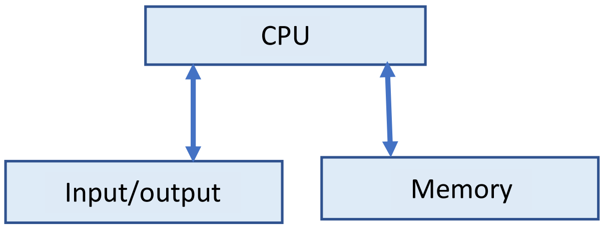
Computer Architecture

1. Computer Architecture

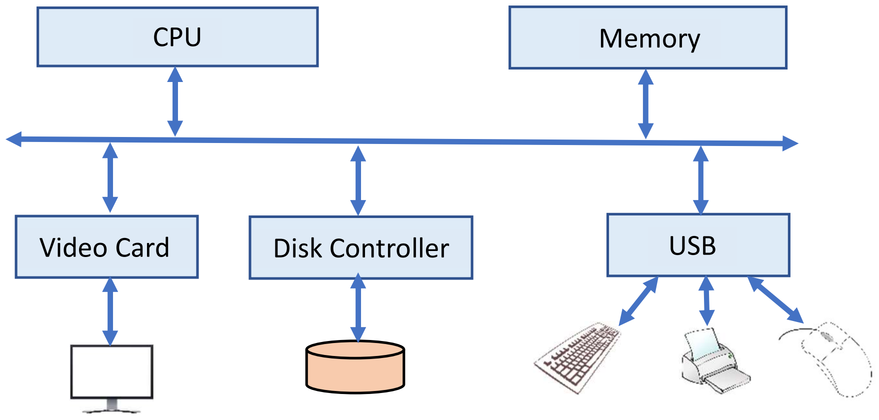
A. The Von Neumann Architecture

- CPU: performs operations and controls the sequence of operations
- Memory: stores both instructions and data. Instructions and data can only be fetched by the CPU for executing and accessing.
- Input/output: inputs and outputs data from devices such as keyboard and to devices such as printer.



B. Today's Computer Architecture

- All components are connected via a bus.
- Each I/O device has a controller such as video card for monitors and disk controllers for hard disks.
- Controllers hides physical differences between different makers and different models.
- Hard disks are I/O devices to the CPU.



2. CPU

- Comprised of a Control Unit, an ALU (Arithmetic and Logic Unit), and
- A set of registers:
    1. PC – Program Counter - pointing to the next instruction to be executed.
    2. SP – Stack Pointer – used for push and pop operations
    3. PSW – Program Status Word - storing the flag of computation results and CPU status
    4. R1 … Rn - general purpose registers storing memory addresses and data

- NOTE: When a program is being executed, the values of all the registers are the **context** of the **execution** of the program. A program may be stopped and resumed its execution if we can restore its context – the values of all the registers.

3. Instruction Set

- Each CPU has a set of machine instructions it understands and executes
- Commonly used instructions:

1.	LOAD Ri, M(addr)	// reads the content of memory at addr into register Ri
2.	STORE Ri, M(adder)	// stores the contents of Ri to memory location addr.
3.	ADD Ri, Rj	// add the value in Ri to Rj.
4.	SUB Ri, Rj	// subtracts the value in Ri from Rj
5.	IN Ri, P(number)	// reads a value from I/O port number to Ri
6.	OUT Ri, P(number)	// writes the value in Ri to I/O port number.
7.	CMP Ri, Rj	// compares the contents of Ri and Rj – 1: equal, 0: not equal
8.	JMP Z addr	// if the last CMP is not equal, jump to the instruction at addr.
9.	JMP LT addr	// if Ri > Rj in last CMP, jump to addr.
10.	STI	// sets interrupt enable bit
11.	CLI	// clears interrupt enable bit
12.	TRAP	// cause a software interrupt

