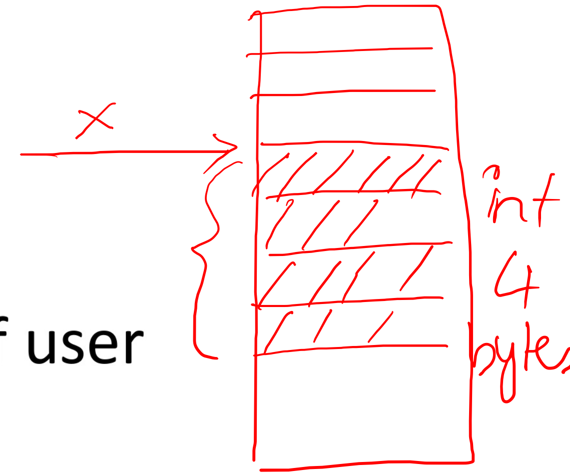
Design and Engineering of Computer Systems

Lecture 4: Overview of memory and I/O hardware

Mythili Vutukuru

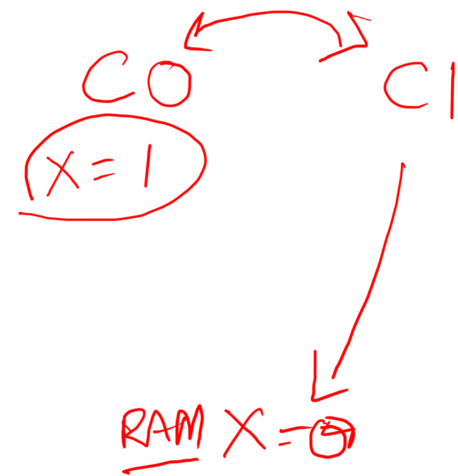
IIT Bombay

Main memory

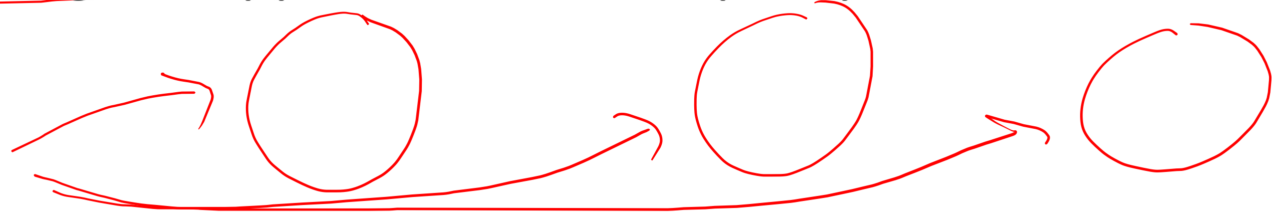


- Random Access Memory (RAM) stores instructions and data of user programs
- Byte addressable: data stored in memory can be accessed via memory address / location / byte number
  - Every instruction/variable can be stored/retrieved using its memory address
  - Instructions/variables can occupy multiple contiguous bytes in memory
- Accessed at granularity of multiple bytes at a time (say, 4 byte words)
- Symmetric multiprocessor systems (SMP): multiple CPU cores share the same main memory
- Non uniform memory access (NUMA): some CPU cores are closer to some part of the memory

Memory coherence



- Ideally, memory reads should return the latest value written
- Difficult to guarantee memory coherence in multicore systems
  - A CPU core may have modified data at a memory address in its private cache
  - Cache coherence protocols required to ensure coherent view of shared memory systems, impose overhead
  - Other optimizations in modern hardware make memory consistency harder too
- Cannot have a very large number of CPU cores sharing the same main memory coherently
  - Beyond a point, distributing the application to multiple systems makes sense



Memory allocation

- When is memory allocated for code/data in a program?
- Memory allocated in the compiled executable for:
  - Code (instructions) in the program
  - Global/static variables
- Should we allocate memory for local variables, arguments of functions in executable?
  - No, since we do not now if/how many times the function will be called at runtime
- Similarly, malloc is for dynamic memory allocation at runtime, not compile time
- When program is run, some memory reserved in process memory image for such dynamic, runtime allocations

