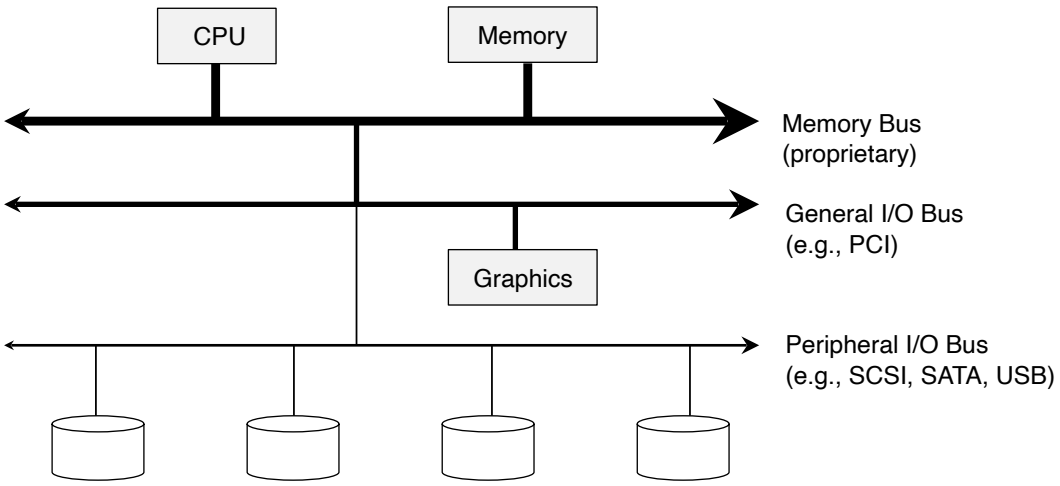
Persistence

- 36 - I/O Devices
- 37 - Hard Disk Drives
- 38 - RAID
- 39 - File and Directories

