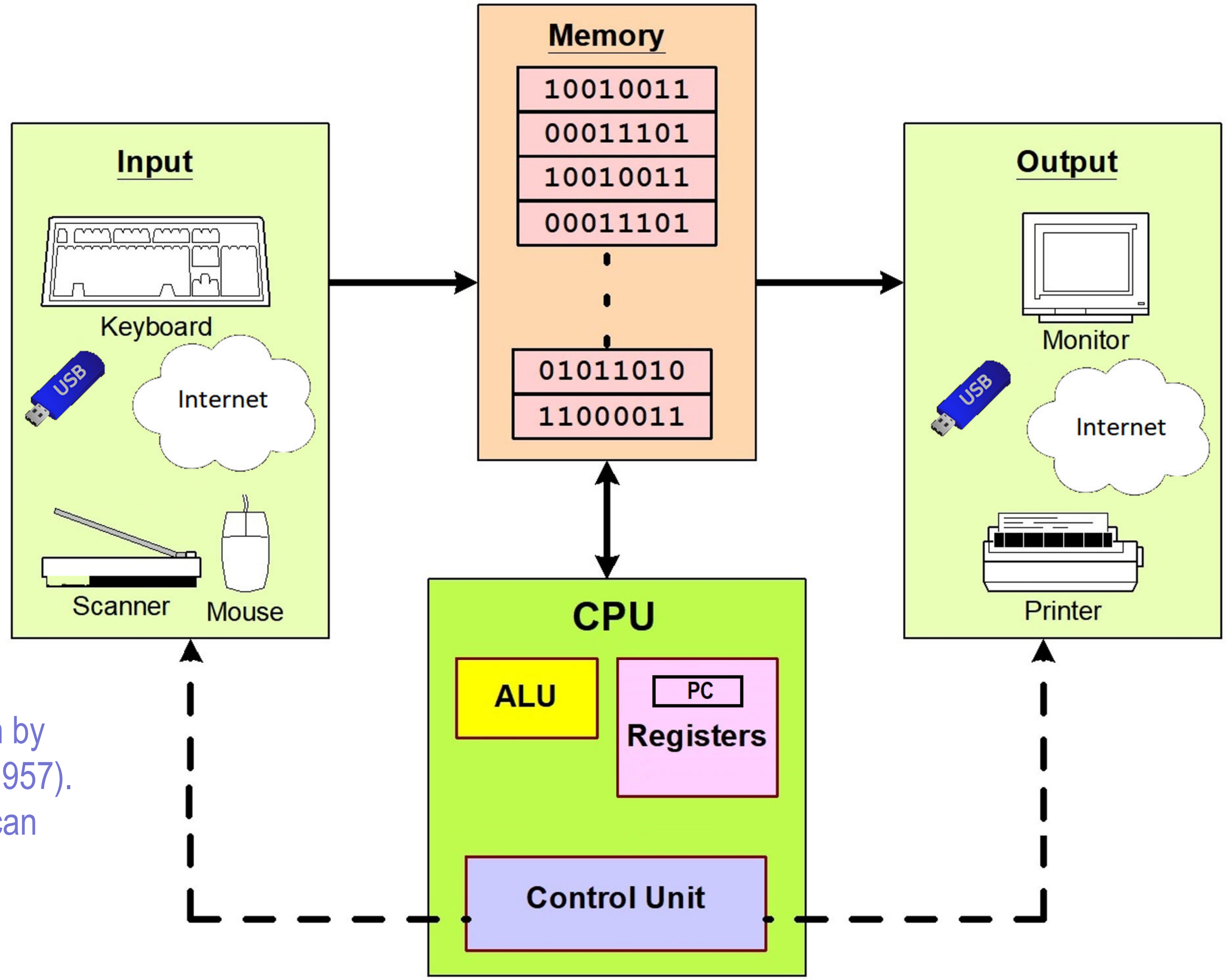
EEECE7376: Operating Systems Interface and Implementation