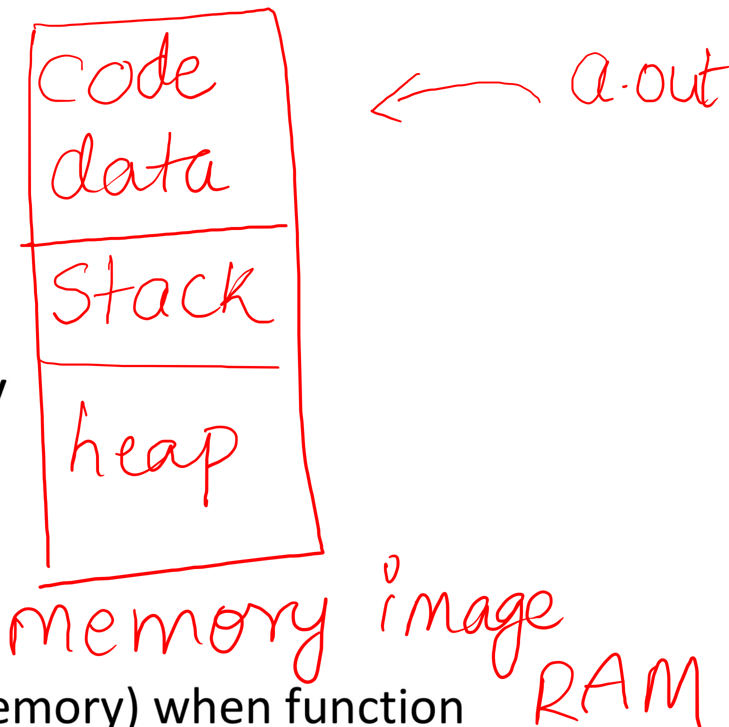
a.out

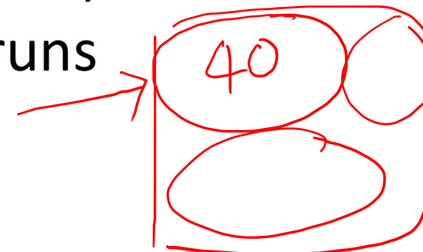
g

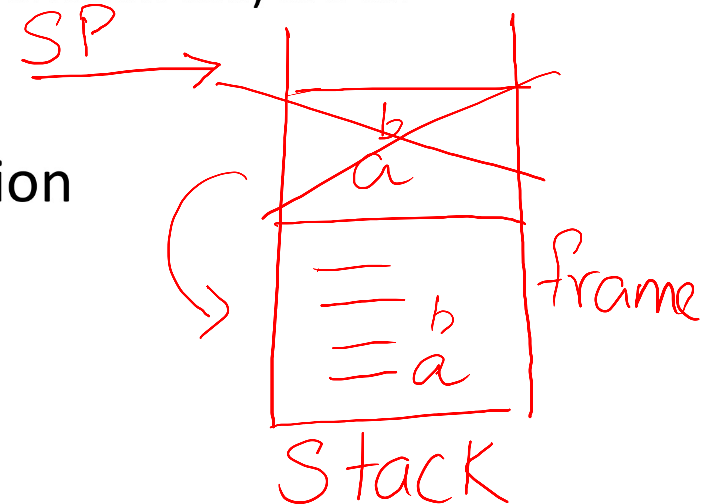
```
int g;

int increment(int a) {
    int b;
    b = a+1;
    return b;
}

main() {
    int x, y;
    x = 1;
    y = increment(x);

    int *z = malloc(40);

}
```

