Interrupts, Etc.

Disk I/O

```
int disk_write(…) {
  …
  startIO(); // start disk operation
  …
  enqueue(disk_waitq, CurrentThread);
  thread_switch();
    // wait for disk operation to
    // complete
  …
}
```

```
void disk_intr(…) {
  thread_t *thread;
  …
  // handle disk interrupt
  …
  thread = dequeue(disk_waitq);
  if (thread != 0) {
    enqueue(RunQueue, thread);
    // wakeup waiting thread
  }
  …
}
```


Improved Disk I/O

```
int disk_write(…) {
  …
  oldIPL = setIPL(diskIPL);
  startIO();      // start disk operation
  …
  enqueue(disk_waitq, CurrentThread);
  thread_switch();
      // wait for disk operation to complete
  setIPL(oldIPL);
  …
}
```

Modified *thread_switch*

```
void thread_switch() {
  thread_t *OldThread;
  int oldIPL;
  oldIPL = setIPL(HIGH_IPL);
    // protect access to RunQueue by masking all interrupts
  while(queue_empty(RunQueue)) {
    // repeatedly allow interrupts, then check RunQueue
    setIPL(0);  // IPL == 0 means no interrupts are masked
    setIPL(HIGH_IPL);
  }
  // We found a runnable thread
  OldThread = CurrentThread;
  CurrentThread = dequeue(RunQueue);
  swapcontext(OldThread->context, CurrentThread->context);
  setIPL(oldIPL);
}
```

Preemptive Kernels on MP

- **What's different?**
- **A thread accesses a shared data structure:**
  1. it might be *interrupted* by an interrupt handler (running on its processor) that accesses the same data structure
  2. *another thread* running on another processor might access the same data structure
  3. it might be forced to *give up its processor* to another thread, either because its time slice has expired or it has been preempted by a higher-priority thread
  4. an *interrupt handler* running on *another processor* might access the same data structure

Solution?

```
int X = 0;
SpinLock_t L = UNLOCKED;
```

```
void AccessXThread() {
  SpinLock(&L);
  X = X+1;
  SpinUnlock(&L);
}
```

```
void AccessXInterrupt() {
    …
    SpinLock(&L);
    X = X+1;
    SpinUnlock(&L);
    …
}
```

Solution …

```
int X = 0;
SpinLock_t L = UNLOCKED;
```

```
void AccessXThread() {
    MaskInterrupts();
    SpinLock(&L);
    X = X+1;
    SpinUnlock(&L);
    UnMaskInterrupts();
}
```

```
void AccessXInterrupt() {
    …
    SpinLock(&L);
    X = X+1;
    SpinUnlock(&L);
    …
}
```


Quiz 1

We have a **single-core** system with a preemptible kernel. We're concerned about data structure *X*, which is accessed by kernel threads as well as by the interrupt handler for dev.

a) It's sufficient for threads to mask dev interrupts while accessing *X*
b) In addition, threads must lock (blocking) mutexes before masking interrupts and accessing *X*
c) b doesn't work. Instead, threads must lock spinlocks before accessing *X*
d) In addition to c, the dev interrupt handler must lock a spinlock before accessing *X*
e) Something else is needed

Deferred Work

- Interrupt handlers run with interrupts masked
  - may interfere with handling of other interrupts, particularly if they do a lot of computation
- Solution
  - do minimal work now
  - do rest later without interrupts masked

Deferred Processing

```
void TopLevelInterruptHandler(int dev) {
  InterruptVector[dev](); // call appropriate handler
  if (PreviousContext == ThreadContext) {
    UnMaskInterrupts();
    while(!Empty(WorkQueue)) {
      Work = DeQueue(WorkQueue);
      Work();
    }
  }
}

void NetworkInterruptHandler() {
  // deal with interrupt
  …
  EnQueue(WorkQueue, MoreWork);
}
```

Windows Interrupt Priority Levels

hardware	31	High
	30	Power fail
	29	Inter-processor
	28	Clock
	. . .	
	4	Device 2
	3	Device 1
software	2	DPC
	1	APC
	0	Thread

Deferred Procedure Calls

```
void InterruptHandler( ) {
  // deal with interrupt
  …
  QueueDPC(MoreWork, arg);
  /* enqueues MoreWork on
     the DPC queue and
     requests a DPC
     interrupt
  */
}
```

```
void DPCHandler( … ) {
  while(!Empty(DPCQueue)) {
    Work = DeQueue(DPCQueue);
    Work();
  }
}
```


Software Interrupt Threads

```
void InterruptHandler() {
    // deal with interrupt
    …
    EnQueue(WorkQueue,
          MoreWork);
    SetEvent(Work);
}
```

```
void SoftwareInterruptThread() {
    while(TRUE) {
        WaitEvent(Work)
        while(!Empty(WorkQueue)) {
            Work = DeQueue(
                WorkQueue);
            Work();
        }
    }
}
```


Preemption: User-Level Only

```
void ClockHandler() {
  // deal with clock interrupt
  …
  if (TimeSliceOver())
    ShouldReschedule = 1;
}
```

```
void TopLevelInterruptHandler(int dev) {
  InterruptVector[dev]();
  if (PreviousMode == UserMode) {
    // the clock interrupted user-mode code
      if (ShouldReschedule)
        Reschedule();
  }
  …
}

void TopLevelTrapHandler(…) {
  SpecificTrapHandler();
  …
  if (ShouldReschedule) {
    /* the time slice expired while the thread
       was in kernel mode */
    Reschedule();
  }
}
```