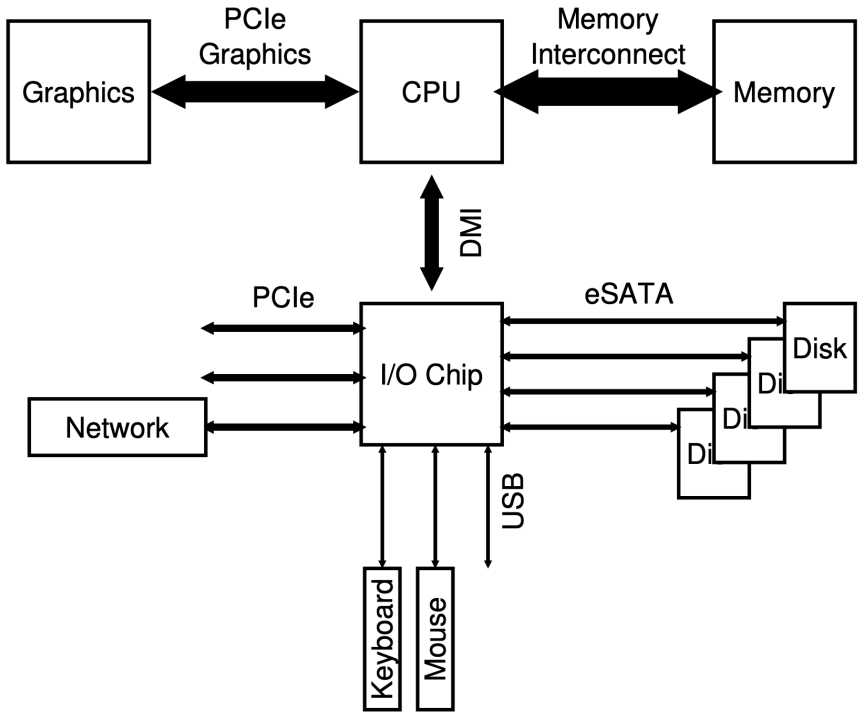
Classic I/O Architecture



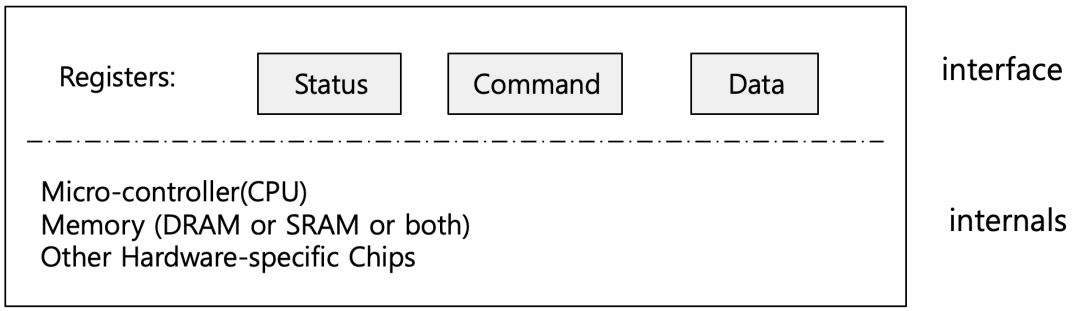
Prototypical System Architecture

- How should I/O devices be integrated into systems?
- What are the general mechanisms?
- How can we make them efficient?

Modern Architecture



A Canonical Device



- status register: read current device state
- command register: send commands to device
- data register: read or write data a word at a time

Interaction *w/ the canonical device*

```
While (STATUS == BUSY)
    ; // wait until device is not busy
Write data to DATA register
Write command to COMMAND register
    (starts the device and executes the command)
While (STATUS == BUSY)
    ; // wait until device is done with your request
```


Interaction *polling*

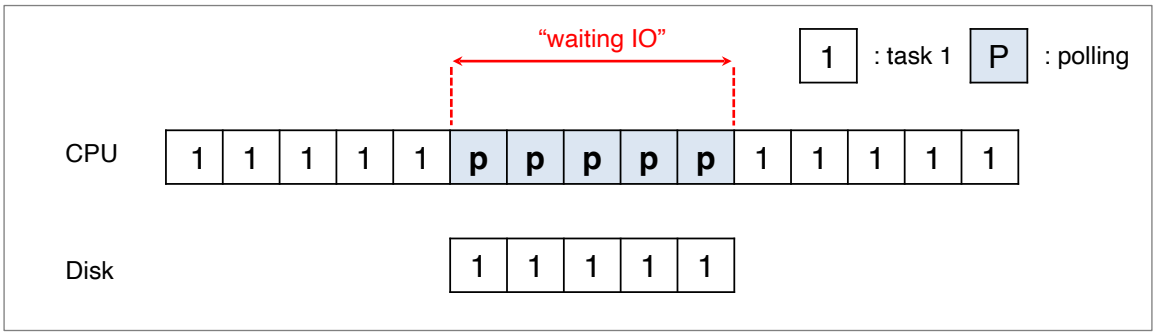
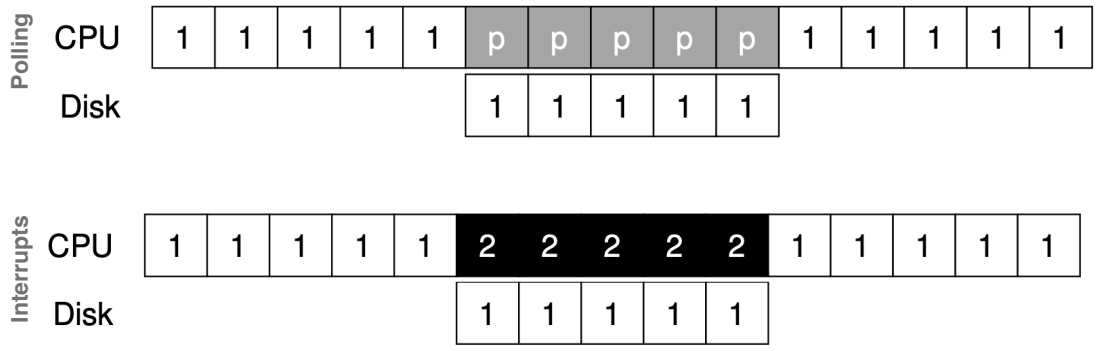