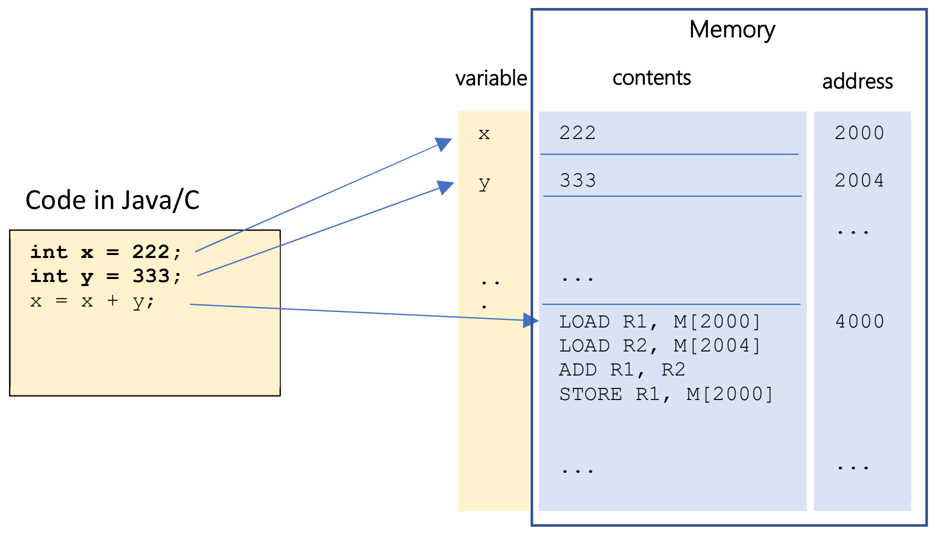
- Each instruction corresponds to a sequence of binary digits – called machine language.

4. Memory

- It is often called Random Access Memory (RAM).
- Memory is used to store instructions to be executed and data to be processed
- Each memory location is one byte or 8 binary digits.
- All memory locations are assigned with unique addresses from 0 to max.
- A 4GB RAM has about 4 billion bytes

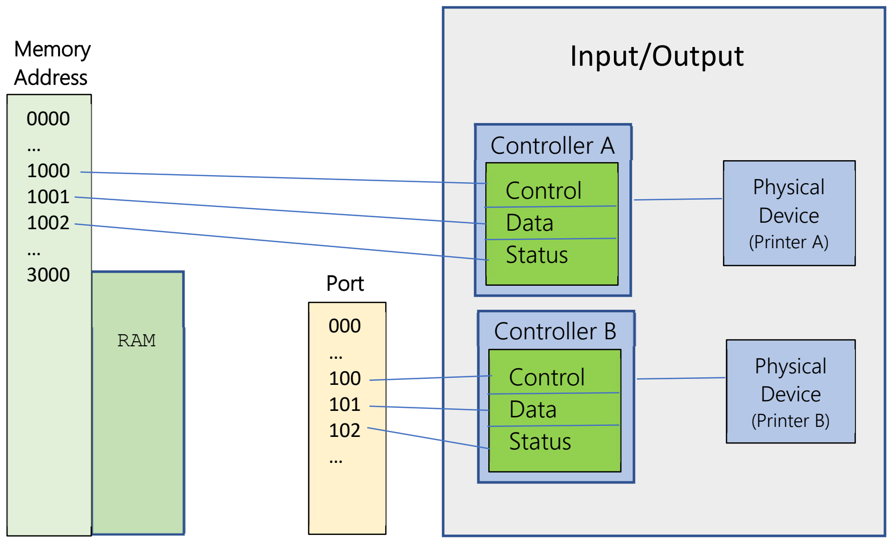
5. Programs in Assembly/Machine Language – an example

- Variables x and y are located in memory at locations 2000 and 2004, respectively. Each integer takes 4 bytes of memory space.
- The x = x + y statement is compiled into machine code (assembly) starting at memory address 4000.
- When the statement is to be executed, PC first points to the first LOAD instruction. When the instruction is being executed, PC is moved to next instruction – 2nd LOAD instruction, and repeats.
- NOTE: a variable, e.g., x, has an **address** (2000) and a **value** (222). The separation of address and value is important in understanding pointers in C.



6. Input/Output

A. I/O Hardware



- A controller normally consists of three registers: a **status** register indicating the status of the device (busy or idle for example), a **command** or **control** register telling the device to perform a certain task, and a **data** register, holding data to be read from the device or to be sent to the device.
- Those registers from all I/O controllers may be mapped to a separate address space called ports, each port corresponds to register, or mapped to memory address space.