Memory image of a process

- Memory image of a process: code+data of process in memory
  - Code: CPU instructions in the program
  - Compile-time data: global/static variables allocated at compile time
  - Stack and heap for dynamic memory allocation at runtime
- Stack: memory allocation during function call
  - Push data (allocate memory) on stack for a function call, pop (free memory) when function returns
  - Memory for function arguments, local variables, return address (instruction to return to after completing function), register context (to continue computation after function call) are all part of function's stack frame
- Current position on stack stored in stack pointer CPU register
- Heap: dynamic memory allocation by malloc and related function
  - Malloc returns address on heap of allocated memory chunk
- Heap and stack can grow/shrink as process runs

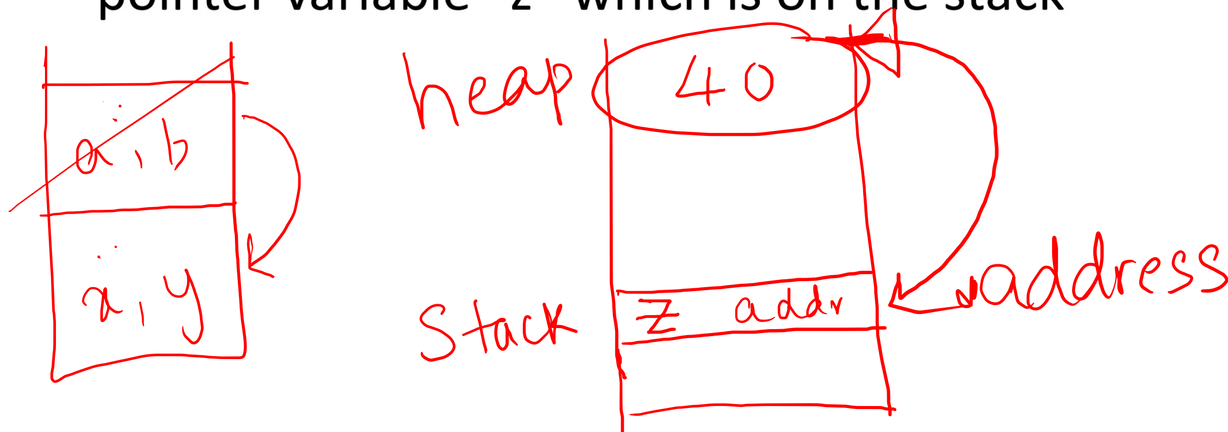






Example

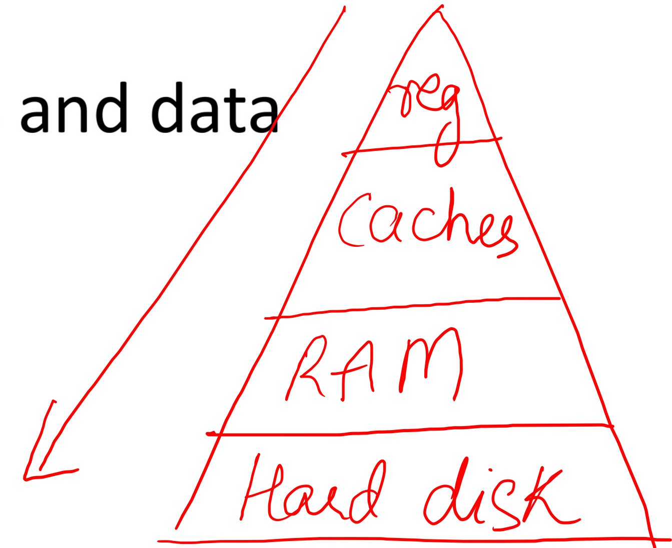
- Variable "g" allocated at compile time (data)
- Main is also considered function, so variables "x", "y" are allocated on stack when program starts
- When function "increment" is called, variables "a", "b" are allocated on stack
- When malloc is called, 40 bytes allocated on heap
  - Memory address of 40 bytes on heap is stored in pointer variable "z" which is on the stack