Diagram of CPU utilization by polling

- Polling:
  - repeatedly reading status register to determine readiness
  - simple
  - inefficient:
    - CPU occupied doing nothing
    - switching to another ready process may be better

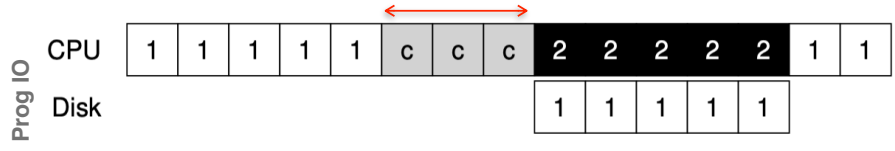
Devices: interrupts

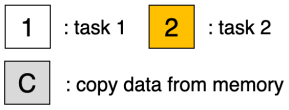


- send request
- block requesting process
- reschedule process only when interrupt signals completion

Efficiency Issues *with interrupts*

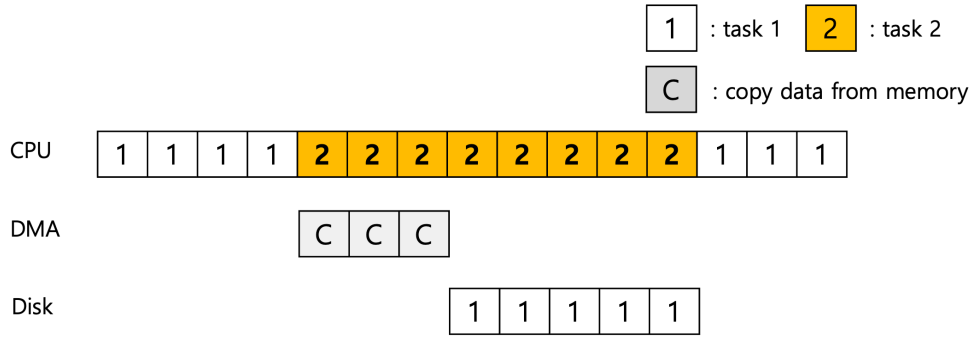
- Fast jobs: first poll might have found job finished
  - Hybrid: poll for a bit, then block
- Livelock: per-packet interrupts might monopolize CPU
  - Coalescing: device delays to combine multiple interrupts
- Writing large blocks to device is a poor use of CPU:





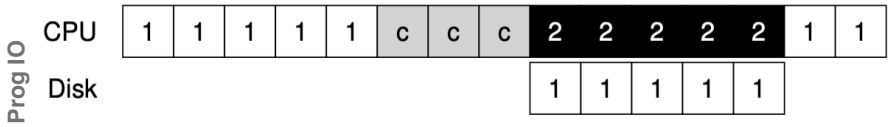
DMA *direct memory access*

- Starting:
  - write address, length of data block to device data registers
  - start by writing to control register
  - *do something else*
- Finish:
  - raise interrupt to signal finish

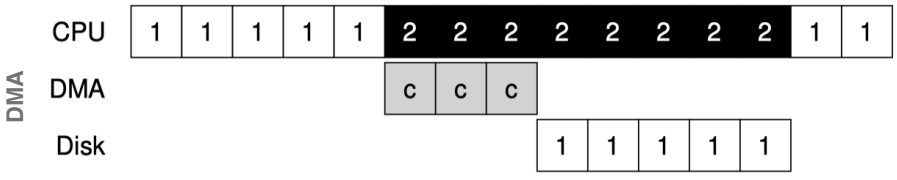


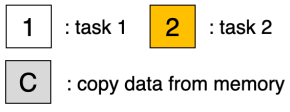
DMA vs Programmed I/O

- Programmed I/O:



- Direct Memory Access (DMA)