B. I/O with Separate Address Space

- Each register of a controller is mapped to an address called a port in an address space separate from the memory
- The IN and OUT instructions are used for read and write operations to the register at each port.
  1. IN R1, Port[101] // read Data register at port 101 to R1
  2. OUT R3, Port[100] // write R3 to Control register at port 100
- Example: Prints a char saved in R2

```
 Loop: IN R1, Port[102]     // read Status register
```

```
        CMP R1, 0x00        // if printer busy ("00" ready for next char to print)
        JMP Z Loop          // if busy, check again
        OUT R2, Port[101]   // write char in R2 to Data register
        OUT 0xFF, Port[100] // Write "FF" to Control register to starting printing
```

C. Memory-Mapped I/O

- Each register of a controller is mapped to an address of the memory space.
- Instead of a location of RAM, what behind the memory address is a register of a controller
- The LOAD and STORE instructions are used to access the I/O registers.
  1. LOAD R1, M[1001]   // read from Data register at memory address 1001 to R1
  2. STORE R3, M[1000]   // write R3 to Control register at memory address 1000.
- Example: Prints a char saved in R2

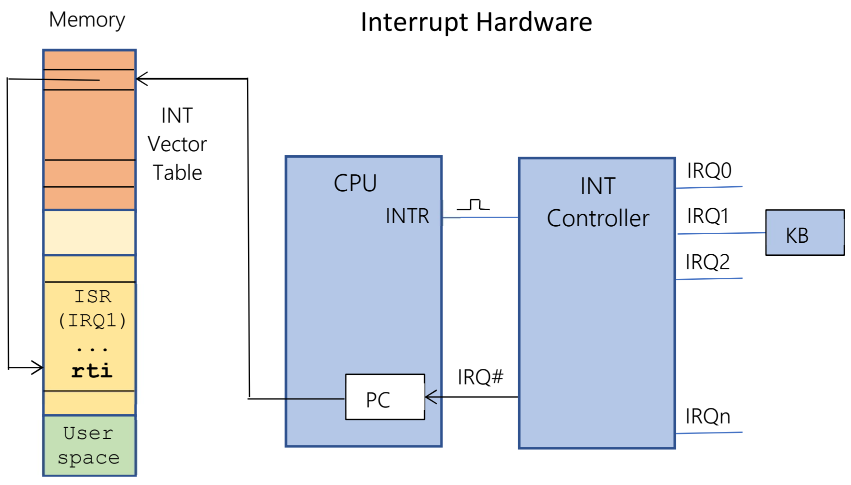
```
  Loop: LOAD R1, M[1002]     // read Status register
        CMP R1, 0x00         // if printer busy ("00" ready for next char to print)
        JMP Z Loop           // if busy, check again
        STORE R2, M[1001]    // write char in R2 to Data register
        STORE 0xFF, M[1000]  // Write "FF" to Control register to starting printing
```

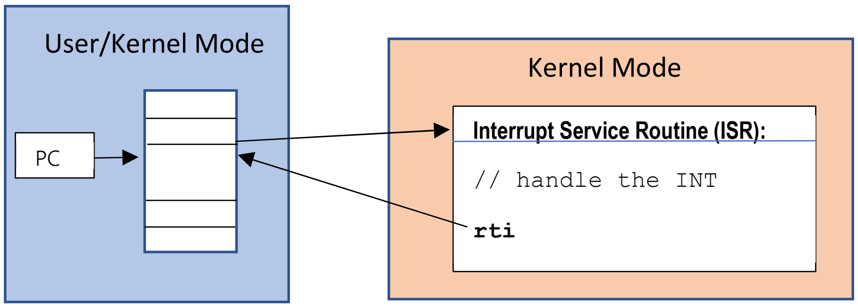
7. Interrupts

- Both printing sample programs above used a loop to check the status of the printer and wait until the status has changed. This is termed "busy waiting" for it is using the CPU time to continuously check the status until it changes.
- "Busy waiting" wastes CPU time. To solve this problem, how about we ask the I/O control/device to signal the CPU when its status has changed, e.g., printer ready for next char. This is the basic idea of Interrupt.

A. Basic Components of Interrupt

- Interrupt Controller: a hardware component that is connected to I/O controllers that send interrupt signals to INT Controller. E.g., keyboard controller is connected to input IRQ1 of INT Controller.
- INT controller prioritizes interrupts from I/O controllers. E.g., interrupts from network have a priority than those from keyboard since network is much faster.
- INT Controller sends a pulse (signal) to an input pin of the CPU commonly marked as INTR (INTerrupt Request) and the IRQ#, to the bus, of the I/O device that sent an interrupt with highest priority.
- The IRQ# is used to search the Interrupt Vector Table by the CPU to find the routine (Interrupt Service Routine (ISR)) to be executed for the interrupt.
- A special instruction rti (return from interrupt) is executed at the end of the ISR.