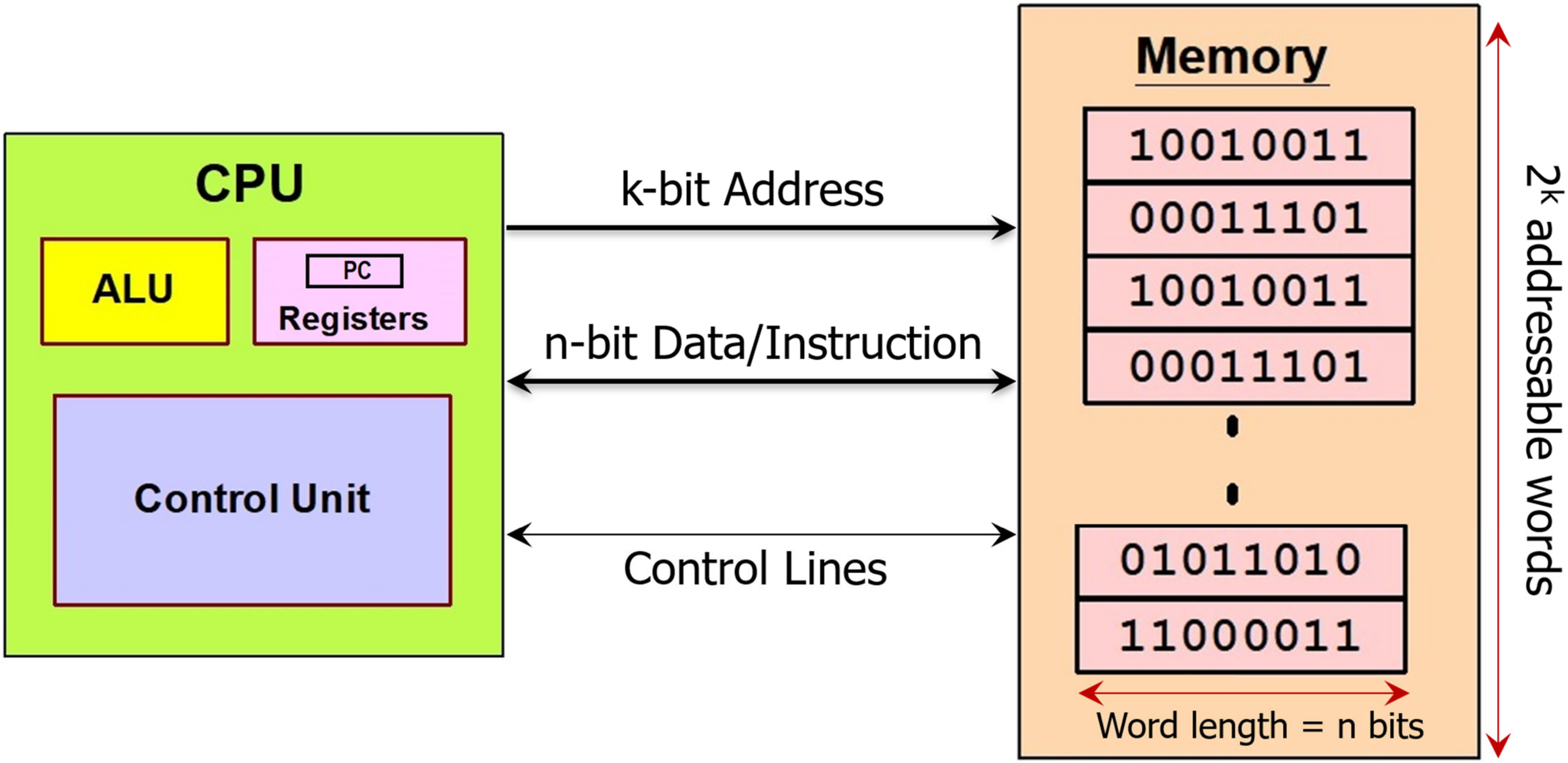
Introduction

The Von Neumann Computer Model



It is a computer architecture based on a 1945 description by John Von Neumann (1903-1957). He was a Hungarian-American mathematician, physicist, inventor, computer scientist