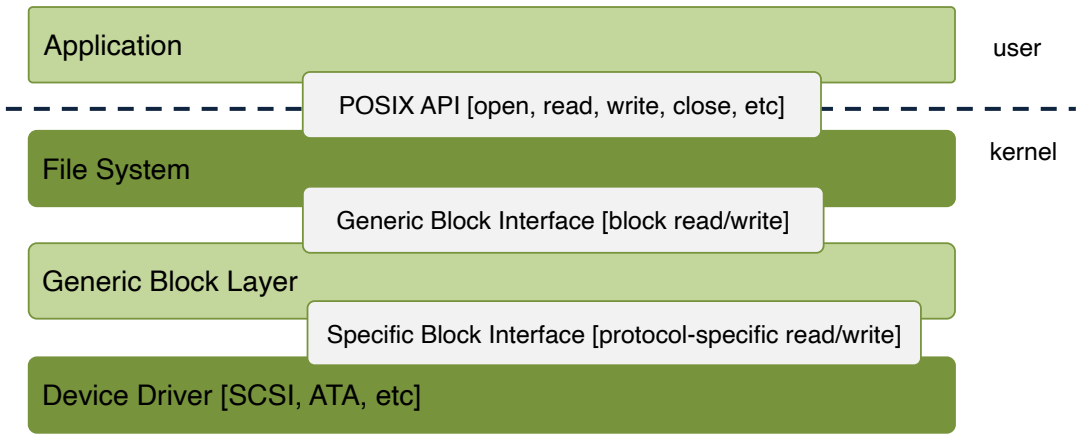




Device Interaction

- How does the OS communicate w/ the device?
  - *I/O instructions*: special instructions
    - example: `in` and `out` instructions on x86
  - *memory-mapped I/O*:
    - device registers mapped to physical addresses
    - use generic OS `load` and `store` instructions for commands and data transfer
- But…but… there are many different devices!
  - Encapsulate device-specific interactions in *generic block interface*

File system abstraction



The File System Stack

Issues *remaining*

- Devices w/ special capabilities
  - might not be able to use w/ generic layer
- Bugs!
  - device drivers are specific to hardware, written by companies that build the hardware
  - over 70% of linux source is in device drivers
  - primary source of bugs and kernel crashes

Example Device: IDE interface

```
Control Register:
  Address 0x3F6 = 0x08 (0000 1RE0): R=reset,
                  E=0 means "enable interrupt"

Command Block Registers:
  Address 0x1F0 = Data Port
  Address 0x1F1 = Error
  Address 0x1F2 = Sector Count
  Address 0x1F3 = LBA low byte
  Address 0x1F4 = LBA mid byte
  Address 0x1F5 = LBA hi  byte
  Address 0x1F6 = 1B1D TOP4LBA: B=LBA, D=drive
  Address 0x1F7 = Command/status

Status Register (Address 0x1F7):
    7      6     5      4     3     2     1     0
   BUSY  READY FAULT SEEK  DRQ  CORR IDDEX ERROR

Error Register (Address 0x1F1): (check when ERROR==1)
    7      6     5      4     3     2     1     0
   BBK    UNC   MC   IDNF  MCR  ABRT T0NF AMNF

   BBK  = Bad Block
   UNC  = Uncorrectable data error
   MC   = Media Changed
   IDNF = ID mark Not Found
   MCR  = Media Change Requested
   ABRT = Command aborted
   T0NF = Track 0 Not Found
   AMNF = Address Mark Not Found
```

LBA = “logical block address”

I/O *outline*

- Wait for drive to be ready:
  - read Status Register (0x1F7) until drive is not busy, and READY
- Write parameters to command registers:
  - Write the sector count, logical block address (LBA) of the sectors to be accessed, and drive number (master=0x00 or slave=0x10, as IDE permits just two drives) to command registers (0x1F2-0x1F6)
- Start the I/O:
  - issue read/write to command register (0x1F7)
- Data transfer (for writes):
  - wait until drive status has READY and DRQ (drive request for data)
  - write data to data port
- Handle interrupts:
  - In the simplest case, handle an interrupt for each sector transferred; more complex approaches allow batching and thus one final interrupt when the entire transfer is complete.
- Error handling:
  - read the status register after each operation. If ERROR bit is on, read the error register for details

Example IDE Driver

- determining readiness:

```
static int ide_wait_ready() {
        while (((int r = inb(0x1f7)) & IDE_BSY) || !(r & IDE_DRDY))
        ; // loop until drive isn't busy
}
```