Preemption: Full

```
void ClockInterruptHandler( ) {
  // deal with clock interrupt
  …
  if (TimeSliceOver)
    QueueDPC(Reschedule);
}
```

Directed Processing

- **Signals: Unix**
  - perform given action in context of a particular thread in user mode
- **APC: Windows asynchronous procedure calls**
  - roughly same thing, but also may be done in kernel mode

Asynchronous Procedure Calls

- Two uses
  - kernel APC: release of kernel resources
  - user APC: notifying a thread of an external event

Kernel APC

- Release of kernel resources
  - interrupt handler has information that must be copied to user process
  - can’t be done unless in context of process
    - otherwise address space not mapped in
  - interrupt handler requests kernel APC to have user thread, running in kernel mode, copy information to user space, and then free data in the kernel

User APC

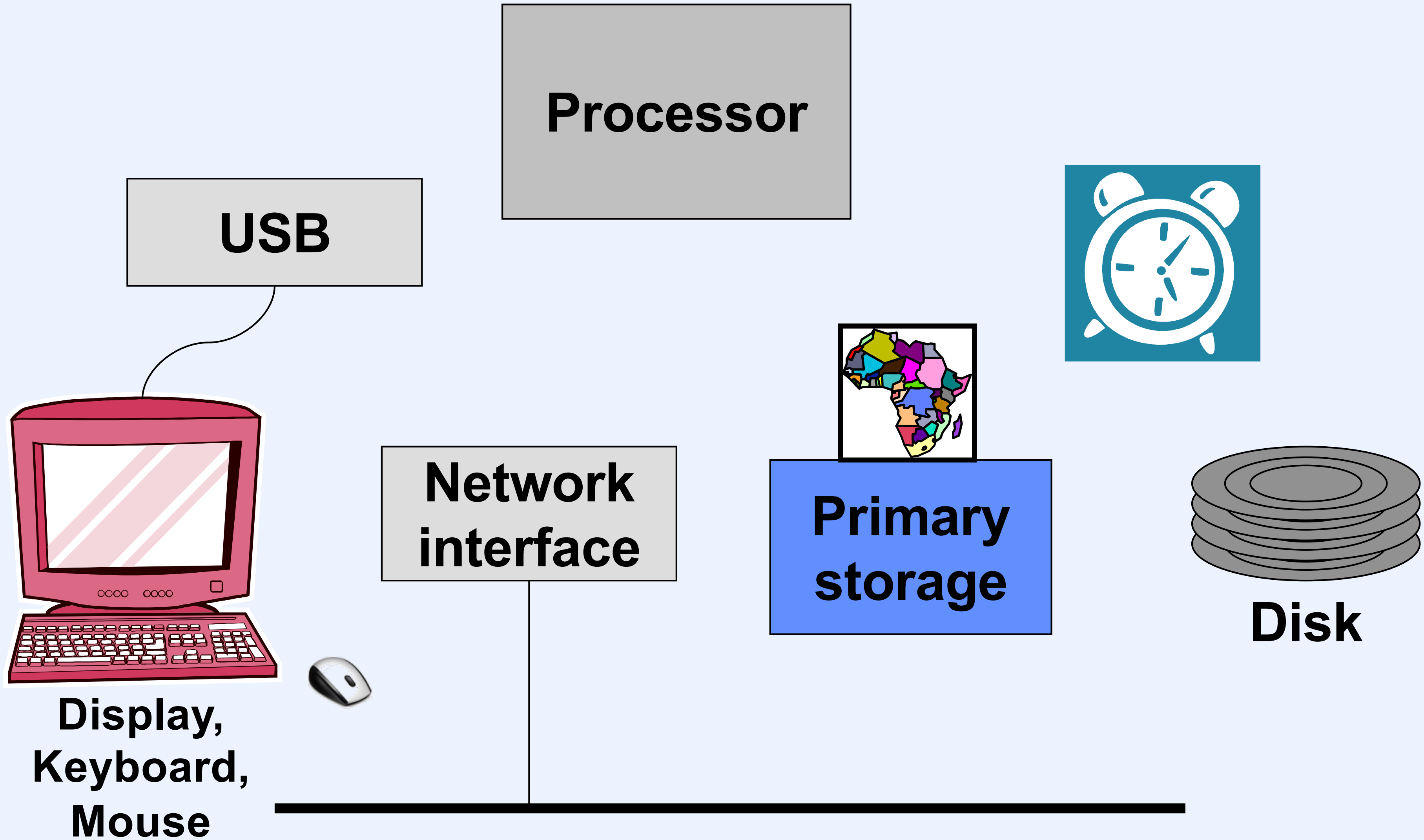
- **Notifying thread of external event**
  - **example: asynchronous I/O**
    - **thread supplies *completion routine* when starting asynchronous I/O request**
    - **called in thread's context when I/O completes**
      - **similar to a Unix signal**
      - **called only when thread is in *alertable wait state***
        - **an option in certain blocking system calls**