B. The Interrupt Processing Sequence:

- A user presses a key on the keyboard (KB) and the KB sends an interrupt to the interrupt controller (INT Controller).
- The INT Controller determines there is an INT from IRQ1 and it is not disabled and there is no other interrupt with higher priority, it sends an interrupt signal to the INTR pin of the processor chip and sends the ID of the interrupt the address bus.
- The processor detects the interrupt after finishing the execution of the current instruction, saves PC, PSW, and SP on the stack, find the address of the IRS by looking up the Interrupt Vector Table (IVT) using the ID of the interrupt from the address bus. The processor also switches to kernel mode (a bit in PSW).
- The processor starts executing the ISR and the last instruction rti (return from interrupt), which pops SP, PSW, and PC from the stack and continue from the instruction the new PC points, effectively continuing the execution of the interrupted program. The rti also restores the processor mode back to the one before the interrupt.
- NOTE: Interrupts are a key feature of today's computer. Computers would not be able to do multitasking with interrupts
- NOTE: The execution of each instruction is atomic. In other words, interrupts are only handled before or after the execution of an instruction.
- NOTE: Interrupts are caused by events external to the CPU. No one can predict at which instruction (or the PC value) an interrupt may occur. We say interrupts are asynchronous with the execution of instructions.

8. Kernel Mode and User Model

- Privileged instructions
  1. IN and OUT instructions should not be executed by user programs.
     - Consider if two programs execute OUT instructions to prints at the same time.
  2. STI and CLI should not be executed by user programs
     - A user program can disable all interrupts – system becomes unresponsive.
- To prevent user programs from executing privileged instructions
  1. Kernel Mode of the CPU
     - All instructions including privileged instructions can be executed.
  2. User mode of the CPU
     - Only non-privileged instructions can be executed.
     - Privileged instructions are illegal instructions in this mode
     - User programs are executed in user model
  3. Then, how user programs do I/O?

        ▪ Use system calls.

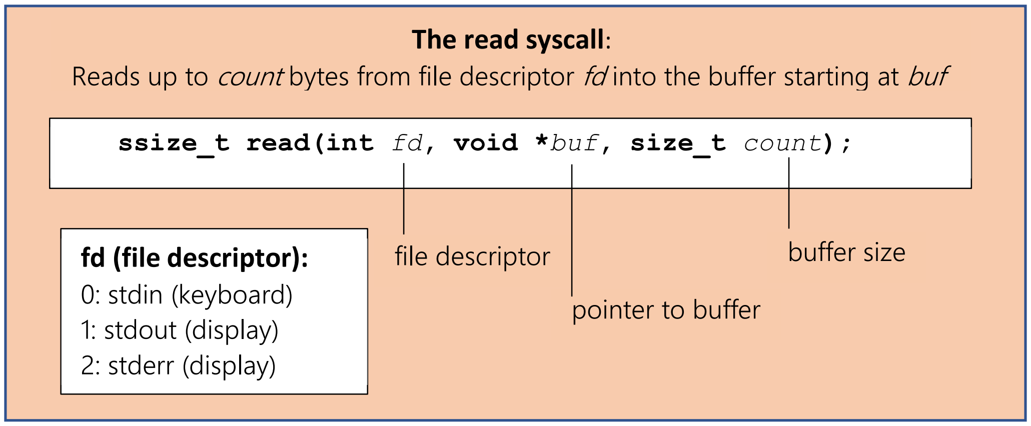
9. System Calls

- The OS is executed in kernel mode and user programs are only executed in user mode.
- Thus, privileged instructions can only be used by the OS..
- The OS performs I/O tasks (and other tasks) on the behalf of the user programs.
- Those tasks are organized as a set of system calls.
- System calls are the API of the operating system for user programs.
- Examples of system calls:
    1. open: opens a file
    2. read: reads data from a file
    3. write: writes data to a file
    4. fork: creates a new process
    5. pthread_create: creates a new thread.