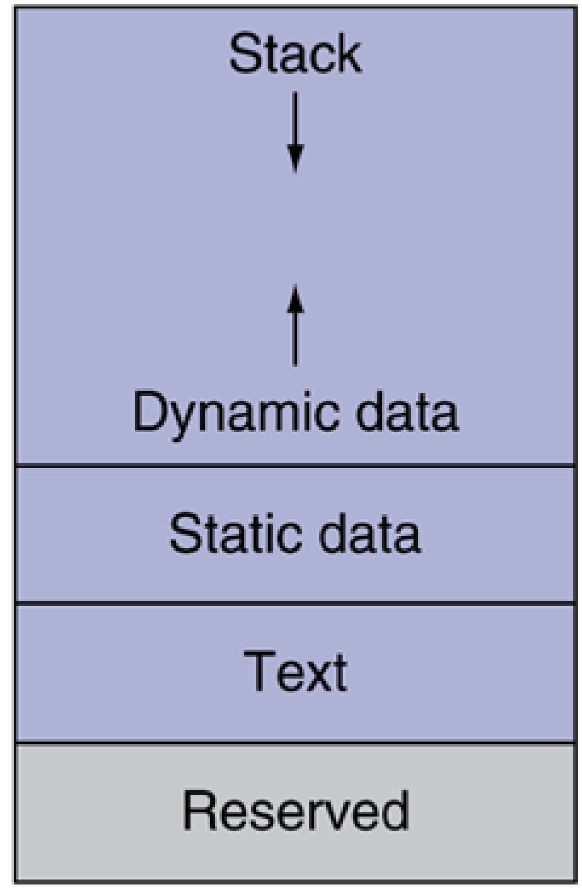
Processor/Memory Interface



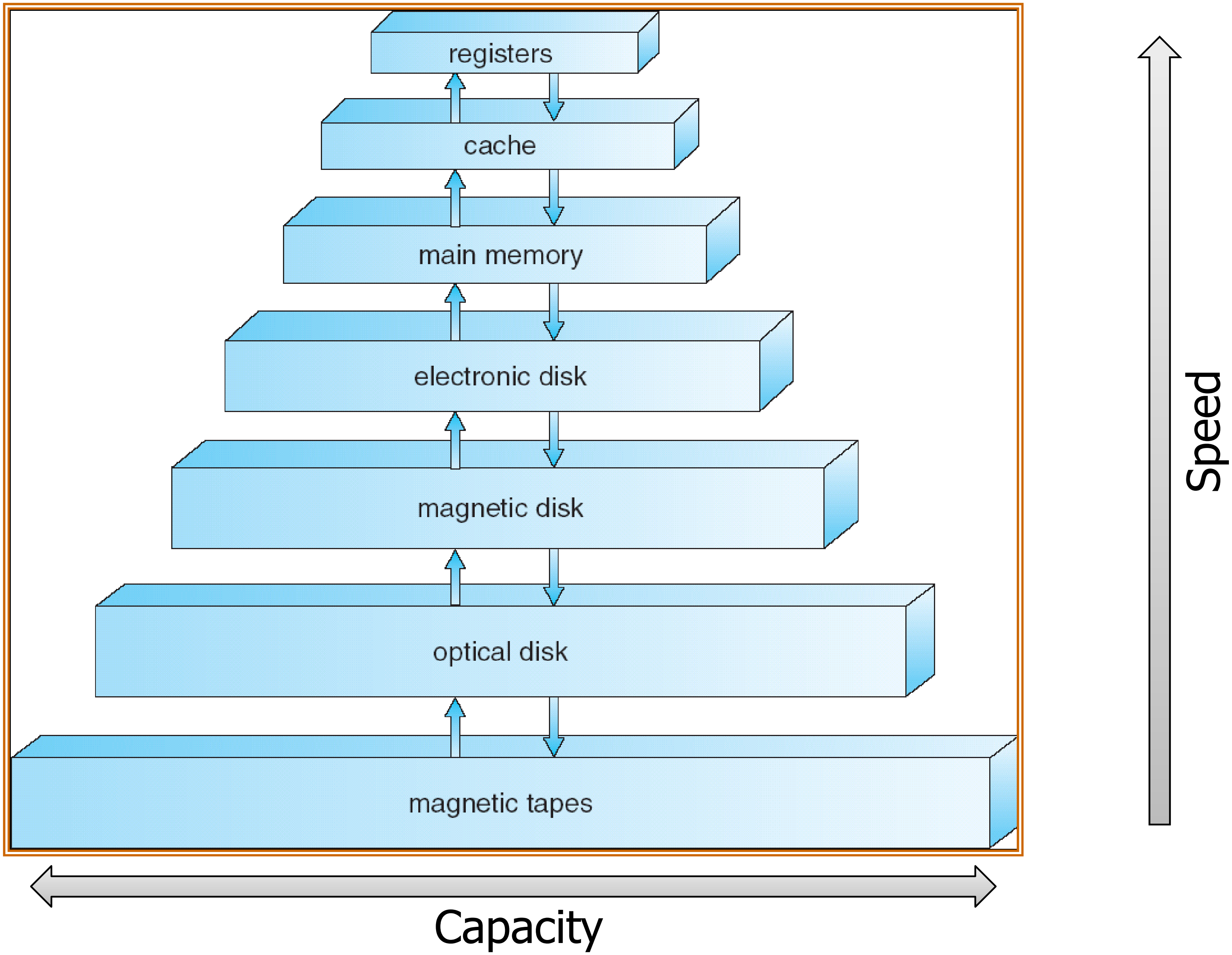
When a computer starts, what is the address of the first instruction to be executed and is this first instruction part of the operating system?

Memory Layout of a Program

- Text: program code
- Static data:
  - Global and static variables C++
- Stack:
  - Local variables of functions
- Dynamic data: heap
  - In C++, created by **`new`** and removed by **`delete`**