Example IDE Driver

- queueing an I/O request:

```
void ide_rw(struct buf *b) {
  acquire(&ide_lock);
  for (struct buf **pp = &ide_queue; *pp; pp=&(*pp)->qnext)
    ;                                 // walk queue
  *pp = b;                            // add request to end
  if (ide_queue == b)                 // if q is empty
    ide_start_request(b);             // send req to disk
  while ((b->flags & (B_VALID|B_DIRTY)) != B_VALID)
    sleep(b, &ide_lock);              // wait for completion
  release(&ide_lock);
}
```

- write parameters to command register, start the I/O:

```
static void ide_start_request(struct buf *b) {
  ide_wait_ready();
  outb(0x3f6, 0);                  // generate interrupt
  outb(0x1f2, 1);                  // how many sectors?
  outb(0x1f3, b->sector & 0xff);   // LBA goes here ...
  outb(0x1f4, (b->sector >> 8) & 0xff);   // ... and here
  outb(0x1f5, (b->sector >> 16) & 0xff);  // ... and here!
  outb(0x1f6, 0xe0 | ((b->dev&1)<<4) | ((b->sector>>24)&0x0f));
  if(b->flags & B_DIRTY){
    outb(0x1f7, IDE_CMD_WRITE);    // this is a WRITE
    outsl(0x1f0, b->data, 512/4);  // transfer data too!
  } else {
    outb(0x1f7, IDE_CMD_READ);     // this is a READ (no data)
  }
}
```


Example IDE Driver

- handling completion interrupt:

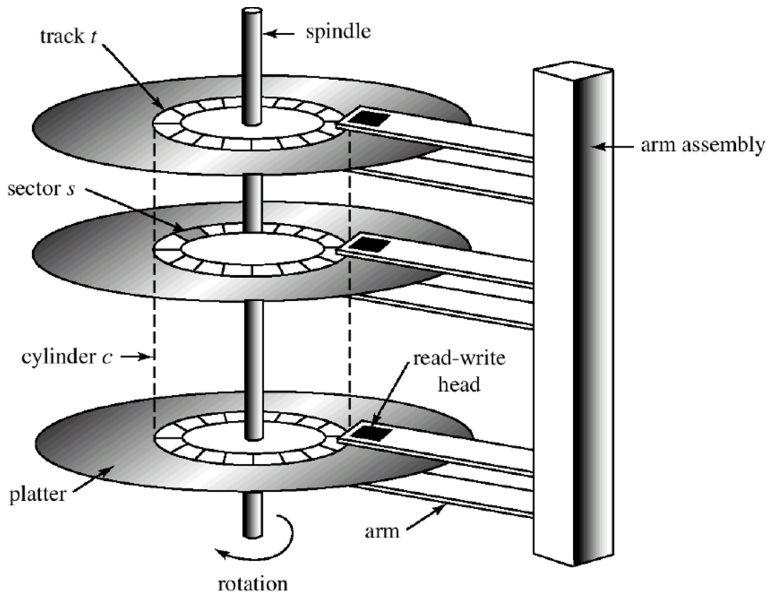
```
void ide_intr() {
  struct buf *b;
  acquire(&ide_lock);
  if (!(b->flags & B_DIRTY) && ide_wait_ready(1) >= 0)
    insl(0x1f0, b->data, 512/4); // if READ: get data
  b->flags |= B_VALID;
  b->flags &= ~B_DIRTY;
  wakeup(b); // wake waiting process
  if ((ide_queue = b->qnext) != 0) // start next request
    ide_start_request(ide_queue); // (if one exists)
  release(&ide_lock);
}
```

Persistence

- 36 - I/O Devices
- 37 - Hard Disk Drives
- 38 - RAID
- 39 - File and Directories

Magnetic Hard Drives

- *platter* has set of concentric tracks
- each track divided into sectors
- sectors read by read-write head



Computing the Cost

- Cost is:
  - + seek time: move to correct track
  - + rotational delay: disk must rotate until we get to correct sector
  - + transfer time: time to read a sector
- Also, disk has:
  - track cache: head always reading, remembering
  - scheduler: more later…