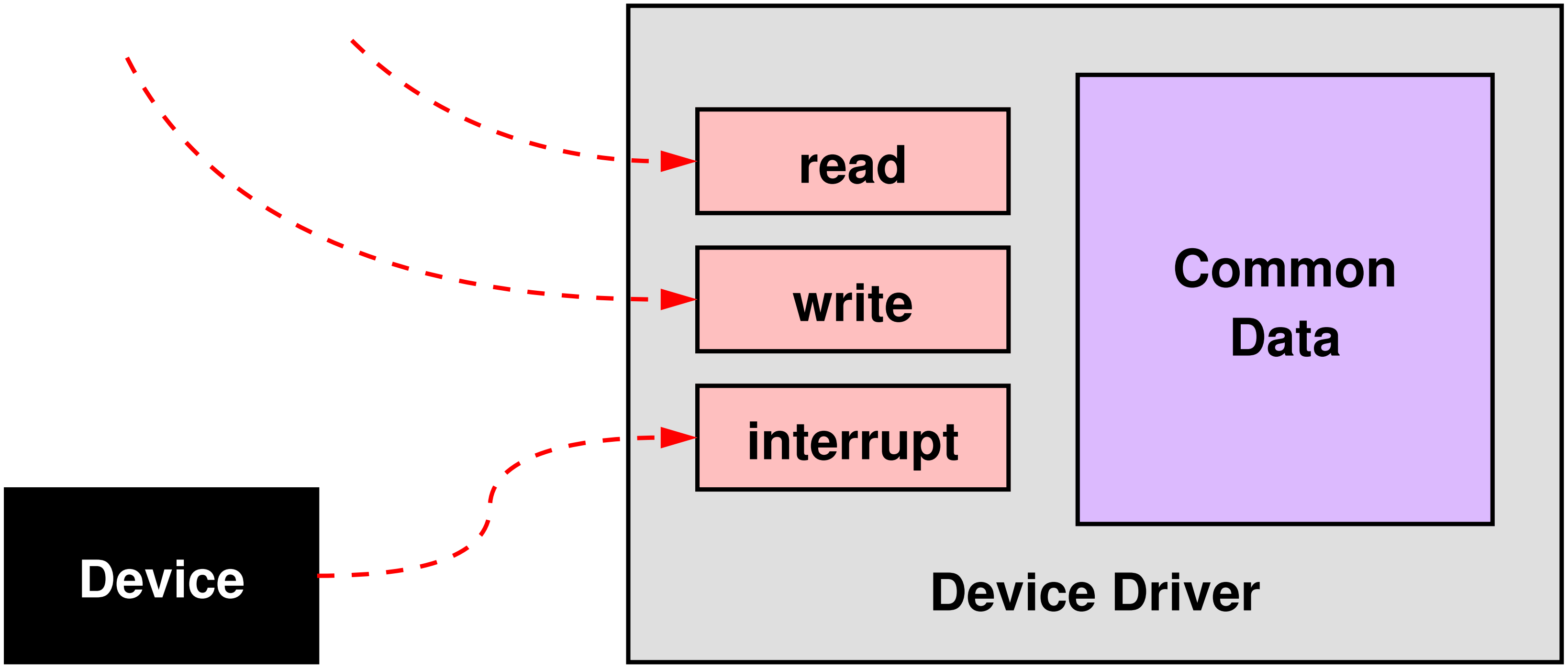
Legend:	Go	Start an operation
	Op Code	Operation code (identifies the operation)
	IE	Enable interrupts
	Rdy	Controller is ready

- **This is the abstraction of a DMA device**
  - **you need to know this well for your project**

Direct Memory Access

- E.g.: Disk controller (in the simulator, i.e., class projects)
- 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



- 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 how to talk to devices)
  - OS can treat I/O in a *device-indepdendent* manner

... in C++

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

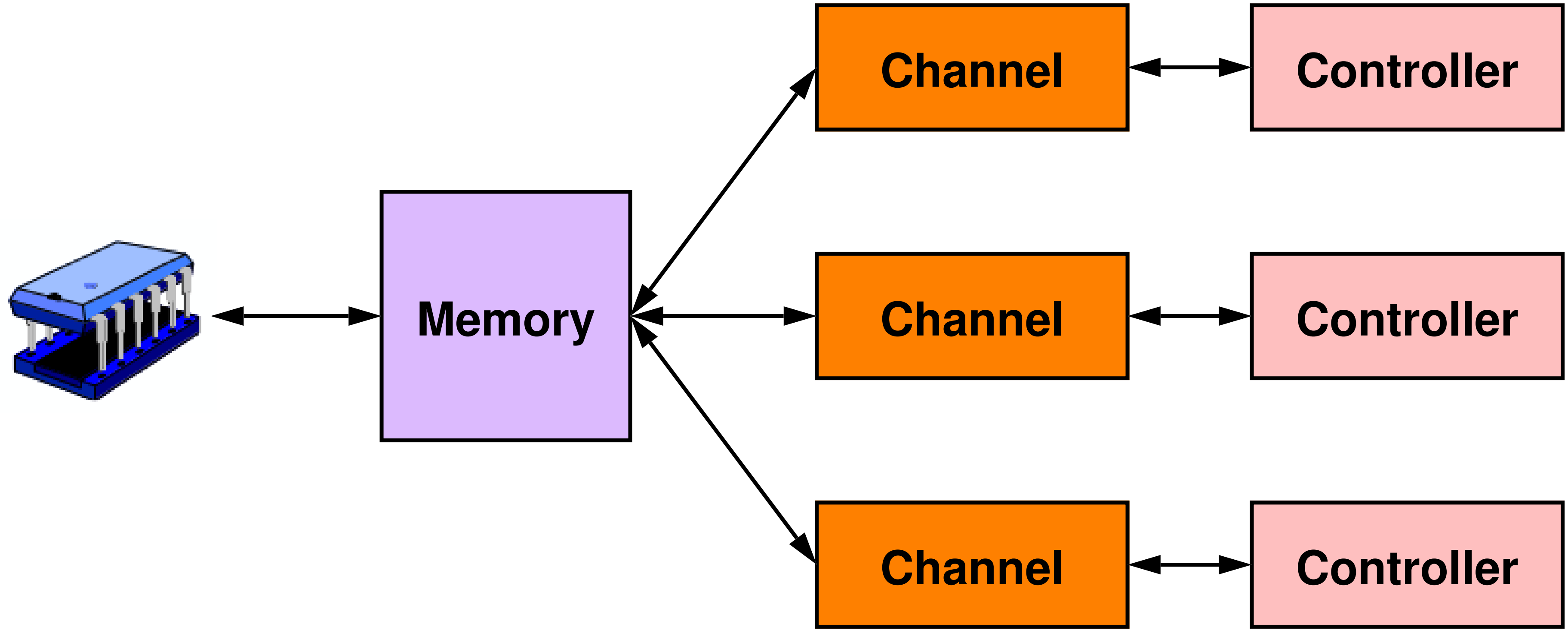
- this is a synchronous interface
  - a user thread would call the `read/write()` method
  - this starts the device and the user thread would block
  - the device driver's interrupt method is called in the interrupt context
    - if I/O is completed, the thread is unblocked and return from the `read/write()` method

A Bit More Realistic

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

- 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
    - it's possible for multiple threads to invoke `wait()` with the same handle, if they all want the same block from a file

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

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.