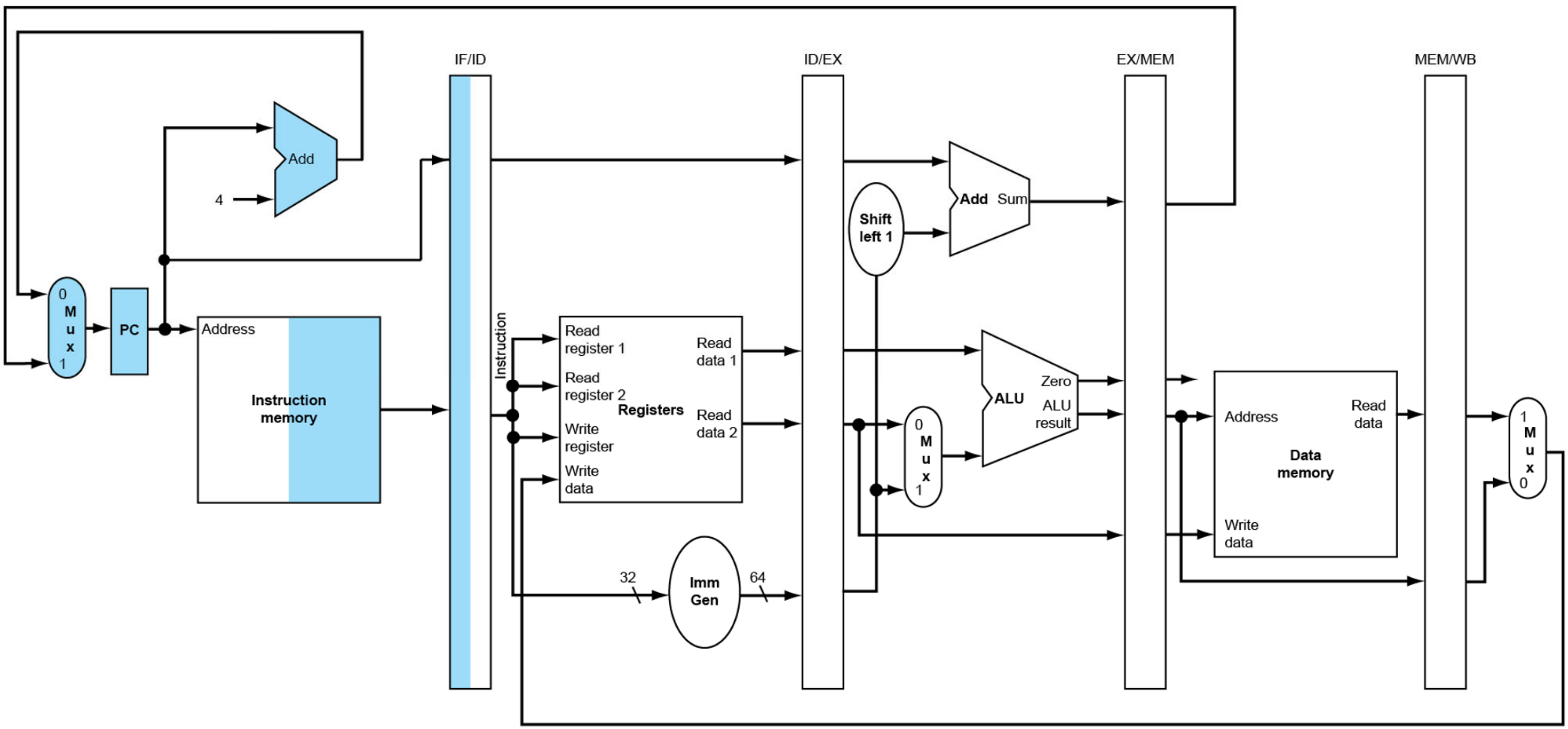
Memory/Storage Hierarchy



An Instruction Execution Steps

1. **Fetch and PC update**
   - Use the program counter (PC) to supply the instruction address and fetch instruction from memory.
   - Update the PC for the next fetch (e.g., PC $\leftarrow$ PC + 4)
2. **Decoding** decodes instruction and reads operands
   - Extract opcode: determine what operation should be done
   - Extract operands: register numbers or immediate from the instruction
3. **Execution**
   - Use ALU to calculate one of the following tasks:
     – Arithmetic or logical operations on registers contents.
     – Calculating the memory address for load/store
     – Calculating the branch target address
   - Access memory for load/store
4. **Write back one of the following results**
   - PC $\leftarrow$ target address for jump and taken branch
   - Update destination register for R-type, addi, or lw instructions.
   - Update destination memory for the sw instruction.

RISC Pipelined Microarchitecture



A processor guarantees **atomic execution** of each single instruction (either executes it as a whole or none of it).

Sample Assembly Program

- The following RISC-V program prints the contents of an array of one-byte integers on the screen.

```
.data
   v: .byte 2, 7, 1, 4, 3
.text
main:
  li   t0, 0     # Array index
cond:
  li   t1, 5     # Array size
  bge t0, t1, endloop
body:
  la   t2, v     # Array base address
  add t2, t2, t0  # base + offset
  lb   t2, 0(t2)
```

```
   li    a7, 1  # print int
   mv    a0, t2
   ecall

   addi t0, t0, 1
   j cond

endloop:
   li a7, 10 # exit
   ecall
```

Why does the program needs the OS to handle this output operation instead of writing directly to the screen?

What if the exit ecall is missing?

C Pointers Review

- A pointer is merely an address of where a datum or structure is stored
  - Pointers operations are based on the type of entity that they point to.
  - To declare a pointer, use * preceding the variable name: `int *p;`
- To set a pointer to a variable's address use & before the variable as in `p = &x;`
  - & means "return the memory address of"
  - in this example, p will now point to x
- If you access p, you merely get the address
- To get the value that p points to, use * as in *p
  - `*p = *p + 1;` will add 1 to x
- * is known as the ***indirection*** (or **dereferencing**) operator because it requires **a second access**.
- *p and x are known as aliases, two ways of accessing the same memory location.