APC Implementation

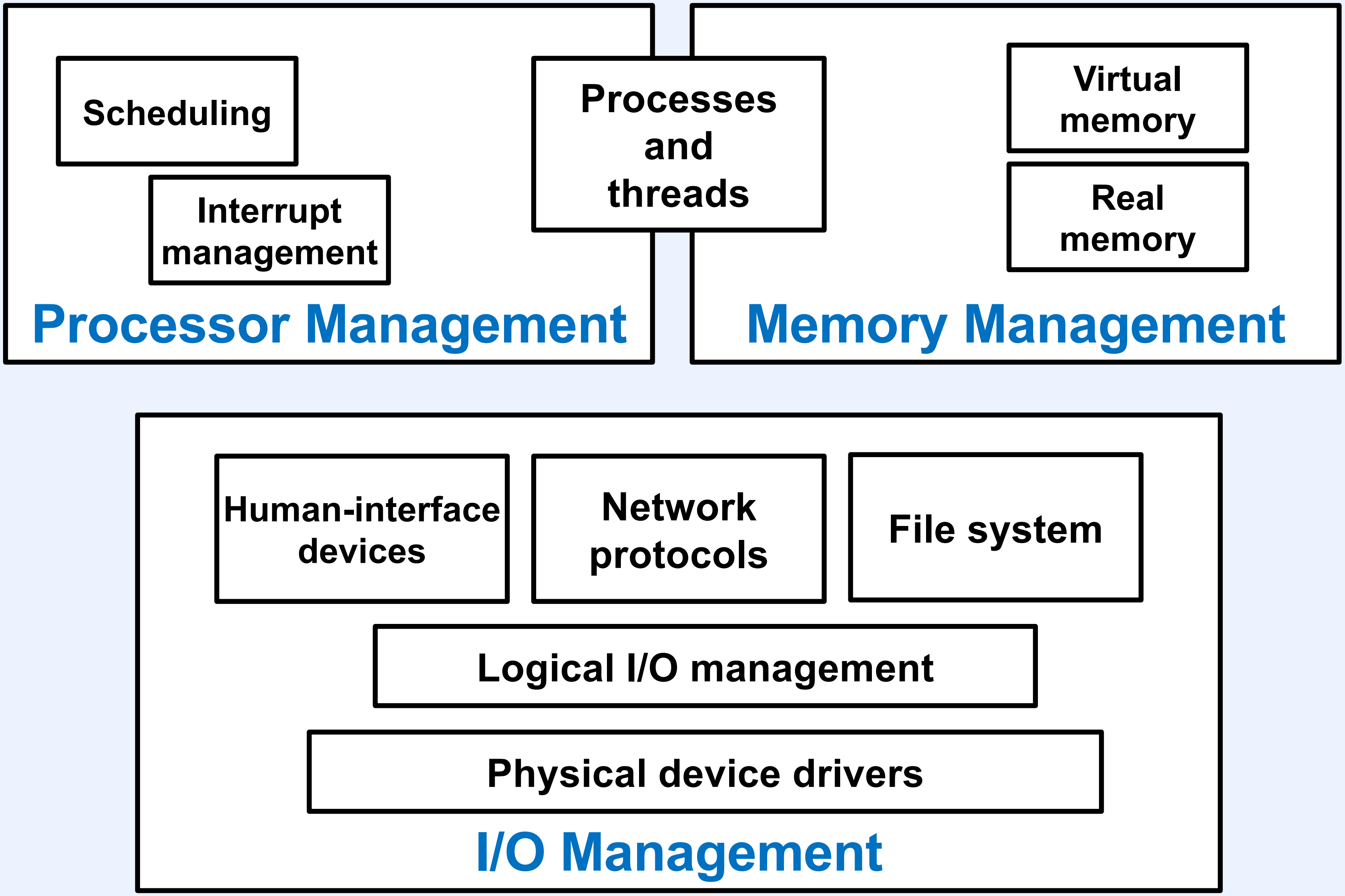
- Per-thread list of pending APCs
  - on notification, thread executes them
- User APC
  - thread in alertable state is woken up and executes pending APCs when it returns to user mode
- Kernel APC
  - running thread interrupted by APC interrupt (lowest-priority interrupt)
  - waiting thread is "unwaited"
  - execute pending kernel APCs