A. Implementation of System Calls

- A user program calls a system call:
    1. the user program executes in user mode, and
    2. the system call executes in kernel
- Thus, the call to a system call must switch the CPU from user to kernel mode.
- When the system call returns, it switches the CPU back to user mode.
- An example:

```
ssize_t read(int fd, void *buf, size_t count);
```



The read syscall:
Reads up to *count* bytes from file descriptor *fd* into the buffer starting at *buf*

```
ssize_t read(int fd, void *buf, size_t count);
```

fd (file descriptor):
0: stdin (keyboard)
1: stdout (display)
2: stderr (display)

The read system call is used for all I/O on the Unix operating systems. All I/O devices are abstracted as files and an ID called the file descriptor is assigned once a file is open. For example, the standard input (i.e., keyboard) has an ID of 0, the standard output (i.e., the monitor) is 1.

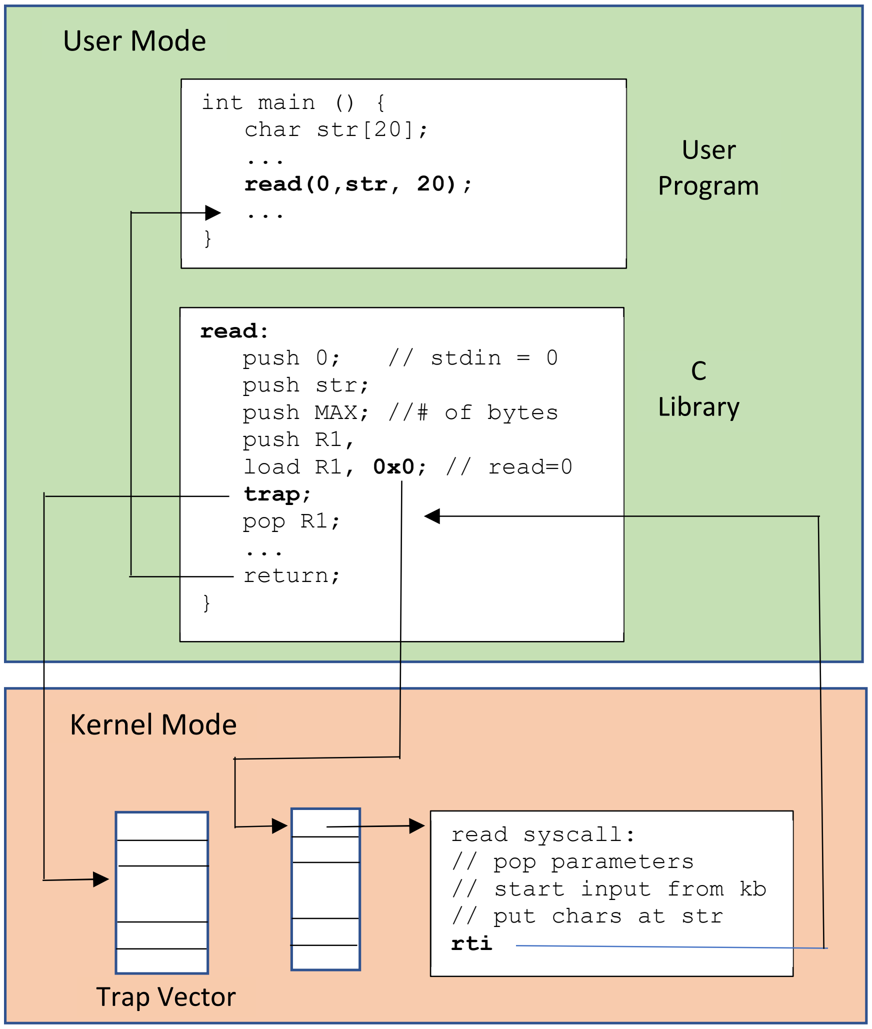
Note that each programming language offers its own set of I/O functions. For example, scanf and printf are functions in C for inputting from keyboard and outputting to display. Those language specific functions are implemented by the system calls when they are compiled for the given operating systems. For example, in C on Unix, scanf is implemented with the read system call as the keyboard is abstracted as an input file.

On Unix operating systems, system calls are often presented as a set of C functions and they can be called in C programs directly. The C library function for the system call is listed above. The three parameters are the file descriptor of the input file (of the input device), the address of the buffer and the size of the buffer. Data type size_t is of unsigned long int (64 bits on 64-bit Unix).