Pointers Example 1

```
int main() {
 int x = 1, y = 2;
 int z[3] = {10, 20, 30};
 int *p;
 p = &x;          // p now points at the location where x is stored
 y = *p;          // set y equal to the value pointed to by p, or y = x
 printf("x= %d  y= %d\n",x,y); // x=1  y=1
 *p = 0;          // now change the value that p points to to 0, so now x = 0
                  // but will that change y's value?
 printf("x= %d  y= %d\n",x,y); // x=0  y=1
 p = &z[0];       // now p points at the first location in the array z, z[0]
 *p = *p + 1;     // the value that p points to (z[0]) is incremented
 ++p;             // now p points at the second location in the array z, z[1]
 *p = *p + 1;     // the value that p points to (z[1]) is incremented
 printf("z[0]= %d  z[1]= %d\n",z[0],z[1]); // z[0]=11  z[1]=21
 return 0;
}
```

By how much is the content of **p** incremented after the p++ operation?

Arrays and Pointers

Example: `int z[3] = {10, 20, 30};`

- In C, the name of the array is actually a pointer to the first array element (i.e., `z = &z[0]`)
- Therefore, there are two ways to access the first element of array z: either `z[0]` or `*z`
- What about accessing `z[1]`?
  - We can do `z[1]` as usual, or we can add 1 to the location pointed to by z, that is *(z+1). Is this equivalent to *(++z)?
- While we can update a pointer value (like ++p in Example 1), we cannot update the value of z (e.g., ++z) , otherwise we would lose access to the first array location since z is our array variable.
  - Notice in Example 1, if an int size is 4 bytes then when we did p++, p's value is incremented by 4. This is because p is of type "`int *`".
- If p is pointing to doubles, the increment ++p would increment p by 8 bytes away, and by 1 if it points to char type.

Iterating Through the Array

- The following are two ways to print an array's contents:

```
int j;
int z[3] = {10, 20, 30};
 for(j = 0; j < 3; j++)
     printf("%d \n", z[j]);
```

```
int *p;
int z[3] = {10, 20, 30};
 for(p = z; p < z + 3; p++)
        printf("%d \n", *p);
```

- For the code on the right:
  - p started by having the value of z, that is the address of `z[0]`.
  - The loop iterates while p `< z + 3;` where z+2 is the address of the last element of the 3-element array z.
  - `p++` increments the pointer to point at the next element in the array.
- The code on the right is more efficient as the only operation inside the loop is incrementing p. While the code on the left increments j and then add j*4 to the base address of the array.
- *Note:* `++(*p);` increments the value of the element to which p points. While `*(++p);` increments the pointer to point at the next array element and then return the value of that element.

Pointers Example 2

```
int main()
{
 int x[4] = {12, 20, 39, 43}, *y;
 y = x;                          // y points to the beginning of the array
 printf("%d \n", x[0])  ;        // outputs 12
 printf("%d \n", *y)    ;        // also outputs 12
 printf("%d \n", *y+1)  ;        // outputs 13 (12 + 1)
 printf("%d \n", *(y+1));        // outputs x[1] or 20
 y+=2;                           // y now points to x[2]
 printf("%d \n", *y)    ;        // prints out 39
 *y = 38;                        // changes x[2] to 38
 printf("%d \n", *y-1)  ;        // prints out x[2] - 1 or 37
 y++;                            // sets y to point at the next array element
 printf("%d \n", *y)    ;        // outputs x[3] (43)
 (*y)++;                         // sets what y points to, to be 1 greater
 printf("%d \n", *y)    ;        // outputs the new value of x[3] (44)
 return 0;
}
```

Passing Arrays to Functions

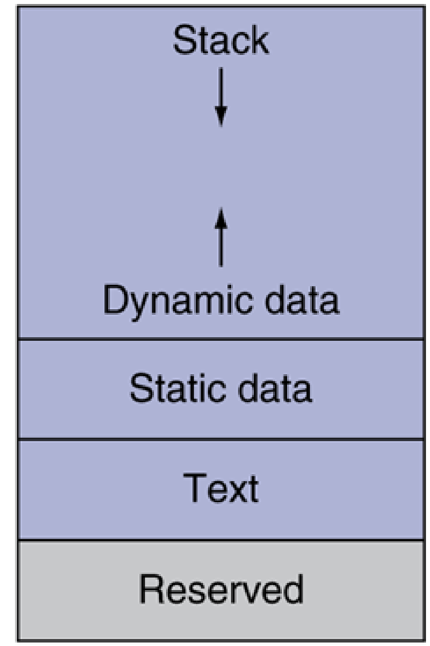
- When an array is passed to a function, what is being passed is a pointer to the array
  - In the formal parameter list, you can either specify the parameter as an array or a pointer

```
int array[100];
…
afunction(array);
…

void afunction(int *a) {…}
      or
void afunction(int a[]) {…}
```

Dynamic Memory Allocation (1 of 2)

- To this point, we have been declaring pointers and having them point at already created variables.
- An important use of pointers is to create dynamic structures.
  - structures that can have data added to them or deleted from them such that the amount of memory being used is equal to the number of elements in the structure.
  - this is unlike an array which is static in size.
- Creating and maintaining dynamic data structures requires dynamic memory allocation – the ability for a program to obtain more memory space at execution time and to release the space that is no longer needed.