I/O

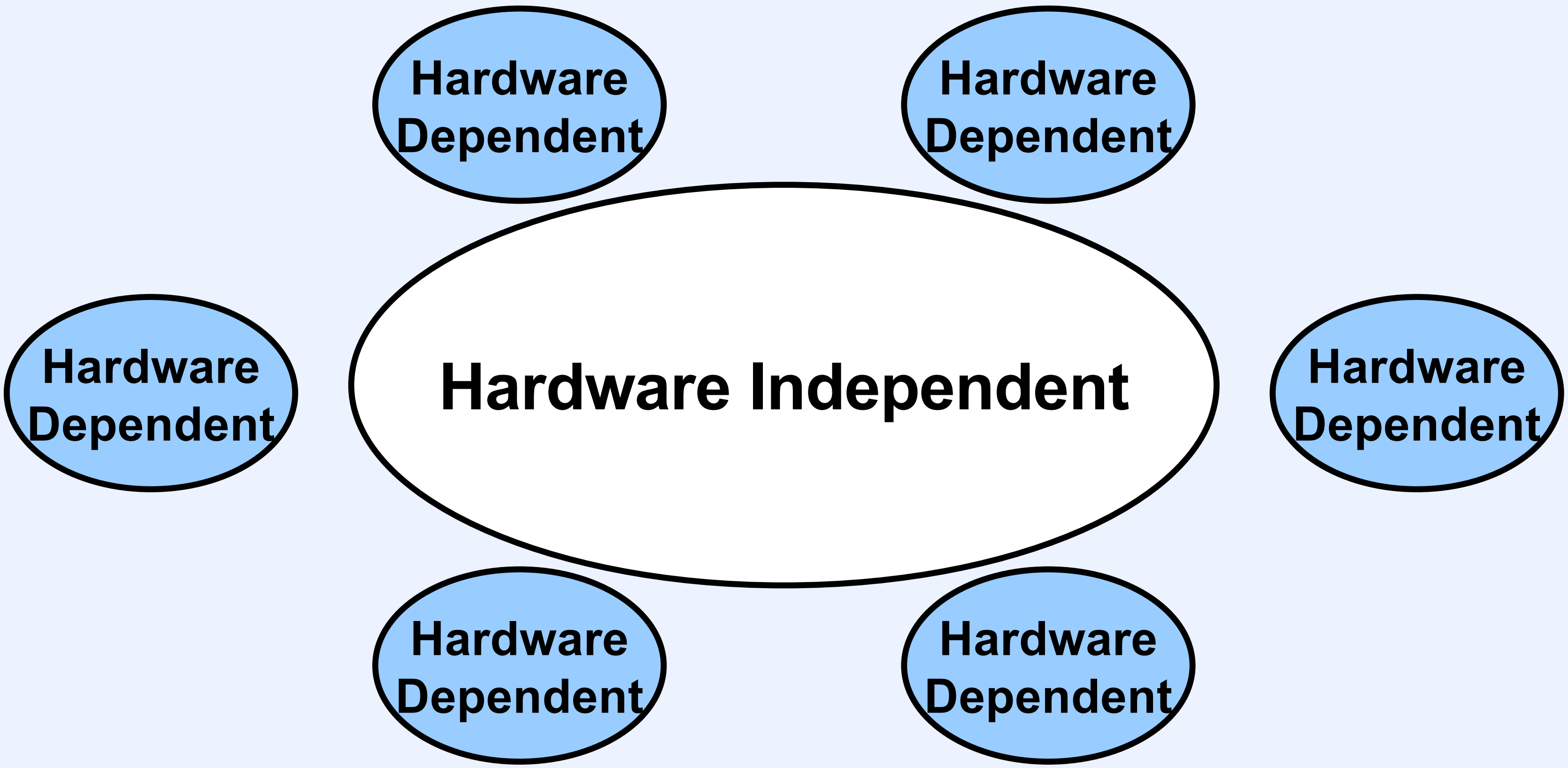
Simple Configuration



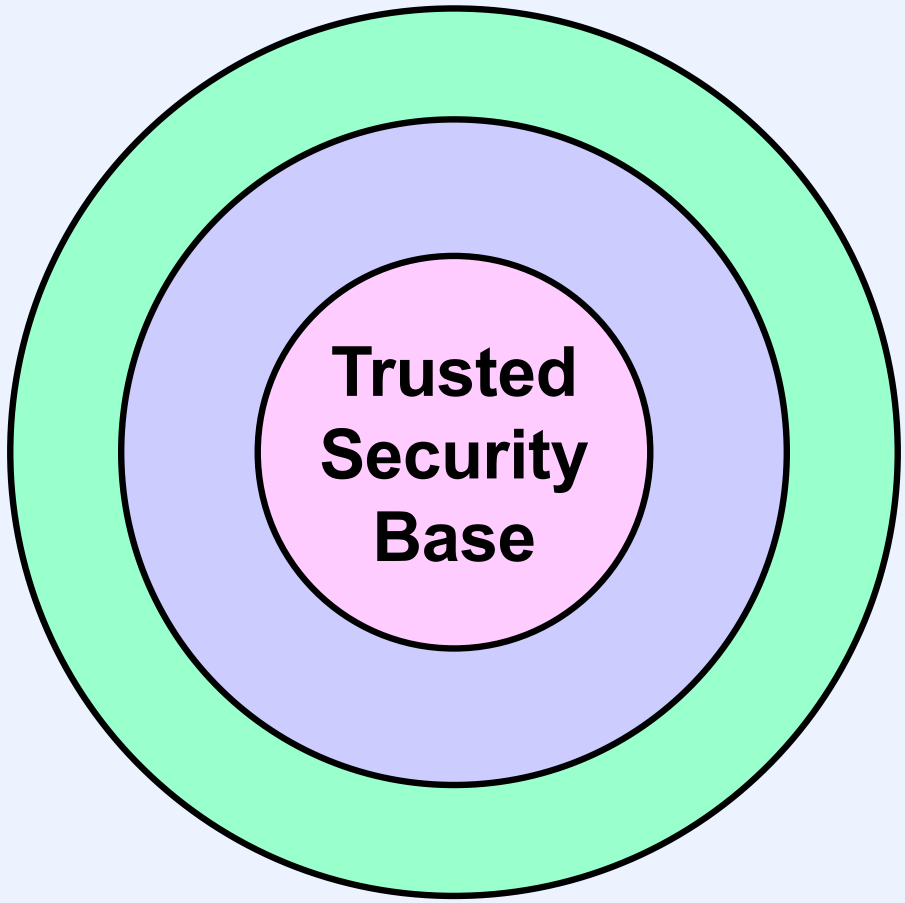
OS Components: Functional



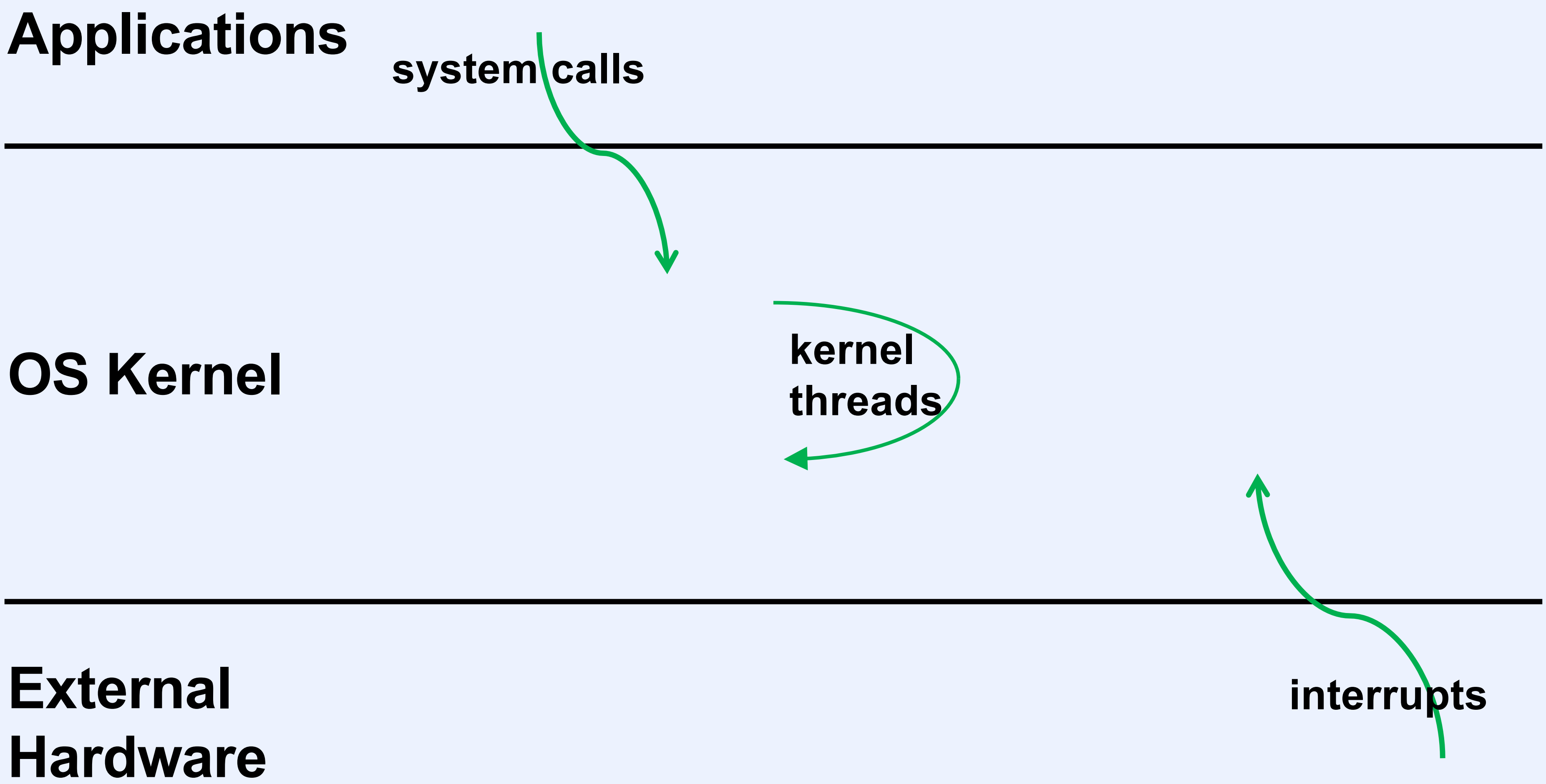
OS Components: Portability



OS Components: Importance



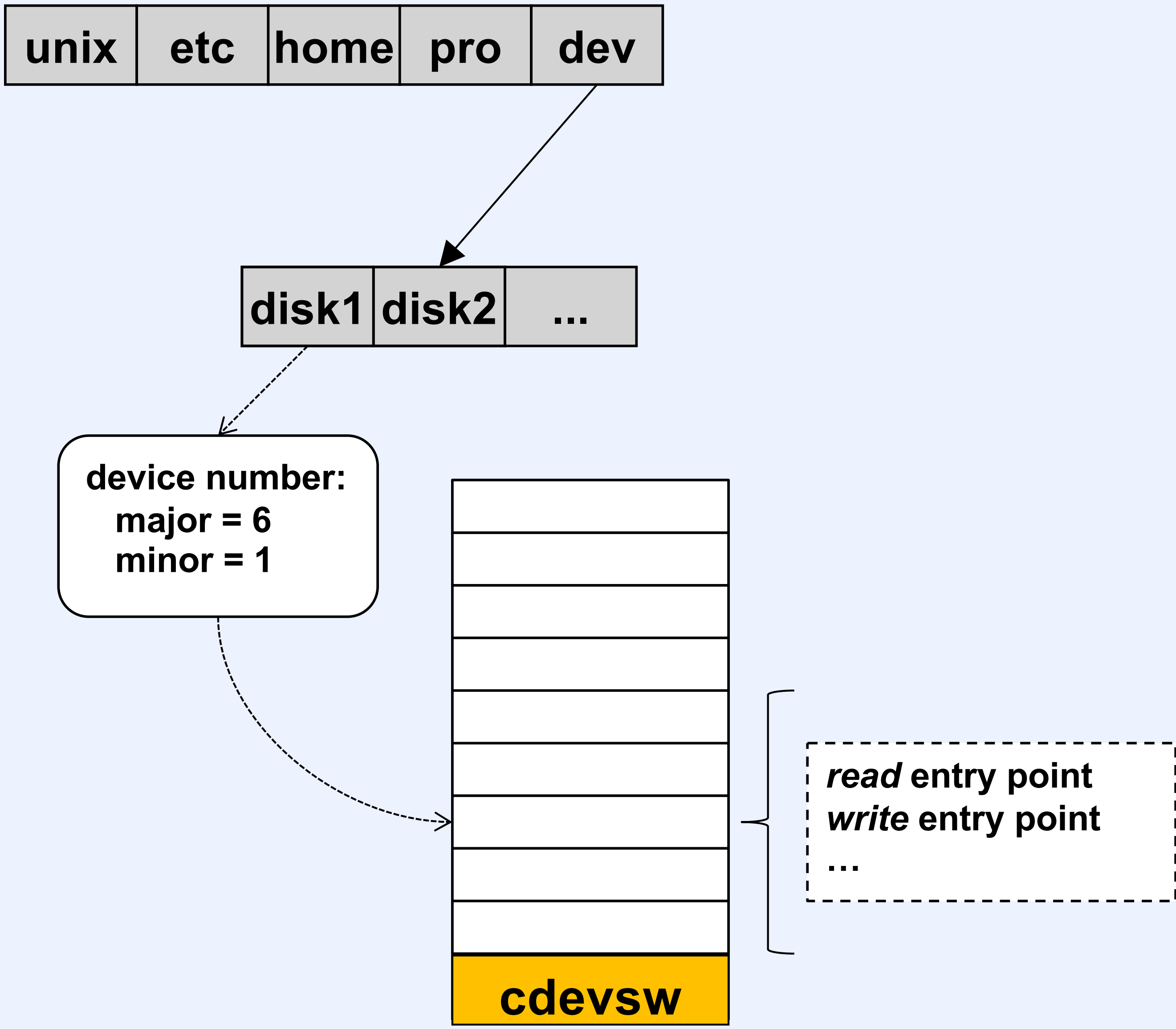
OS Components: Flow of Control



To Be Discussed

- What is the functionality of the components?