Below shows how an input function in C is implemented in Unix. When a call to C's read function, the user program is executed in user mode along with the read function. The C function first pushes the parameters on the stack, saves register R1 and load the predefined ID for the read system call into R1. The execution of the **trap** instruction causes a software interrupt or an exception, which transfers the CPU from user mode to kernel mode. The OS uses R1 to determine the read system call was requested using the ID as an index to the system call table to find the actual routine of the read system call. The routine pops the parameters from the stack and starts the input. Once the input is completed, the routine executes the rti instruction to return to the read function, which pops the original value of R1 and returns program that made the call. The execution of the rti instruction also switch the CPU from kernel mode back to user mode.

NOTE: The execution of the **trap** instruction causes a software interrupt or an exception. The ISR, which is part of the OS, determines that it is a request for a system call.

NOTE: Software interrupts or exceptions are **synchronous** with the execution of instructions. In other words, each time **trap** is executed, an interrupt occurs.



```
int main () {
   char str[20];
   ...
   read(0,str, 20);
   ...
}
```

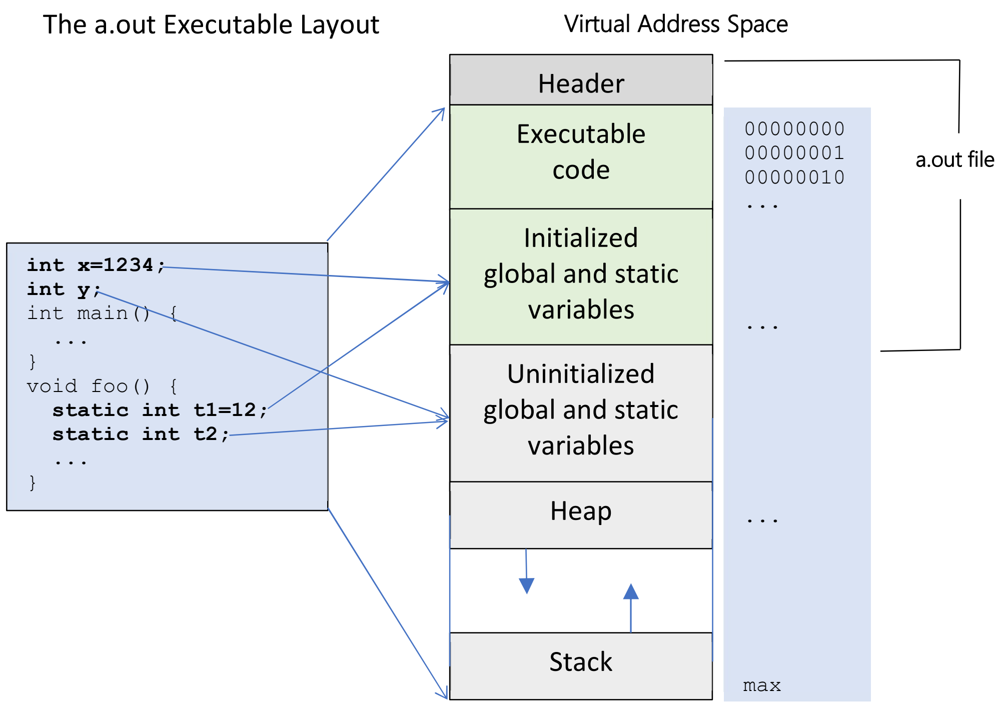
```
read:
   push 0;    // stdin = 0
   push str;
   push MAX;  //# of bytes
   push R1,
   load R1, 0x0;  // read=0
   trap;
   pop R1;
   ...
   return;
}
```

```
read syscall:
// pop parameters
// start input from kb
// put chars at str
rti
```

10. Physical Address Space and Virtual Address Space

A. The a.out Layout

- All programs are compiled into their own virtual space, starting from address 0 to max size.
- The binary layout of the program when loaded in memory consists of the following parts:
    1. Code section (text section) – executable code
    2. Data section - Initialized global and static variables
    3. Uninitialized global and static variables
    4. Heap – for dynamically allocated objects
    5. Stack – Stack for function parameters and local variables.
- The a.out executable layout
    1. Header – sizes of different parts and the total size
    2. Code section
    3. Data section.