Dynamic Storage Allocation

- Goal: allow dynamic creation and destruction of data structures
- Concerns:
  - efficient use of storage
  - efficient use of processor time
- Example:
  - *first-fit* vs. *best-fit* allocation

Allocation Example

1300 bytes free

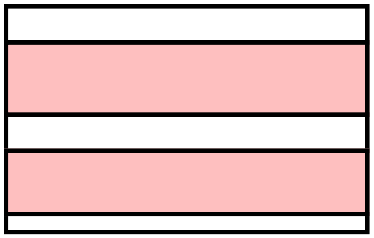
1200 bytes free

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

Allocate 1100 bytes:

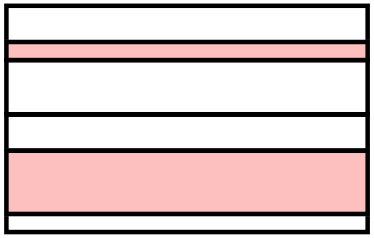
Allocate 250 bytes:

Allocation Example



Allocate 1000 bytes:

First Fit

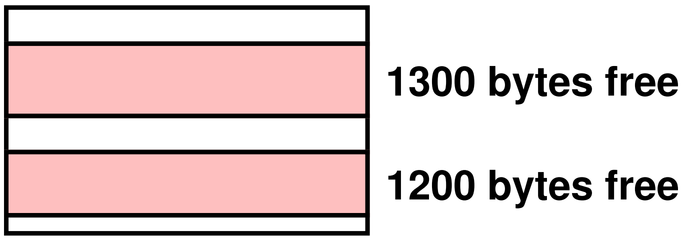


Best Fit

Allocate 1100 bytes:

Allocate 250 bytes:

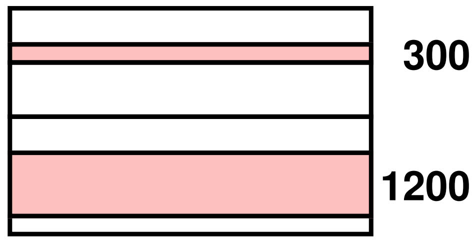
Allocation Example



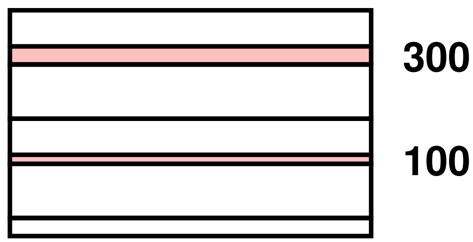
Allocate 1000 bytes:

First Fit

Best Fit

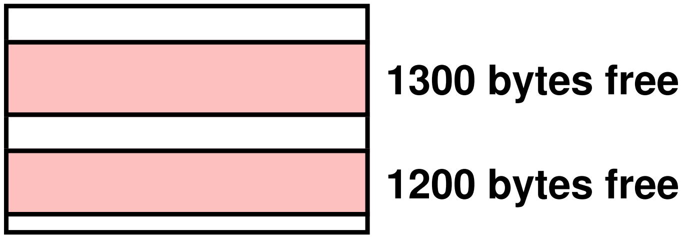


Allocate 1100 bytes:

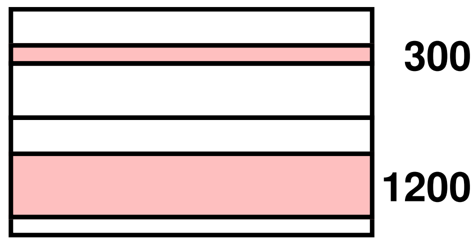


Allocate 250 bytes:

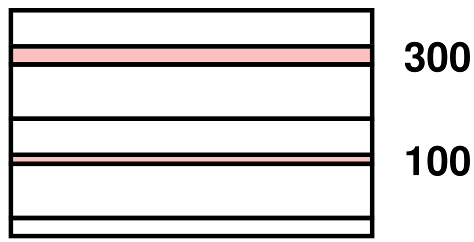
Allocation Example



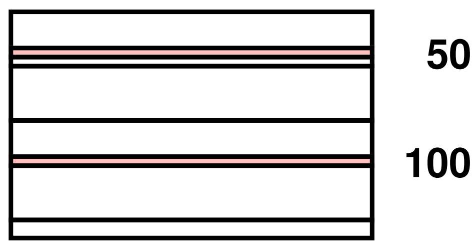
Allocate 1000 bytes: | **First Fit** | **Best Fit**



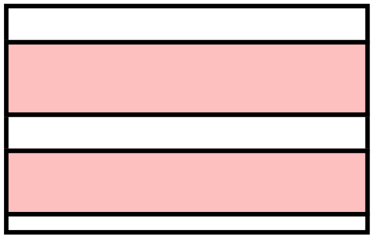
Allocate 1100 bytes:



Allocate 250 bytes:

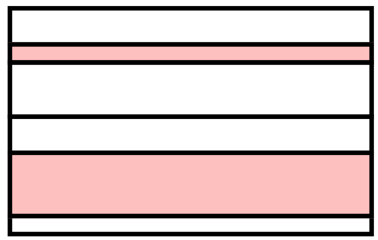


Allocation Example

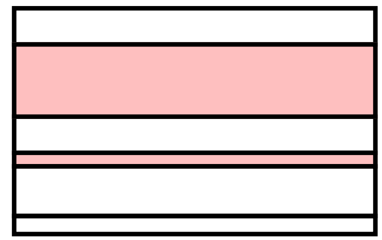


Allocate 1000 bytes:

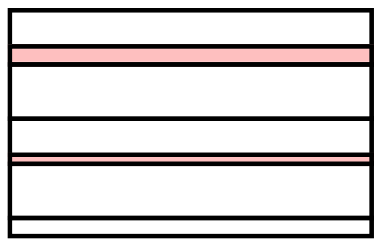
First Fit



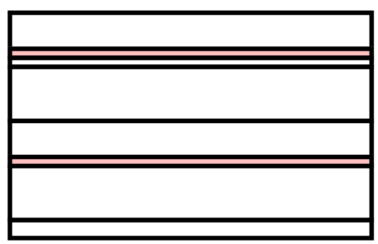
Best Fit



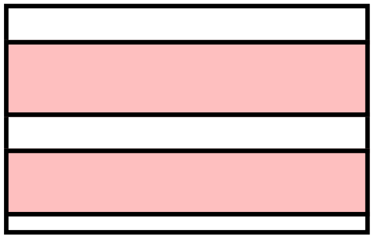
Allocate 1100 bytes:



Allocate 250 bytes:

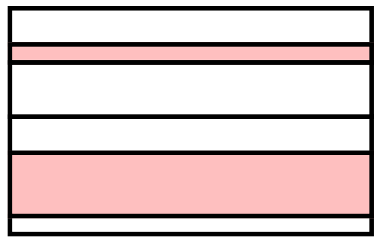


Allocation Example

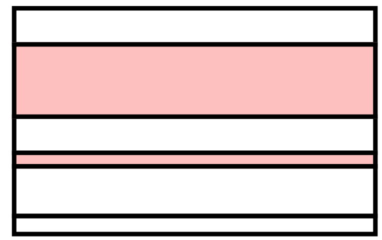


Allocate 1000 bytes:

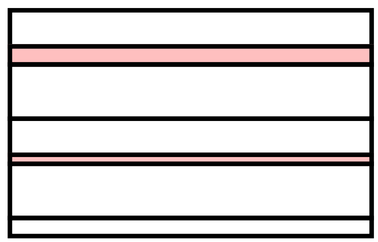
First Fit



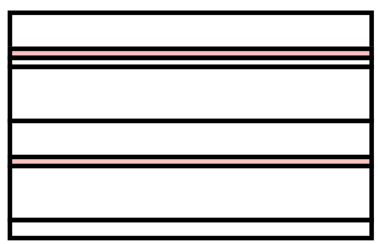
Best Fit



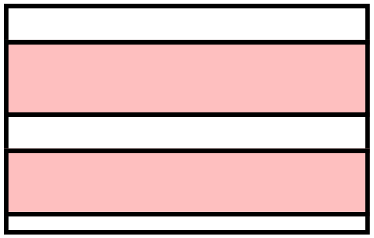
Allocate 1100 bytes:



Allocate 250 bytes:



Allocation Example

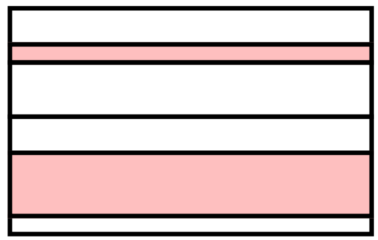


1300 bytes free

1200 bytes free

Allocate 1000 bytes:

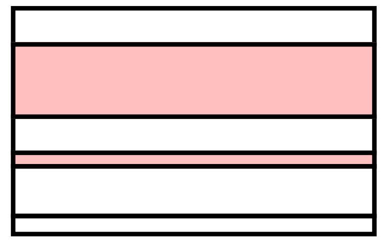
First Fit



300

1200

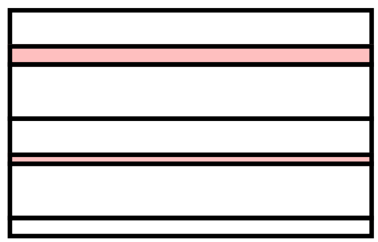
Best Fit



1300

200

Allocate 1100 bytes:



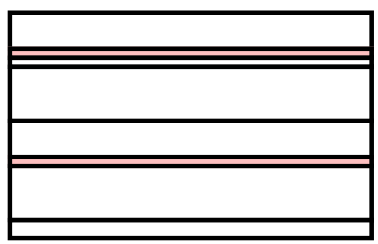
300

100

200

200

Allocate 250 bytes:



50

100

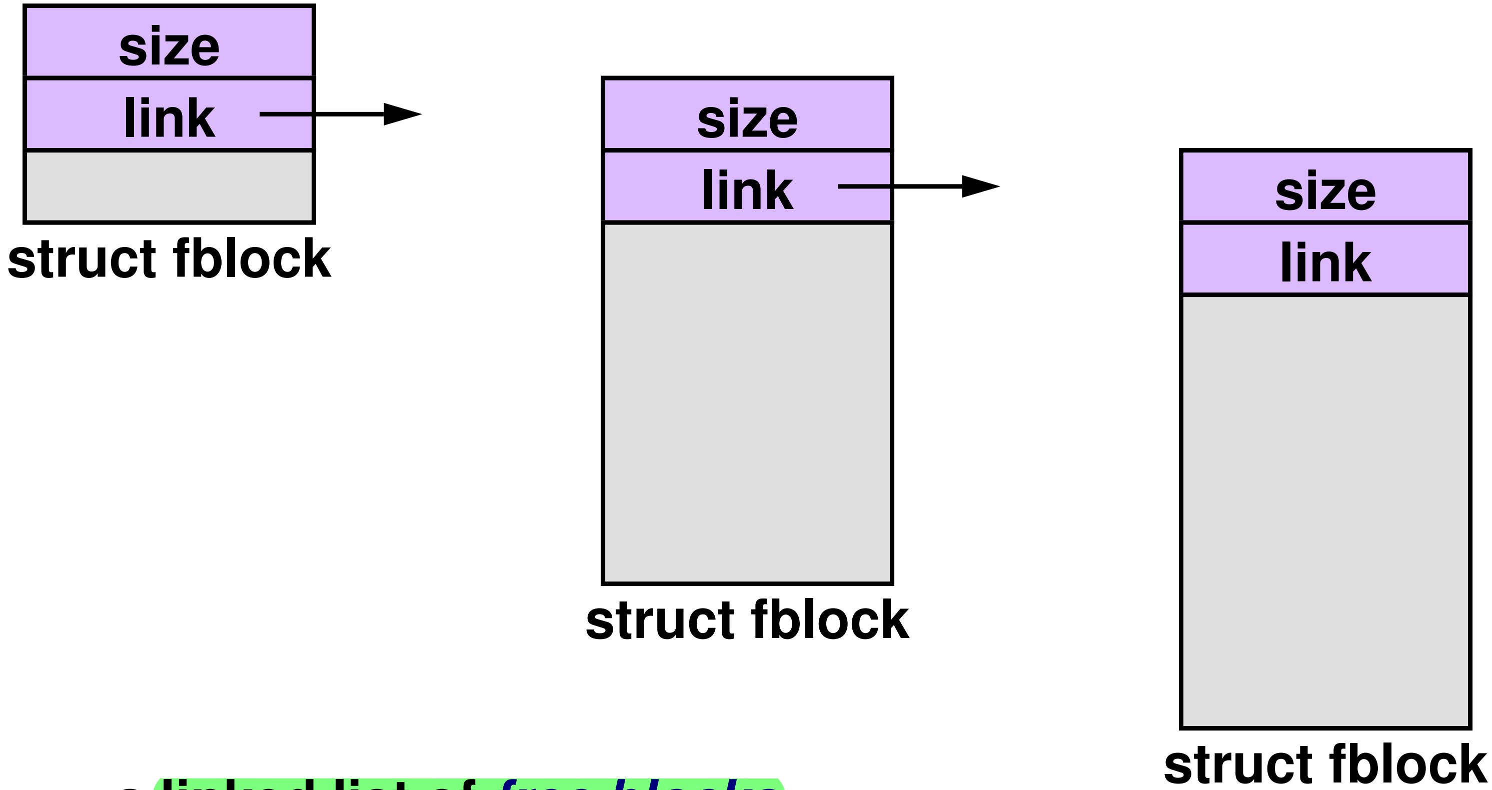
200

200

Fragmentation

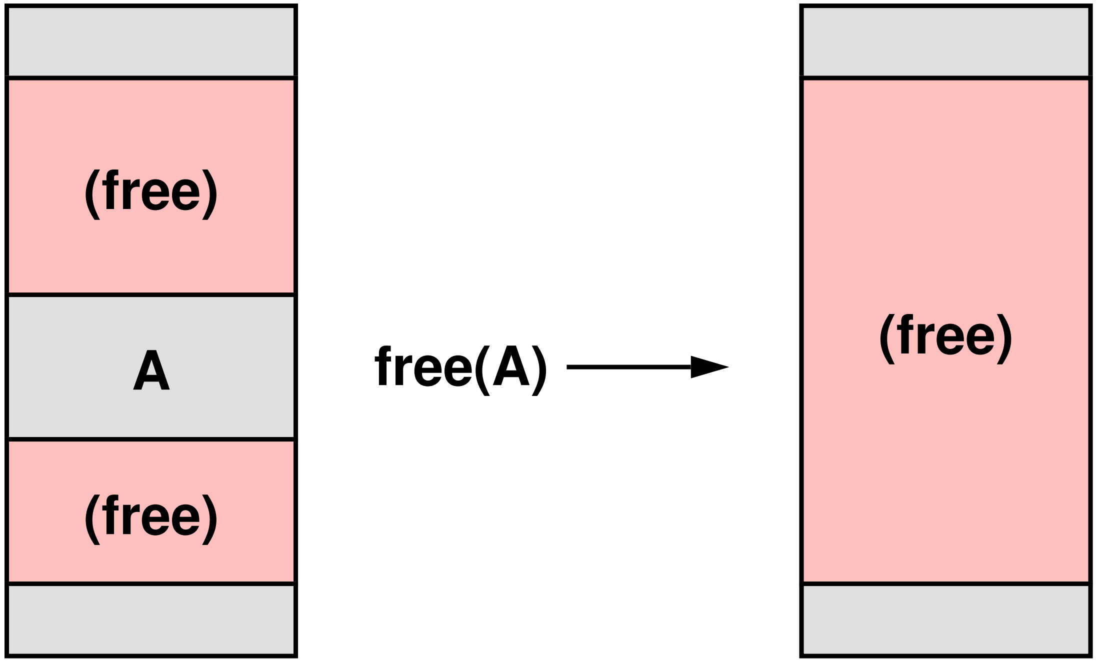
- ***First-fit*** vs. ***best-fit*** allocation
  - studies have shown that first-fit works better
  - best-fit tends to leave behind a large number of regions of memory that are too small to be useful
    - best-fit tends to create smallest left-over blocks!
  - this is the general problem of ***fragmentation***
    - ***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)

Implementing First Fit: Data Structures



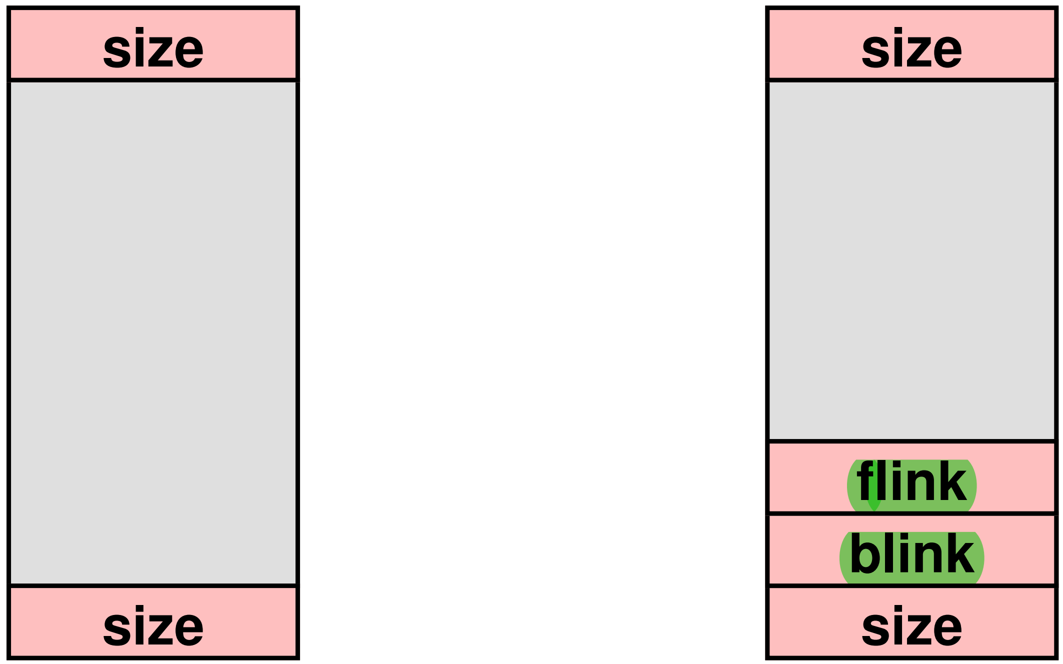
- a linked list of *free blocks*
  - don't need to manage allocated blocks
- use a doubly-linked list
  - insertion and deletion are fast, i.e., *O(1)*, once you know where to insert or delete

Liberation of Storage



- This is known as ***coalescing***
  - in order to make coalescing possible, you need to know that *size* of the blocks above and below the block being freed
    - you also need to know if they are *allocated* or *free*

Boundary Tags

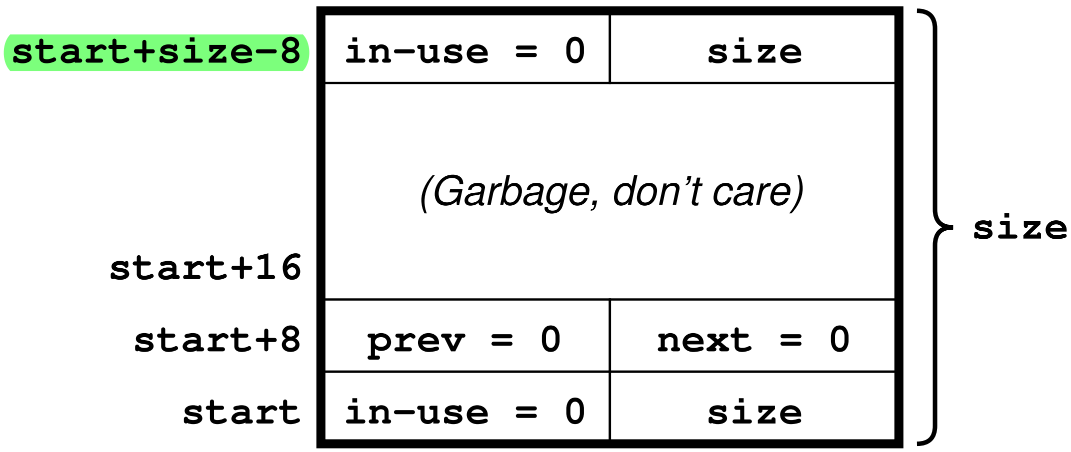


Allocated Block　　Free Block

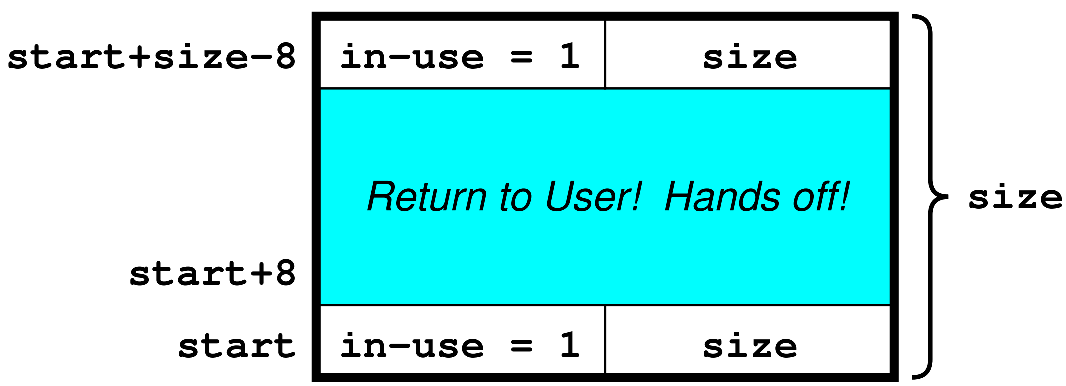
- This is known as *coalescing*
  - in order to make coalescing possible, you need to know that *size* of the blocks above and below the block being freed
    - you also need to know if they are *allocated* or *free*