- What are the key data structures?
- How is the system broken up into modules?
- To what extent is the system extensible?
- What parts run in the OS kernel in privileged mode? What parts run as library code in user applications? What parts run as separate applications?
- In which execution contexts do the various activities take place?

Finding Devices



Discovering Devices

- You plug in a new device to your computer …
  – OS must notice
    - must find a device driver
      - what kind of device is it?
      - where is the driver?
    - must assign a name
      - how chosen?
    - multiple similar devices
      - how does application choose?

Computer Terminal

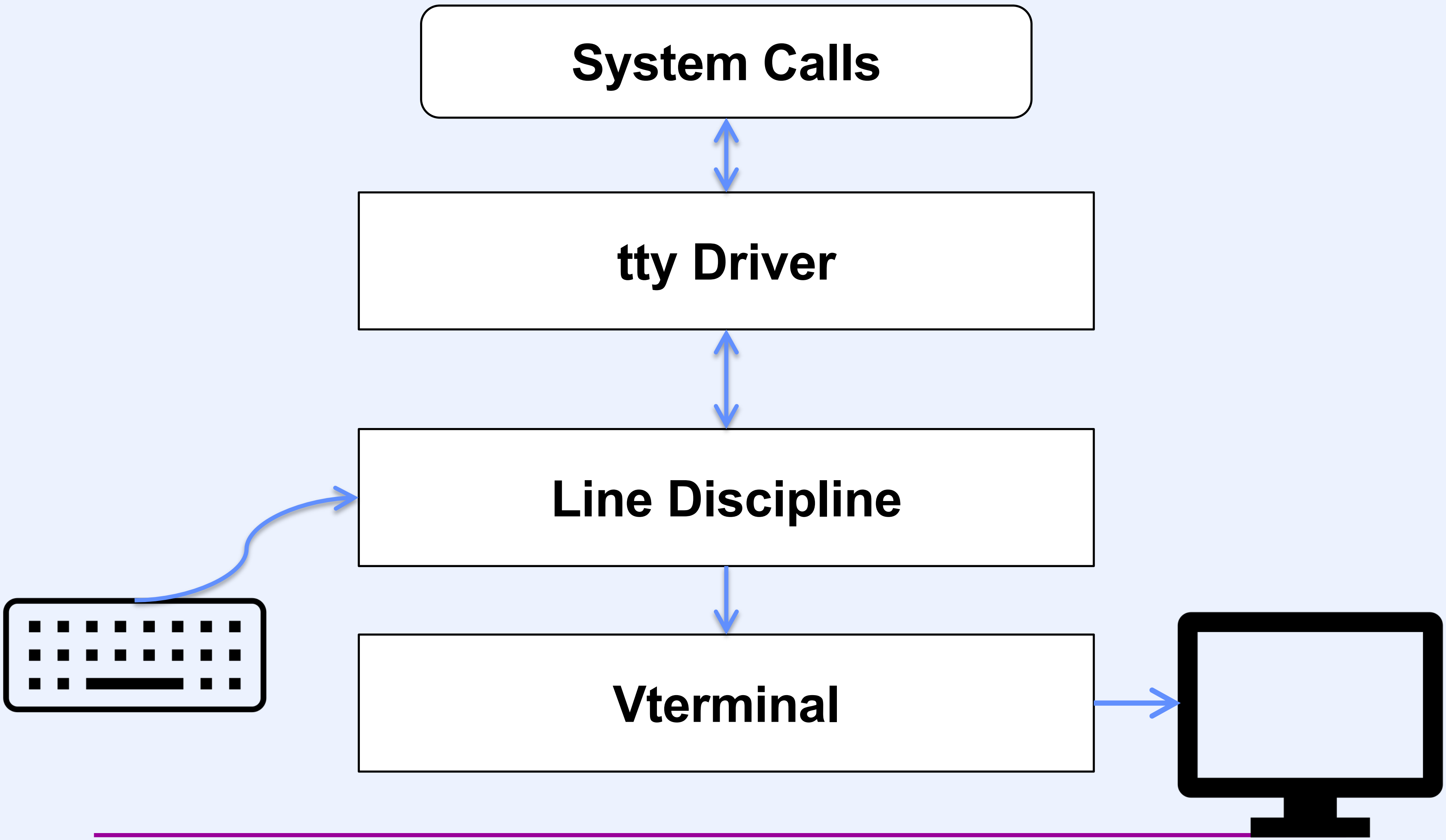
A “tty”