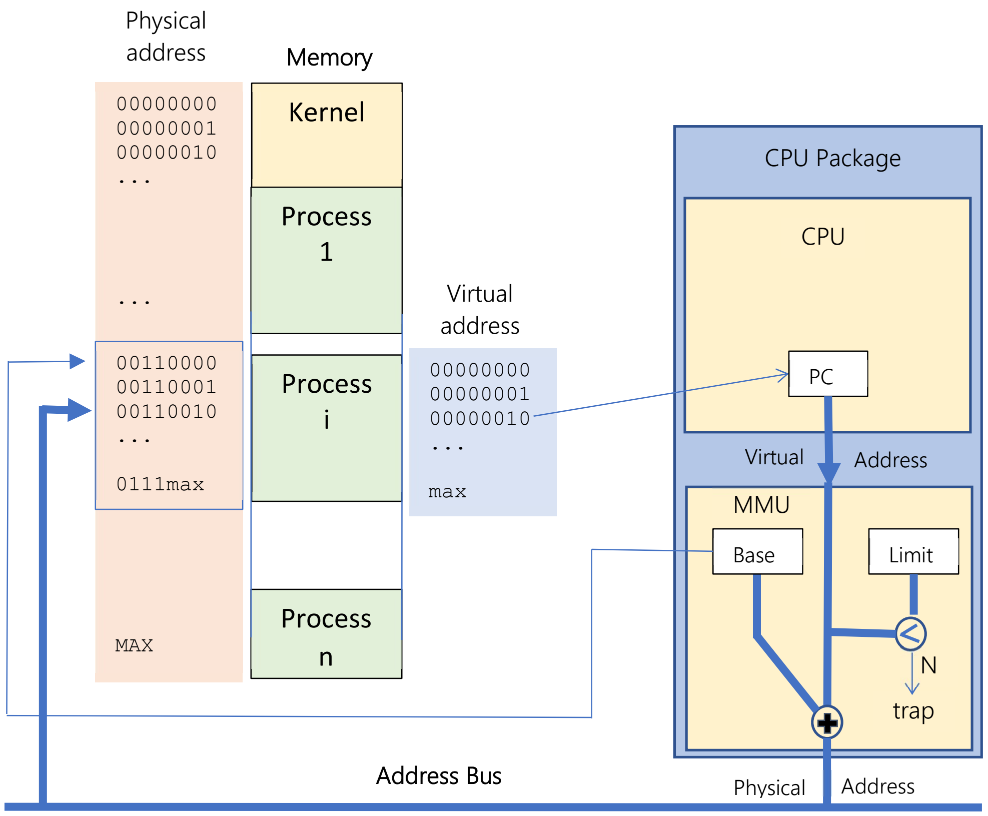

The a.out Executable Layout
Virtual Address Space
Header
Executable code
Initialized global and static variables
Uninitialized global and static variables
Heap
Stack
00000000
00000001
00000010
. . .
. . .
. . .
max
a.out file

```
int x=1234;
int y;
int main() {
  ...
}
void foo() {
  static int t1=12;
  static int t2;
  ...
}
```

B. Address Mapping

- Each program being executed is called a process.
- A program is loaded into memory in its virtual space.
- When a process is running, the PC and all addresses are of the virtual space.
- When a process is being executed, the base and limit registers of the Memory Management Unit (MMU) stores the starting physical address from which the process is loaded and the size of the process.
- When another process is to be executed, the base and limit of the new process are loaded into MMU by the kernel.
- Each virtual address from the CPU (the PC, address registers, etc) goes through the MMU.

    1. if it is larger than the limit register, an illegal memory access exception is raised.
    2. Otherwise, the virtual address plug the base register becomes the physical address and sent to the address bus for memory access.
- For example, if the program is loaded at physical address 12000 in decimal and the size if 2000.
    1. Base = 12000; limit = 2000
    2. If PC = 134, then its physical address = 12134
    3. If variable X is located at 1345, then its physical address is 12000 + 1345 = 13345
    4. If it tries to access 2345, it is larger than the limit, so illegal memory access exception is raised.



3.3. Variables = address + value

- Each variable in a high-level program language has an address and a value.
- For Example: int x = 123, located at 2000 virtual address;
    1. address(x) == 2000
    2. value(x) = 123
- in C,
    1. to access the address of a variable: &x
    2. to access the value of a variable: x.