Dynamic Memory Allocation (2 of 2)

- In C, the **`malloc`** and **`free`** *library functions* are essential to dynamic memory allocation.
- **`malloc`** is used to request memory space enough to hold a specific data type or an array of the data type.
- Dynamically allocate memory to an integer

```
int *ptr = (int *) malloc(sizeof(int));
```

- Dereference the pointer to access the memory

```
*ptr = 8;            // assings 8 to memory
```

- At the end free up the memory for reuse

```
free(ptr);           // returns memory to the system.
```

Dynamic Memory Operators for Arrays

- Dynamically allocate memory to an integer array with 10 elements

```c
int *arr = (int *) malloc(10 * sizeof(int));
```

- Dereference the pointer to access the memory

```c
arr[0] = 8;  // access element at index 0
```

- At the end free up the memory for reuse

```c
free(arr);  // free up memory.
```

Pointers Example 3

```
//Program to read and calculate the average of class grades
float AverageGrades(int grades[], int n) {
    float sum = 0;
    for(int i = 0; i < n; i++) sum += grades[i];
    return sum/n;
}
int main() {
  int* gradesP;
  int ClassSize;
  printf("What is the class size? ");
  scanf("%d", &ClassSize);
  gradesP = (int *) malloc(ClassSize * sizeof(int));

  for(int i = 0; i < ClassSize; i++) {
     printf("What is grade # %d ? ", i);
     scanf("%d", &gradesP[i]);
  }
  printf("Grades average = %f \n", AverageGrades(gradesP, ClassSize));
  free(gradesP);
  return 0;
}
```

Linux and C Resources

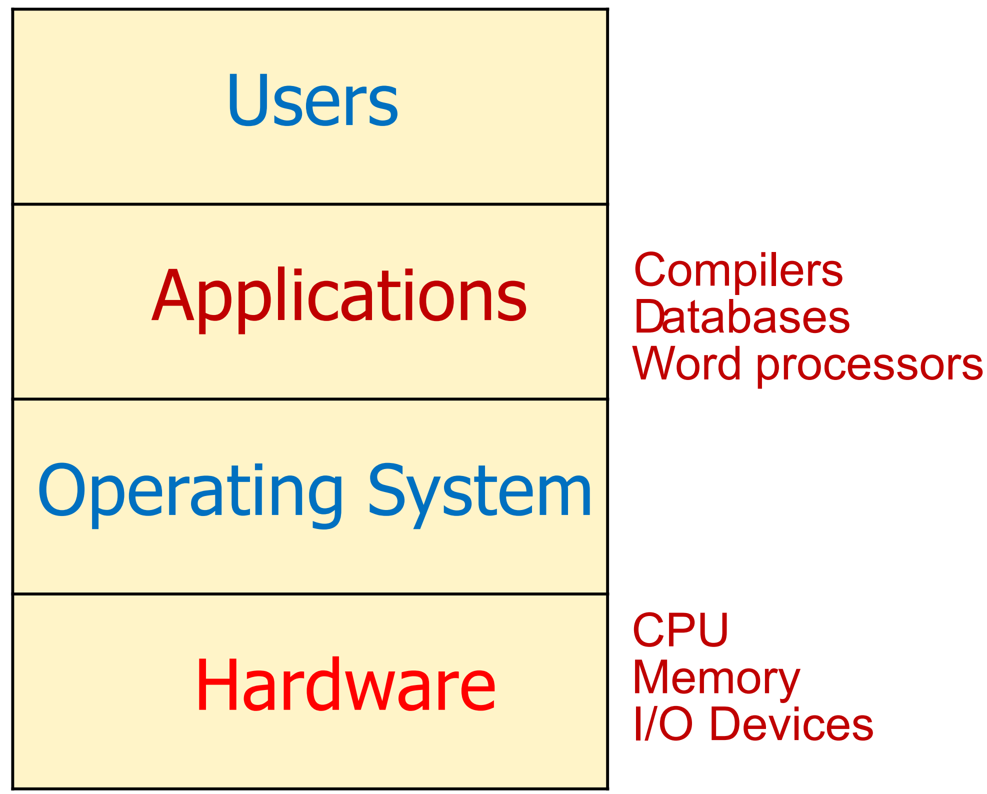
- COE Linux gateway:
  https://wiki.coe.neu.edu/xwiki/bin/view/Main/#HLinuxHelp
  For accounts support: https://coe.northeastern.edu/computer/
  To access from off campus:
  https://wiki.coe.neu.edu/xwiki/bin/view/Main/VPN/

- Installing Windows Subsystem for Linux (WSL) and then the Ubuntu Linux distribution:
  - https://docs.microsoft.com/en-us/windows/wsl/install
  - https://ubuntu.com/tutorials/ubuntu-on-windows#1-overview
  - You will need to install the gcc compiler by running:
    `sudo apt install g++` or `sudo apt install gcc`

- Incomplete introduction to the C programming environment from the book Appendix F:
  - https://pages.cs.wisc.edu/~remzi/OSTEP/lab-tutorial.pdf

What is an Operating System?