A Typewriter

Terminals

- Long obsolete, but still relevant
- Issues
  1) characters are generated by the application faster than they can be sent to the terminal
  2) characters arrive from the keyboard even though there isn't a waiting read request from an application
  3) input characters may need to be processed in some way before they reach the application

Terminals



Quiz 2

In which context are characters transformed from raw into cooked?

a) In the interrupt context (i.e., on a "borrowed" stack)

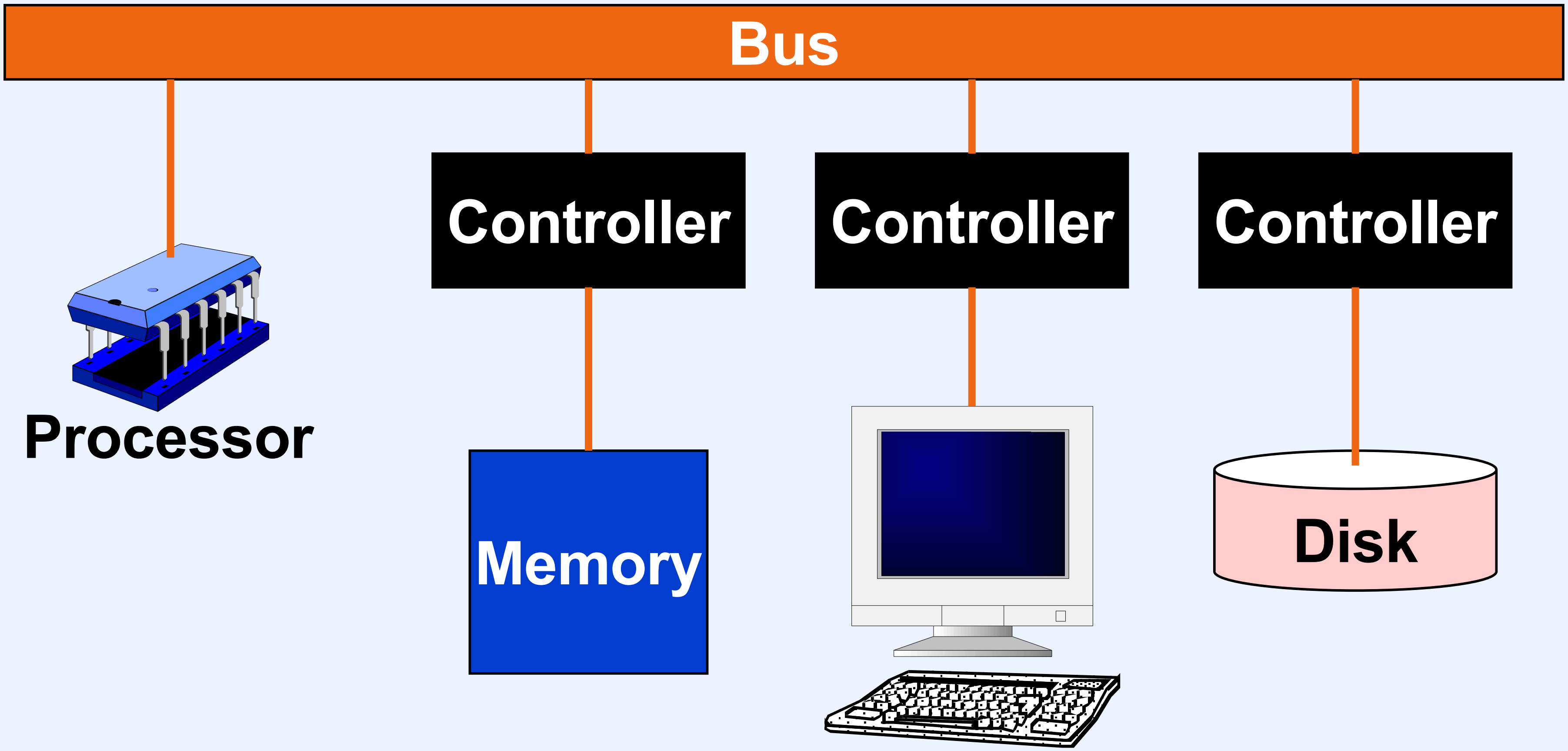
b) In the context of the thread performing the *read* system call