Detailed Examples

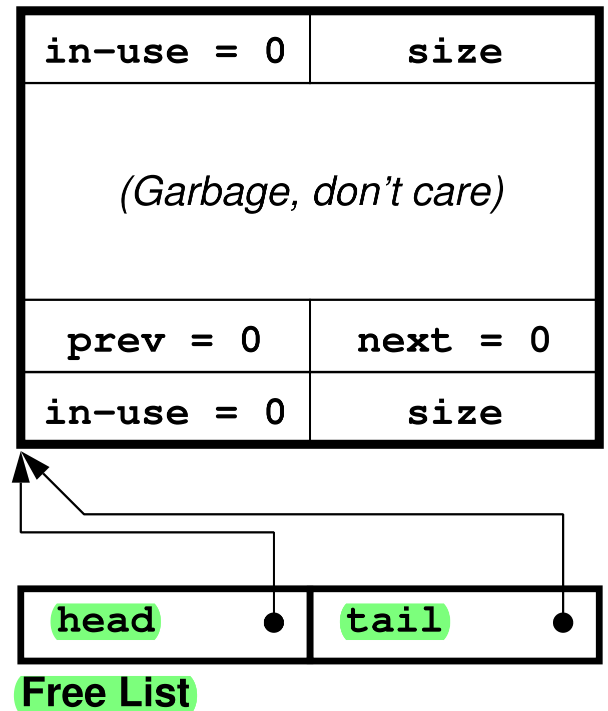
Free block



In-use block

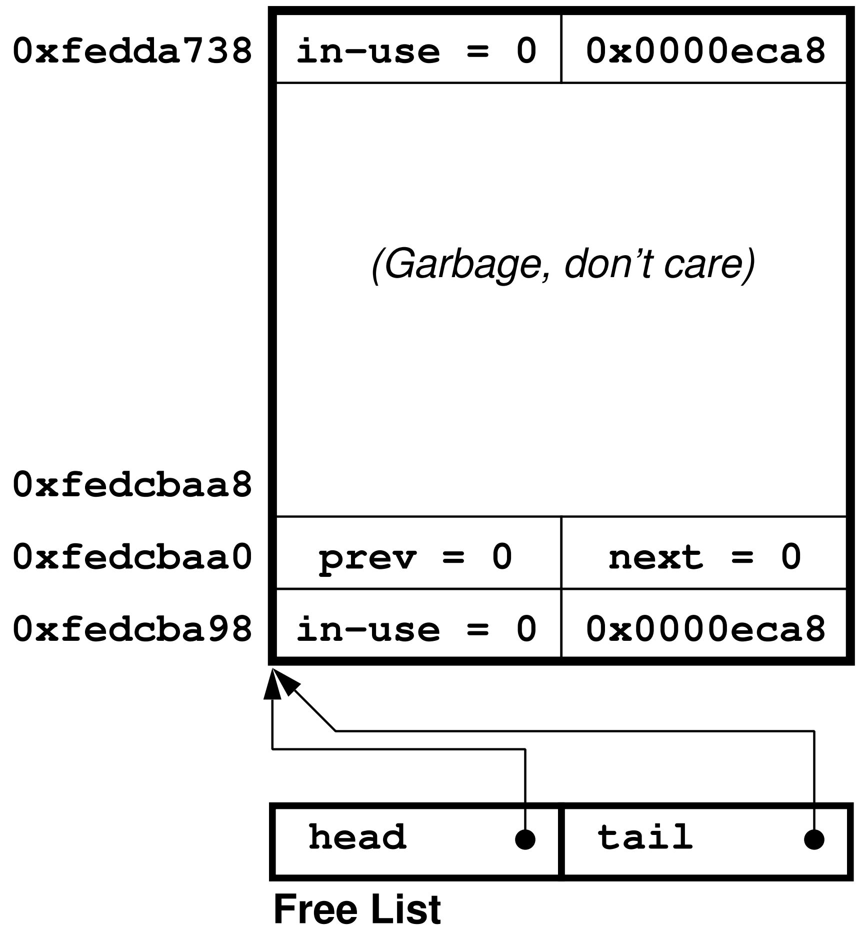


Free list



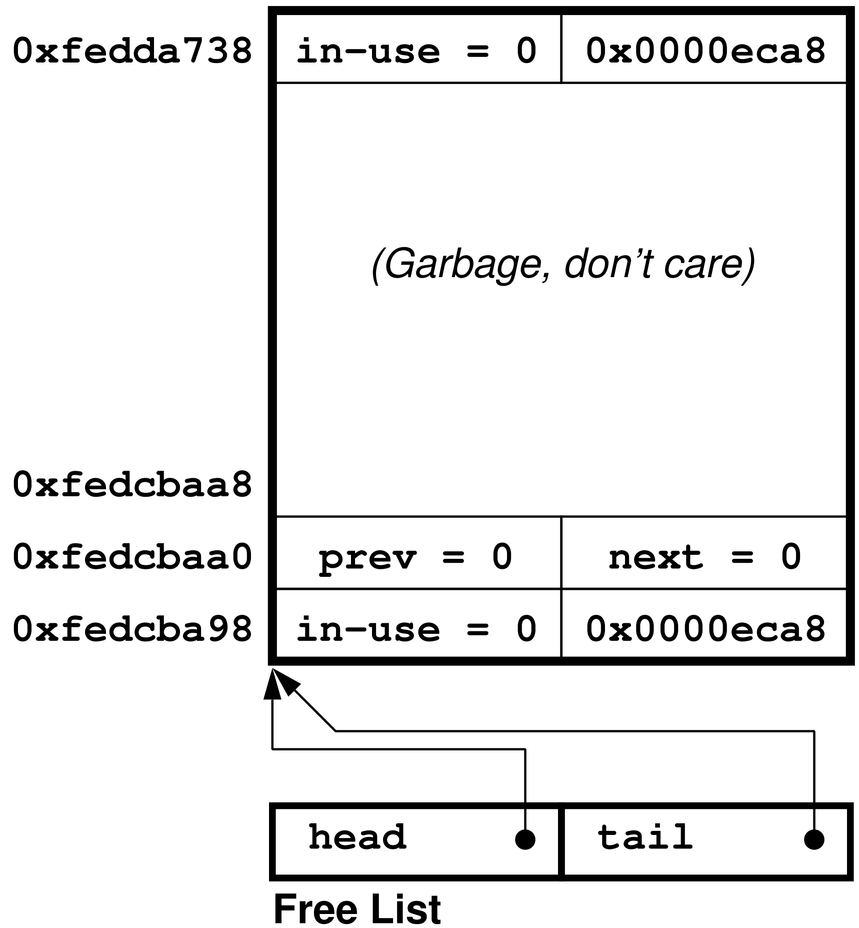
`malloc()` Example

- Ex: Heap starts at `0xfedcba98` and size of the heap is `0x0000eca8` (60,584) bytes
  - the Free List contains one free block and it looks like this:



`malloc()` Example

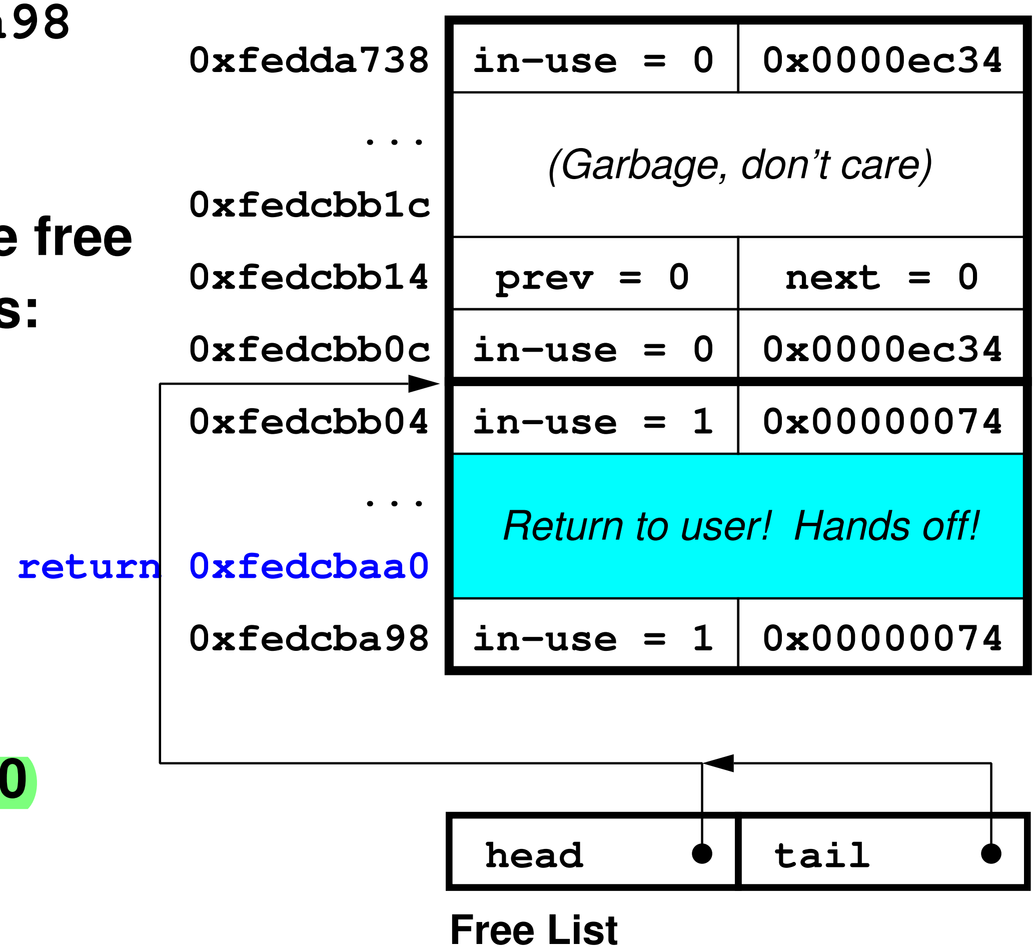
- Ex: Heap starts at `0xfedcba98` and size of the heap is `0x0000eca8` (60,584) bytes
  - the Free List contains one free block and it looks like this:



- Ex: Request block size is 100
  - split the block into two
  - busy block size is 116
  - remaining free block size is 60584-116 =60468=0xec34

`malloc()` Example

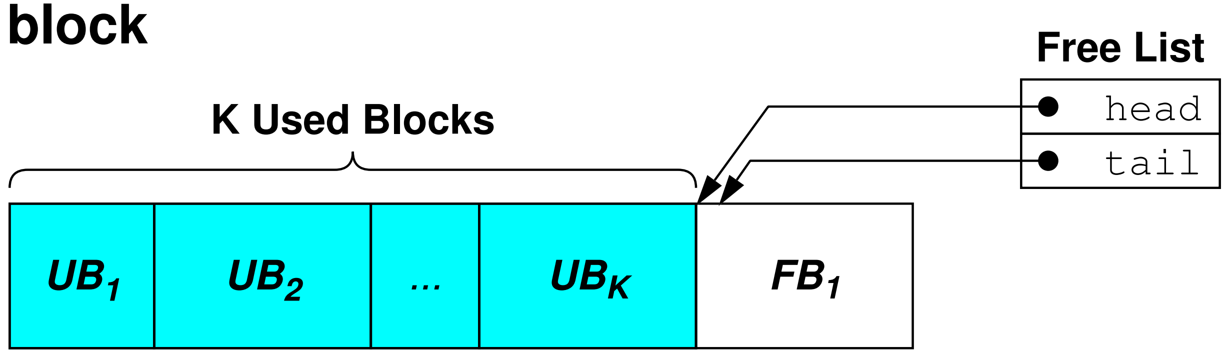
- Ex: Heap starts at `0xfedcba98` and size of the heap is `0x0000eca8` (60,584) bytes
  - the Free List contains one free block and it looks like this:



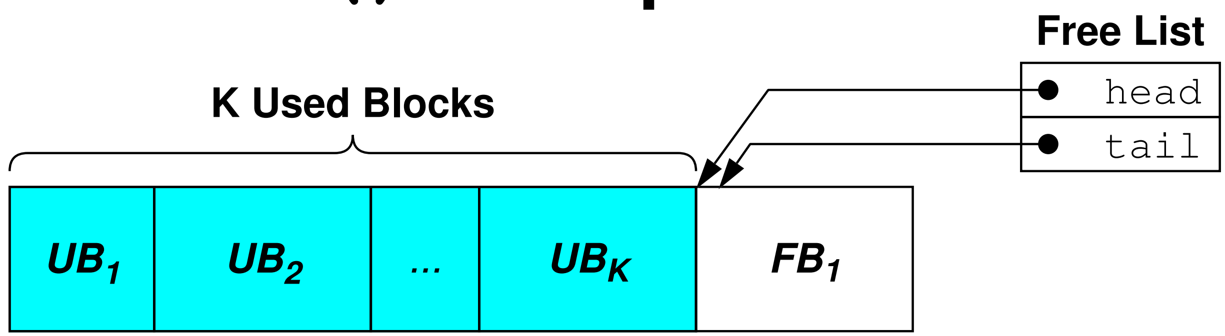
- Ex: Request block size is 100
  - split the block into two
  - busy block size is 116
  - remaining free block size is 60584-116 =60468=0xec34

`free()` Example

- After *K* blocks of memory have been allocated (and assume that none of them have been deallocated)
  - in the memory layout, the first *K* blocks are used block, followed by one free block