- The OS is a software that acts as an intermediary between a user of a computer and the computer hardware.
- It is responsible for making sure the computer system operates correctly, efficiently, fairly, reliably, and in an easy-to-use manner.
- It carries its task by providing:
  - Virtualization
  - Resource management
  - Security and Protection

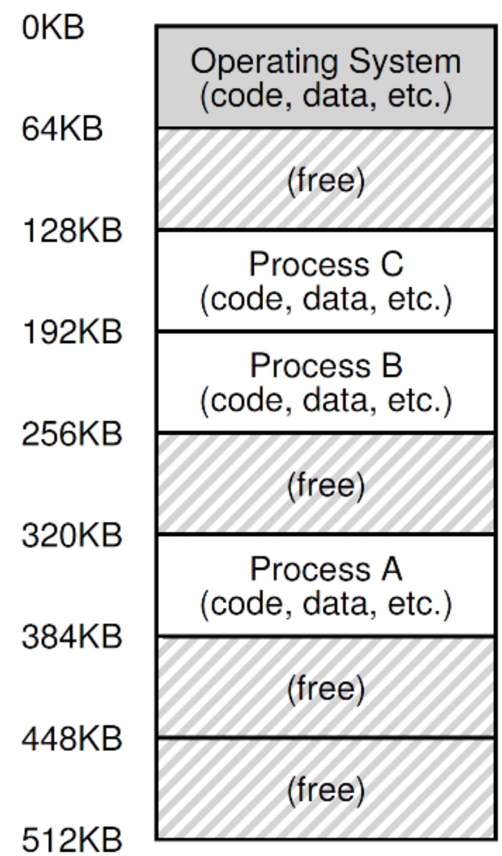


Virtualization

- That is, the OS takes a physical resource (such as the processor, or memory, or a disk) and transforms it into a more general and easy-to-use virtual form of itself.
- The OS manages and abstracts the low-level hardware, so that, for example, a word processor need not concern itself with which type of disk hardware is being used.

OS Kernel and System Calls

- The OS kernel is a special program that provides services to running programs (processes).
- When a process needs to invoke a kernel service, it invokes a procedure call in the operating system interface. Such a procedure is called a **system call**.
- The system call enters the kernel; the kernel performs the service and returns. Thus, a processor alternates between executing in **user space** and **kernel space**.
- System calls are usually called through the programming language standard library. The library makes different devices look the same by providing higher-level abstractions



Resources Management

- Virtualization allows many programs to concurrently:
  - run (thus sharing the CPU),
  - access their own instructions and data (thus sharing memory), and
  - access devices (thus sharing for example the disks),
- The OS is sometimes known as a *resource manager*.
- Advantages of managing computer resources:
  - **Protect** applications from one another
  - Provide **efficient** and **fair** access to resources

CPU Virtualization Experiment

- Code: github.com/remzi-arpacidusseau/ostep-code/tree/master/intro
- Run four background processes of the `cpu.c` code
  `./cpu A &  ./cpu B & ./cpu C & ./cpu D &`
- *Observation*: Even though we have only one processor, somehow all four of these processes seem to be running at the same time.
- *Conclusion*: The OS, with some help from the hardware, oversees giving the illusion that the system has a very large number of virtual CPUs. Allowing many programs to seemingly run at once is what we call virtualizing the CPU.
- *Question*: if two programs start to run at the same time, which should run first? The OS as a **resource manager** needs to answer this by providing a **policy**.
- *Linux tips*:
  - In WSL Ubuntu, to open a new terminal select Ubuntu from the down arrow on the top bar.
  - `ps -A` //lists all running processes in all terminals
  - `killall cpu` //kills the above running background cpu processes
  - `kill <process number>` //kill specific process using its number.

CPU Virtualization Code

```
#include <stdio.h>
#include <stdlib.h>
#include "common.h"

int main(int argc, char *argv[])
{
    if (argc != 2){
     fprintf(stderr, "usage: cpu <string>\n");
     exit(1);     }

    char *str = argv[1];
    while (1)     {
     printf("%s\n", str);
     Spin(1);     }
    return 0;
}
```

Memory Virtualization Experiment

- Code: github.com/remzi-arpacidusseau/ostep-code/tree/master/intro
- Run two background processes of the `mem.c` code (you might need to add `(int *)` before malloc): `./mem 1 &` `./mem 200 &`
- *Observation*:
  - Each of the above process allocates memory at the same address (e.g., 0x200000)*, and yet each seems to be updating the value at 0x200000 independently! It is as if each running program has its own private memory, instead of sharing the same physical memory.
- *Conclusion*:
  - Each process accesses its own private **virtual address space**, which the OS and hardware somehow maps onto the physical memory of the machine.
  - A memory reference within one running program does not affect the address space of other processes.
  - The reality, however, is that physical memory is a shared resource, managed by the operating system (the **resource manager**).