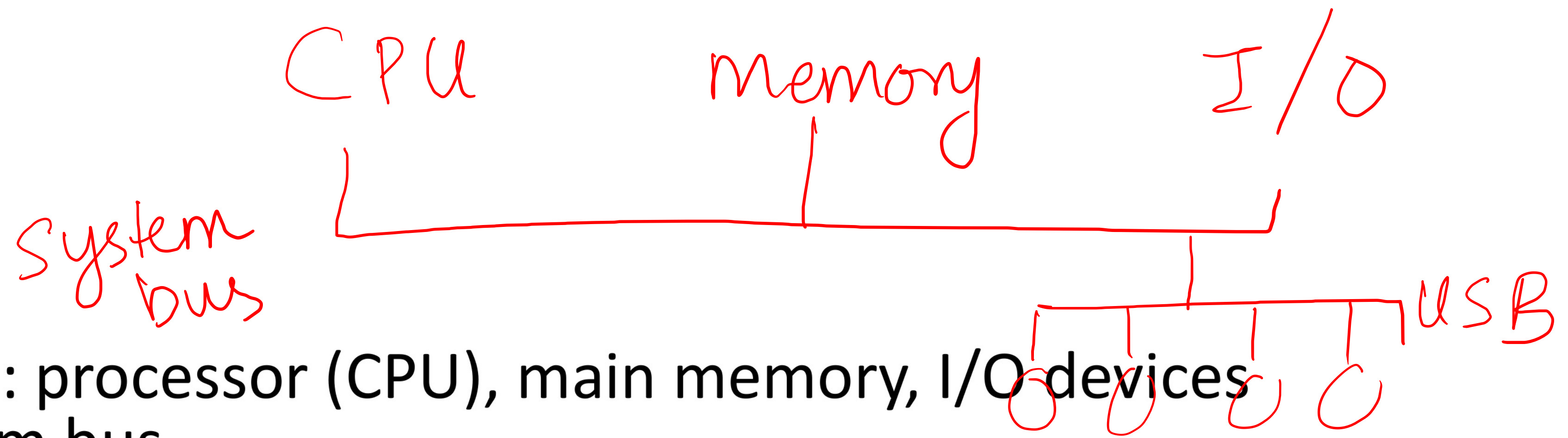

```
int g;

int increment(int a) {
    int b;
    b = a+1;
    return b;
}

main() {
    int x, y;
    x = 1;
    y = increment(x);
    int *z = malloc(40);
}
```

Memory hierarchy

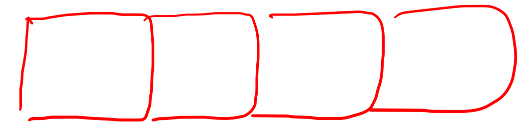
- Hierarchy of storage elements which store instructions and data
  - CPU registers (small number, <1 nanosec)
  - CPU caches (few MB, 1-10 nanosec)
  - Main memory (few GB, ~100 nanosec)
  - Hard disk (few TB, ~1 millisec)
- Hard disk is non-volatile storage, rest are volatile
  - Hard disk stores files and other data persistently
- As you go down the hierarchy, memory access technology becomes cheaper, slower, less expensive



