Before a Program Runs

- a compiler translates C code into an executable containing machine instructions

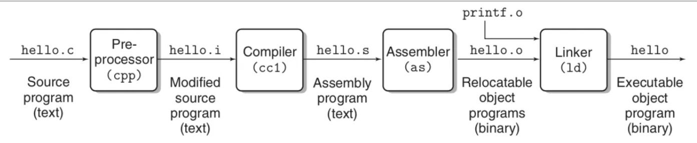


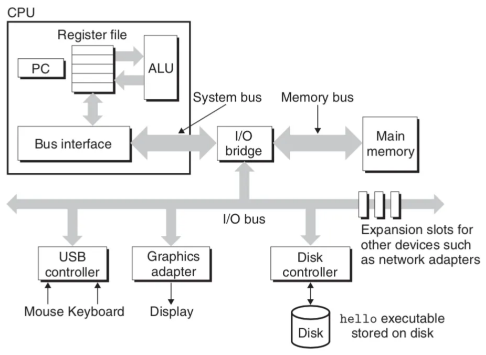
Figure 1.3 **The compilation system.**

- the executable is loaded from disk to main memory when the program starts
  - or as needed

https://medium.com/@wangwei09310931/notes-of-csapp-1-a-tour-of-computer-systems-3ab14138fbd7, from *Computer Systems: A Programmer's Perspective*

System Architecture

Figure 1.4
Hardware organization of a typical system. CPU: Central Processing Unit, ALU: Arithmetic/Logic Unit, PC: Program counter, USB: Universal Serial Bus.



https://medium.com/@wangwei09310931/notes-of-csapp-1-a-tour-of-computer-systems-3ab14138fbd7, from *Computer Systems: A Programmer's Perspective*

When a Program Runs

repeat –

- fetch
  - get the current instruction from main memory and store it in a register
- decode
  - determine which instruction it is
- execute
  - carry out the instruction (add two numbers, access memory, check a condition, jump to another instruction, …)
- next instruction
  - update the program counter to the next instruction

Operating Systems

The OS is responsible for ensuring that the system operates correctly and efficiently.

The OS –

- makes it easy to run programs
- facilitates running many programs at once
- allows programs to share memory
- enables programs to interact with devices

Operating Systems

The OS is a virtual machine.

- provides abstractions of physical resources (CPU, memory, disk)
  - allows applications to reuse common facilities
  - provides a common interface to different devices
  - can provide higher-level functionality
- provides a standard library (system calls) to applications for running programs and accessing memory and devices

The OS is a resource manager.

- manages system resources (CPU, memory, disk)
- allows programs to share resources
  - protects applications from one another
  - provides efficient access to resources
  - provides fair access to resources

Three Key Pieces – 1

- virtualization
  - the OS makes each application think it has the system resources (CPU, memory) to itself
  - the CPU abstraction is the *process*
  - the memory abstraction is the *virtual address space*
    - physical memory is a shared resource, but memory references within one process do not affect the address spaces of other processes

Demo: Virtualizing the CPU

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <assert.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) {
        Spin(1);
        printf("%s\n", str);
    }
    return 0;
}
```

```
prompt> gcc -o cpu cpu.c -Wall
prompt> ./cpu "A"
A
A
A
A
^C
prompt>
```

Demo: Virtualizing the CPU

```
prompt> gcc -o cpu cpu.c -Wall
prompt> ./cpu "A"
A
A
A
A
^C
prompt>
```

```
prompt> ./cpu A & ./cpu B & ./cpu C & ./cpu D &
[1] 7353
[2] 7354
[3] 7355
[4] 7356
A
B
D
C
A
B
D
C
A
...
```

Demo: Virtualizing Memory

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

int
main(int argc, char *argv[])
{
    int *p = malloc(sizeof(int));                   // a1
    assert(p != NULL);
    printf("(%d) address pointed to by p: %p\n",
           getpid(), p);                            // a2
    *p = 0;                                         // a3
    while (1) {
        Spin(1);
        *p = *p + 1;
        printf("(%d) p: %d\n", getpid(), *p);    // a4
    }
    return 0;
}
```

```
prompt> ./mem & ./mem &
[1] 24113
[2] 24114
(24113) address pointed to by p: 0x200000
(24114) address pointed to by p: 0x200000
(24113) p: 1
(24114) p: 1
(24114) p: 2
(24113) p: 2
(24113) p: 3
(24114) p: 3
(24113) p: 4
(24114) p: 4
...
```

Three Key Pieces – 2

- concurrency
  - the OS must juggle many processes at once, and programs can be *multithreaded*
  - OS hides the concurrency from independent processes
  - OS provides tools to manage the concurrency with interacting processes
    - provides abstractions (locks, semaphores, ...) to processes
    - ensures processes don't deadlock

Demo: Multiple Threads

```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;
}

int main(int argc, char *argv[]) {
    if (argc != 2) {
        fprintf(stderr, "usage: threads <value>\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;
}
```

```
prompt> gcc -o threads threads.c -Wall -pthread
prompt> ./threads 1000
Initial value : 0
Final value   : 2000
```

```
prompt> ./threads 100000
Initial value : 0
Final value   : 143012    // huh??
prompt> ./threads 100000
Initial value : 0
Final value   : 137298    // what the??
```

Three Key Pieces – 3

- persistence
  - data typically needs to outlive individual processes
  - main memory is *volatile* – the data is lost when the system crashes or is turned off
  - the disk abstraction is the *file system* (files, directories)
  - the OS takes care of low-level details
    - disk management – the OS figures out where to store data, issues I/O requests
    - reliability – protection against failures during writes
    - optimized performance – disks are slow