*For this to happen, your OS needs to be set to disable address-space layout randomization(ASLR). In Ubuntu this can be done by: (replace 0 with 2 to enable it)

```
echo 0 | sudo tee /proc/sys/kernel/randomize_va_space
```

Memory Virtualization Code

```c
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include "common.h"

int main(int argc, char *argv[]) {
    if (argc != 2) {
        fprintf(stderr, "usage: mem <value>\n");
        exit(1);    }
    int *p;
    p = malloc(sizeof(int));    assert(p != NULL);
    printf("(%d) addr pointed to by p: %p\n", (int) getpid(), p);
    *p = atoi(argv[1]); // assign value to addr stored in p
    while (1) {
        Spin(1);
        *p = *p + 1;
        printf("(%d) value of p: %d\n", getpid(), *p);    }
    return 0; }
```

Concurrency Experiment

- Code: github.com/remzi-arpacidusseau/ostep-code/tree/master/intro
- Compile the `threads.c` code using:

```
g++ threads.c -pthread -o threads
```

  - Run it once with argument 1,000 and several times with argument 100,000.
- *Observation*:
  - With 1000, the final value of the counter is 2000 as expected, as each thread incremented the counter 1000 times.
  - With 100000, every time we get a different wrong value for the counter (we might sometimes get the correct 200000 value).
- *Conclusion*:
  - Both threads **share** the same data segment.
  - The reason for these odd outcomes is that instructions are executed one at a time. Incrementing the counter takes three instructions: load the value of the counter from memory into a register, increment the register content, and finally store this content back into memory.
  - As these three instructions do not execute **atomically** (all at once), strange things happen. It is the problem from a **race condition** with **concurrency**.

Concurrency Code

```c
#include <stdio.h>
#include <stdlib.h>
#include "common.h"
#include "common_threads.h"
volatile int counter = 0;
int loops;

void *worker(void *arg) {
 int i;
 for (i = 0; i < loops; i++)
   counter++;

 return NULL;
}
```

```c
int main(int argc, char *argv[]) {
 if (argc != 2) {
   fprintf(stderr,
       "usage: threads <loops>\n");
        exit(1); }
 loops = atoi(argv[1]);
 pthread_t p1, p2;
 printf("Initial value : %d\n", counter);
 Pthread_create(&p1, NULL, worker, NULL);
 Pthread_create(&p2, NULL, worker, NULL);
 Pthread_join(p1, NULL);
 Pthread_join(p2, NULL);
 printf("Final value   : %d\n", counter);
 return 0;
}
```

Persistence

- We need hardware and software to be able to store data **persistently**; vs. the **volatile** storage of the computer RAM.
  - Example saving files in hard disks or solid-state drives (SSDs).

- The software in the OS that usually manages these storage drives is called the **file system** that provides the user with several systems calls.
  - Unlike the abstractions provided by the OS for the CPU and memory, it does not create a private, virtualized storage drive for each application. Rather, it is assumed that **users want to share information** that is in files (among processes and sometimes among other users).

Persistence Experiment

- Code: github.com/remzi-arpacidusseau/ostep-code/tree/master/intro
- Run one process of the `io.c` code.
- *Observation*:
  - We used three system calls (open, write, and close) to create a file and write a text in it.
  - With the help of the OS **file system** and the various **device drivers**, the file and its content are accessible through directory and file names, regardless of the technology of the physical storage device.
- *Conclusion*:
  - The OS provides a standard and simple way to access devices through its **system calls**.

Persistence Code

```c
#include <stdio.h>
#include <unistd.h>
#include <assert.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <string.h>

int main(int argc, char *argv[]) {
    int fd = open("h.txt",
             O_WRONLY | O_CREAT | O_TRUNC, S_IRUSR | S_IWUSR);
    assert(fd >= 0);
    char buffer[20];
    sprintf(buffer, "hello world!!\n");
    int rc = write(fd, buffer, strlen(buffer));
    assert(rc == (strlen(buffer)));
    fsync(fd);
    close(fd);
    return 0;
}
```

OS Design Goals (1 of 2)

- Build up some **abstractions** in order to make the system convenient and easy to use.
- High **performance**; another way to say this is our goal is to minimize the overheads of the OS.
- **Security** against external malicious applications is critical, especially in these highly-networked times.
- Provide **protection** between processes. Protection include the mechanisms that allow the implementation of security.
  - The heart of one of the main principles underlying an operating system, which is that of **isolation**; isolating processes from one another is the key to protection.

OS Design Goals (2 of 2)

- The operating system must also run non-stop; when it fails, all applications running on the system fail as well. Therefore, operating systems often strive to provide a high degree of **reliability**.
- **Energy-efficiency** is important in our increasingly green world.
- **Mobility** is increasingly important as OSes are run on smaller and smaller devices.

Reading List

- From Arpaci-Dusseau textbook:
  - Section 2.6 Some History