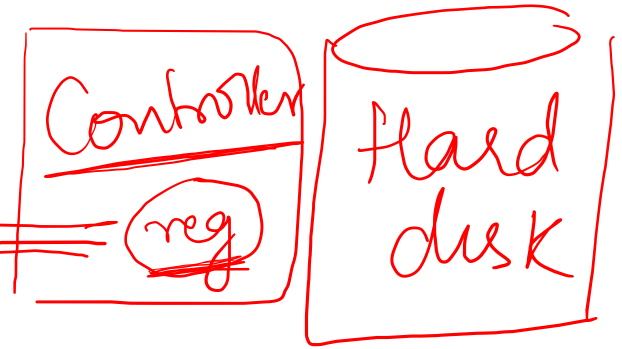
I/O devices



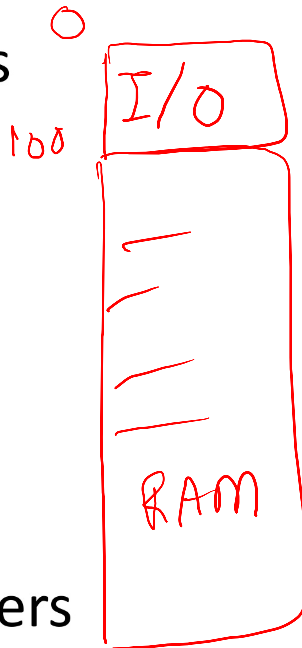
- A computer system: processor (CPU), main memory, I/O devices connected by system bus
    - Some I/O devices also on special I/O buses (like USB)
- Bus: a set of wires carrying data between components
    - Components address each other, coordinate access of bus via bus arbitration protocols
- Examples of I/O devices
    - Hard disk: stores information in blocks persistently
    - Network Interface Card (NIC): streams information from external machines
    - Keyboard, mouse: input stream from user
    - Display monitor: stream output to user



Device controller

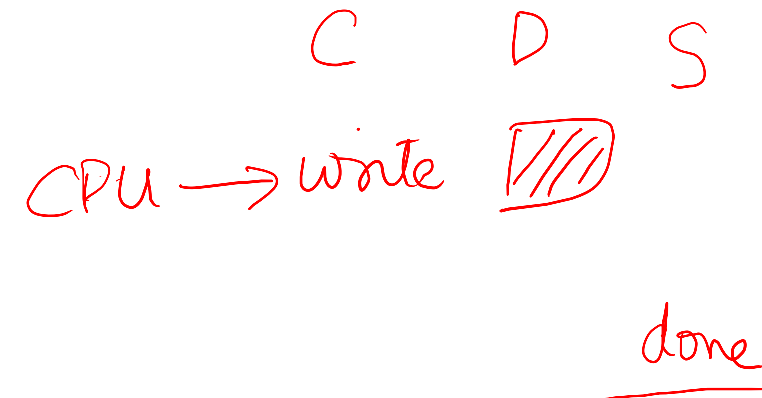


- I/O device is managed by a device controller
  - Microcontroller with registers, communicates with CPU/memory over bus
- Device controller has three main registers (simple model)
  - Command: action to execute given by CPU
  - Data: Information corresponding to action
  - Status: Status of action reported by device
- How does CPU read/write registers?
  - Explicit I/O instructions: special in/out instructions to access device registers
  - Memory mapped I/O: device registers are mapped to memory addresses (like bytes in RAM), and accessed via load/store instructions

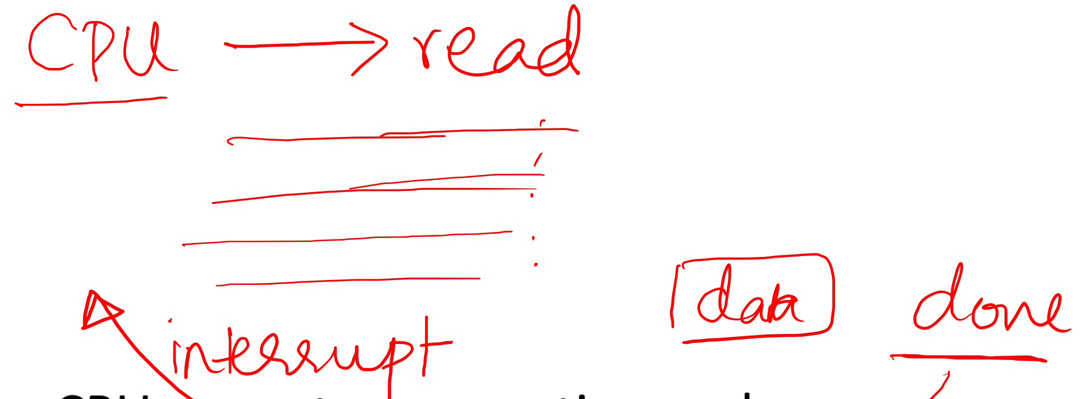


Example of I/O device communication

- Example of reading a block from hard disk
  - CPU gives command to read a specific block number
  - After data is available from disk, it is stored in the data register
  - Status register is set to indicate completion after data is available
- Example of writing a block to hard disk
  - CPU writes command and data registers
  - Status register is set to indicate completion of write