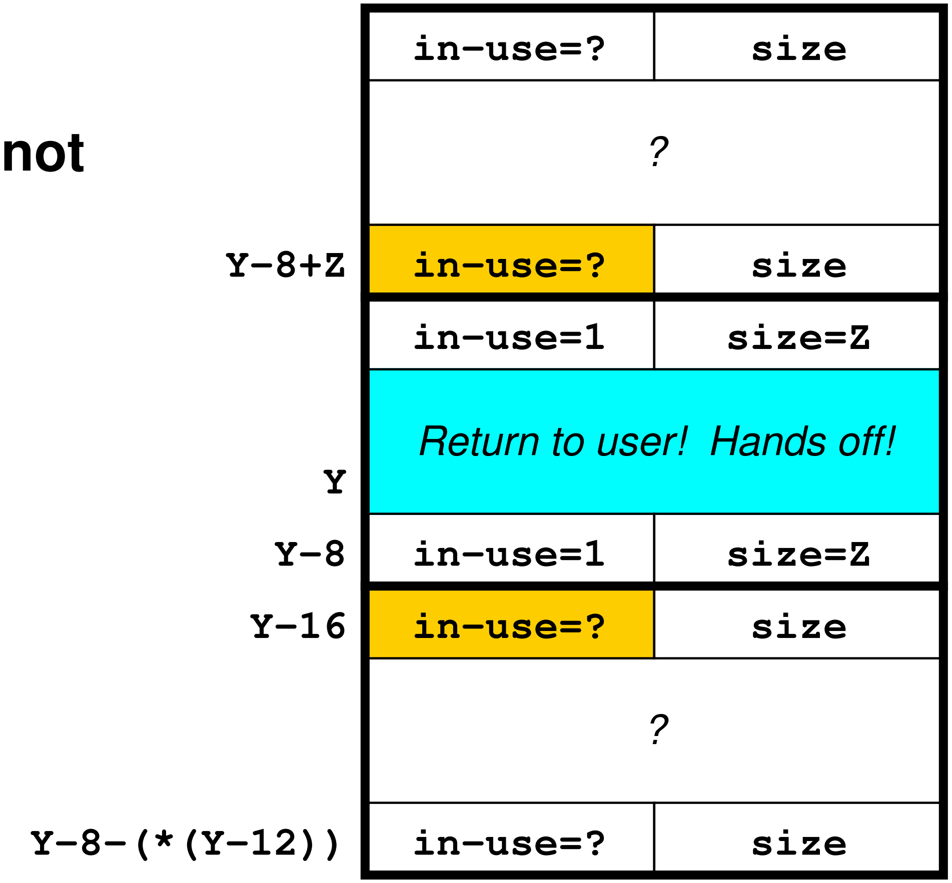
`free()` Example



- Memory blocks can be freed in any order
  - when a memory block is freed, we need to check if the block before it and after it are also free
- If neither of them are free, we just need to insert the newly freed block into the Free List (at the right place)
  - need to *search* the Free List to find insertion point
  - searching through a linear list is "slow", *O(n)*
- Otherwise, we can *merge* the block in question with neighboring free block(s)
  - this is known as *coalescing*

`free()` Example

- Ex: `free(Y)`
  - `Y-16` tells you if the *previous* block is free or not
  - `Y-8+Z` tells you if the *next* block is free or not
    - where `Z` is what's in `Y-4`
- ***Coalescing:***
  - need to make sure that everything is consistent



`free()` Example

- Ex: `free(Y)` and *previous block is free* and *next block is busy*
  - i.e., `Y-16` is `0` and `Y-8+Z` is `1`
    - where `Z` is what's in `Y-4` and `W` is what's in `Y-12`
  - furthermore, `Y-8-W` is on the Free List
  - coalesce this block and the *previous* block

Address		
	`in-use=1`	`size`
	Hands off!	
`Y-8+Z`	`in-use=1`	`size`
`Y-16+Z`	`in-use=1`	`size=Z`
`Y`	*Return to user! Hands off!*	
`Y-8`	`in-use=1`	`size=Z`
`Y-16`	`in-use=0`	`size=W`
	(Garbage, don't care)	
	`prev`	`next`
`Y-8-W`	`in-use=0`	`size=W`

`free()` Example

- Ex: `free(Y)` and ***previous block is free*** and ***next block is busy***
  - i.e., `Y-16` is `0` and `Y-8+Z` is `1`
    - where `Z` is what's in `Y-4` and `W` is what's in `Y-12`
  - furthermore, `Y-8-W` is on the Free List
  - coalesce this block and the ***previous*** block
    - easy!
    - just change `Y-12+Z` and `Y-4-W` to `W+Z` and `Y-16+Z` to `0`
    - don't even need to change prev and next!

	`in-use=1`	`size`
	Hands off!	
`Y-8+Z`	`in-use=1`	`size`
`Y-16+Z`	`in-use=0`	`size=W+Z`
`Y`	*(Garbage, don't care)*	
	`prev`	`next`
`Y-8-W`	`in-use=0`	`size=W+Z`

`free()` Example

- Ex: `free(Y)` and ***previous block is busy*** and ***next block is free***
  - i.e., `Y-16` is `1` and `Y-8+Z` is `0`
    - where `Z` is what's in `Y-4` and `X` is what's in `Y-4+Z`
  - furthermore, `Y-8+Z` is on the Free List
  - coalesce this block and the ***next*** block

`Y-16+Z+X`	`in-use=0`	`size=X`
	(Garbage, don't care)	
	`prev`	`next`
`Y-8+Z`	`in-use=0`	`size=X`
	`in-use=1`	`size=Z`
`Y`	*Return to user! Hands off!*	
`Y-8`	`in-use=1`	`size=Z`
`Y-16`	`in-use=1`	`size=W`
	Hands off!	
`Y-8-W`	`in-use=1`	`size=W`

`free()` Example

- Ex: `free(Y)` and *previous block is busy* and *next block is free*
  - i.e., `Y-16` is `1` and `Y-8+Z` is `0`
    - where `Z` is what's in `Y-4` and `X` is what's in `Y-4+Z`
  - furthermore, `Y-8+Z` is on the Free List
  - coalesce this block and the *next* block
    - just change `Y-4` and `Y-12+Z+X` to `Z+X` and `Y-8` to `0`
    - move prev and next pointers
    - adjust `next` field in previoius block in Free List
    - may need to update where Free List points

`Y-16+Z+X`	`in-use=0`	`size=Z+X`
	(Garbage, don't care)	
`Y`	`prev`	`next`
`Y-8`	`in-use=0`	`size=Z+X`
`Y-16`	`in-use=1`	`size=W`
	Hands off!	
`Y-8-W`	`in-use=1`	`size=W`