c) Some other context

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

Simple I/O Architecture



PIO Registers

GoR	**GoW**	**IER**	**IEW**					**Control register**
RdyR	**RdyW**							**Status register**
								Read register
								Write register

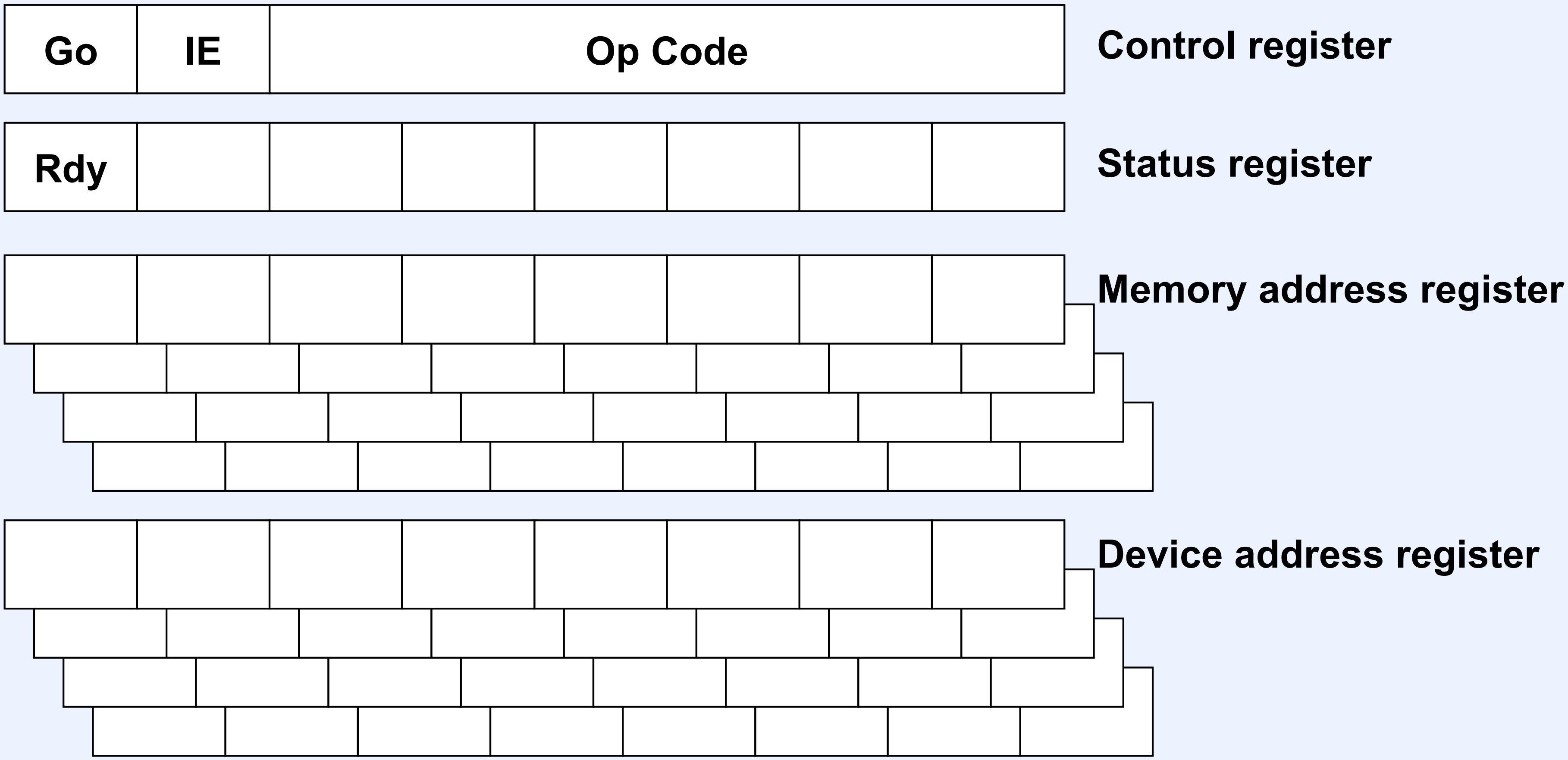
Legend:

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 WGO bit in the *control register*
  - wait for WREADY bit (in *status register*) to be set (if interrupts have been enabled, an interrupt occurs when this happens)

DMA Registers



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 interrupt ENABLE bit in the *control register*
  - wait for interrupt or for READY bit to be set

Device Drivers