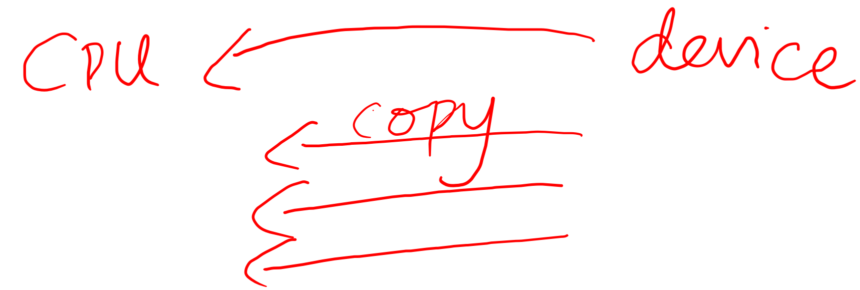


Polling vs. Interrupts

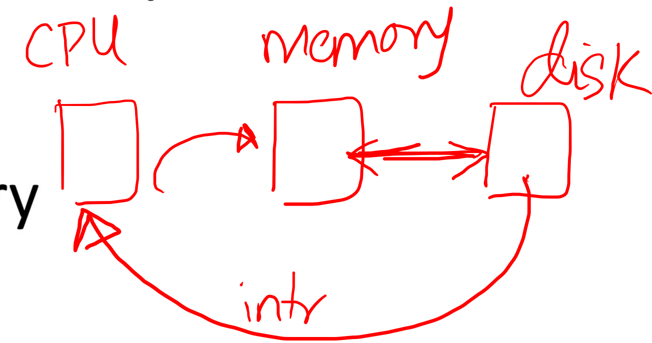


- I/O devices are much slower than CPU
  - Hard disk takes few milliseconds to fetch data, CPU runs at nanosec timescale
- CPU gives command to read a block from disk, what does it do until the data is ready?
- Polling: CPU keeps checking the status register, copies data when status is ready
  - Wastes CPU cycles, inefficient
- Better model: CPU gives command, proceeds to do other work
  - Device raises an interrupt (sends a signal) when data is ready
  - Device is given an interrupt number (IRQ) to identify specific interrupts
  - CPU free to do other work until device raises interrupt
- Interrupt-driven I/O uses CPU more efficiently

Direct Memory Access (DMA)



- Data is ready in device register, but must come into main memory for programs to use
  - How to copy data from device register to main memory?
- One option: CPU reads data from device, writes into memory
  - Inefficient: data goes into CPU registers first, then into RAM
- Better option: Direct Memory Access (DMA)
  - When data is ready, device controller directly accesses RAM via system bus, deposits data into memory (it is told which memory address to use beforehand)
  - When device is transferring data, CPU is free to do other work
  - Once data is copied, device raises interrupt, CPU has data ready
- DMA is efficient, saves CPU cycles, especially for devices which transfer large amounts of data (hard disk, network card)



Device driver

- Device specific knowledge required to correctly read/write registers in device controller to handle I/O operations
  - Done by special software called device driver
  - Part of operating system code
- Functions performed by device driver
  - Starting I/O operations, giving commands
  - Providing device with addresses of DMA buffers, or copying data
  - Handling interrupts from device, or polling

Summary

- In this lecture:
  - Memory hierarchy
  - Communicating with I/O devices
- Explore your computer: how much RAM does it have? What I/O devices are connected to your computer?