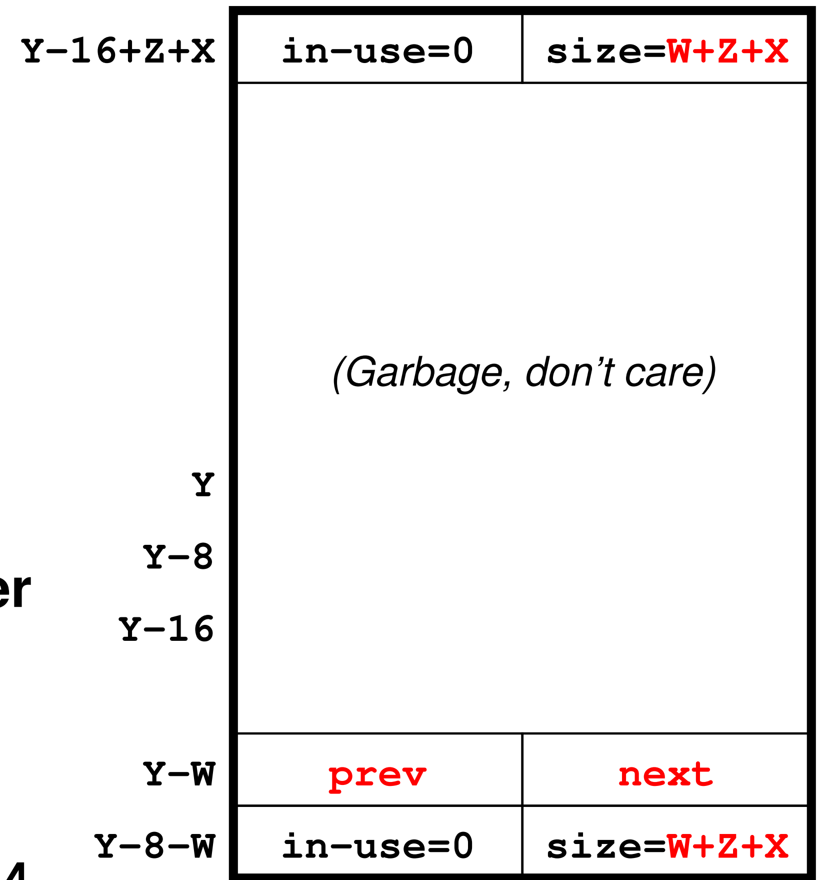
`free()` Example

- Ex: `free(Y)` and *previous block is free* and *next block is also free*
  - i.e., `Y-16` is `0` and `Y-8+Z` is `0`
    - where `Z` is what's in `Y-4`, `X` is what's in `Y-4+Z`, and `W` is what's in `Y-12`
  - blocks starting at `Y-8-W` and `Y-8+Z` are both on the Free List and next to and point at each other
  - coalesce all 3 blocks

	`in-use=0`	`size=X`
	(Garbage, don't care)	
`Y+Z`	`Y-8-W`	`next`
`Y-8+Z`	`in-use=0`	`size=X`
	`in-use=1`	`size=Z`
`Y`	*Return to user! Hands off!*	
`Y-8`	`in-use=1`	`size=Z`
`Y-16`	`in-use=0`	`size=W`
	(Garbage, don't care)	
`Y-W`	`prev`	`Y-8+Z`
`Y-8-W`	`in-use=0`	`size=W`

`free()` Example

- Ex: `free(Y)` and ***previous block is free*** and ***next block is also free***
  - i.e., `Y-16` is `0` and `Y-8+Z` is `0`
    - where `Z` is what's in `Y-4`,
      `X` is what's in `Y-4+Z`, and
      `W` is what's in `Y-12`
  - blocks starting at `Y-8-W` and `Y-8+Z` are both on the Free List and next to and point at each other
  - coalesce all 3 blocks
    - just change `Y-4-W` and `Y-12+Z+X` to `W+Z+X`
    - copy `next` from `Y+Z+4` to `Y-W+4`
    - adjust `prev` field in the new `next` block in Free List to point to `Y-8-W`
    - may need to update where Free List points



First-fit & Best-fit Algorithms

- Memory allocator must run fast
  - it does not check if the free list is in a consistent state
    - just like our warmup 1 assignment

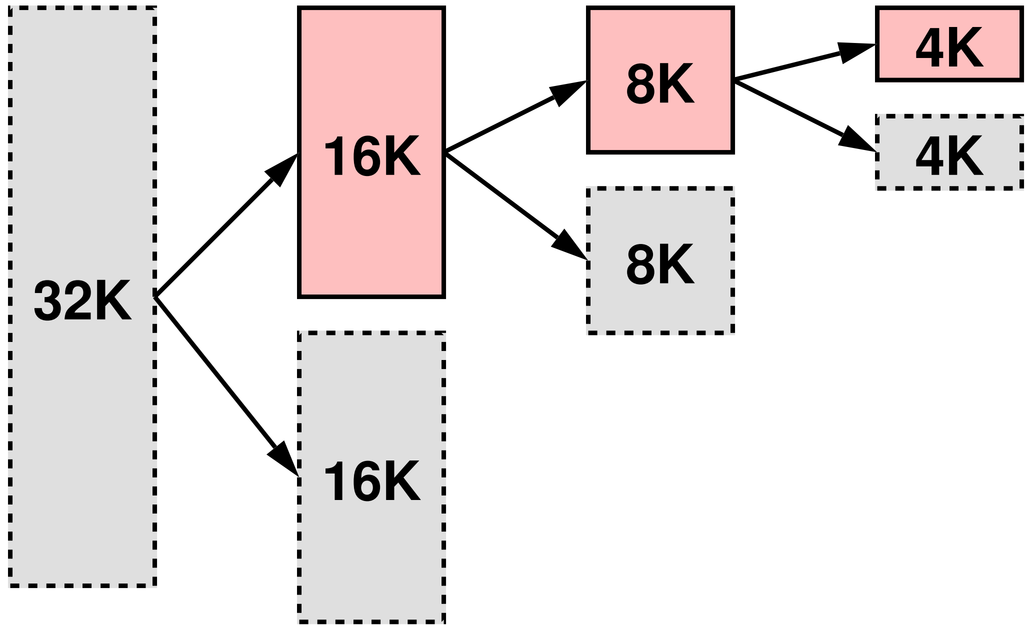
- One bad bit in the memory allocator data structure and it can break the memory allocator code
  - if you write into a *boundary tag*, your program may die in `malloc()` or `free()`
  - what would happen if you call `free()` twice on the same address?
  - user/application code can *corrupt the memory allocation chain* easily
    - the result can lead to *segmentation faults*
    - unfortunately, the corruption can *stay hidden* for a long time and *eventually* lead to a segmentation fault
      - memory corruption bugs are very difficult to squash

3.3 Dynamic Storage Allocation

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

Buddy Lists

Ex: malloc(4000)



- blocks get evenly divided into two blocks that are buddies with each other
  - can only merge with your buddy if your buddy is also free
- *internal fragmentation*
  - Ex: malloc(4000)
  - return a 4K block

Buddy Systems

- Faster memory allocation system (at the cost of more fragmentation)
  - restrict block size to be a power of 2
    1) all blocks of size 2^k start at location x where $x \bmod 2^k = 0$
    2) given a block starting at location x such that $x \bmod 2^k = 0$
       - $BUDDY_k(x) = x + 2^k$ if $x \bmod 2^{k+1} = 0$
       - $BUDDY_k(x) = x - 2^k$ if $x \bmod 2^{k+1} = 2^k$
       - Ex: $BUDDY_2(1010100) = 1010000$
    3) only buddies can be merged
    4) try to coalesce buddies when storage is deallocated
  - k different available block lists, one for each block size
  - When request a block of size 2^k and none is available:
    1) split smallest block $2^j > 2^k$ into a pair of blocks of size 2^{j-1}
    2) place block on appropriate `free` list and try again