Tutorial 1

Gidon Rosalki

2025-03-26

Notice: If you find any mistakes, please open an issue at `https://github.com/robomarvin1501/notes_operating_systems`

1 Motivation

What happens when you press a button? Read or write a file? Reading a file is slow, so how does the computer keep being responsive? How can you do background tasks like listening to music, scan for viruses, with a single processor? How can we run programs that use more RAM than we actually have on the computer? What does segmentation fault actually mean? How can we speed up programs by using a multi core system? How doe we deal with the problems that arise from this? How can you run multiple OSs on a single machine? How do computer communicate in practice?

2 Introduction

2.1 What is an OS

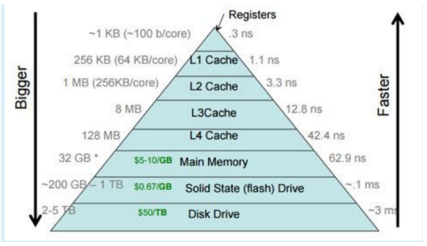
A piece of computer software that manages the resources of the computer. These include Computational Resources (CPU time and scheduling), the memory, files, I/O. It provides an abstraction to the programmer/user for many pieces of the hardware, such as memory allocation, file reading and writing, etc. It is also the only software permitted to execute privileged instructions.

Operating Systems are difficult since unlike many programs, the OS works without knowing the input beforehand. Also, the performance is of paramount importance, any OS slowdown is felt in the entire computer. They must work with the complicated modern computer hardware, such as CPUs, RAM, graphics cards, and so on. Additionally, it must always work no matter what, and be secure whilst doing all this.

2.2 CPU

The CPU contains registers, and only understands registers. Important registers include the Program Counter (PC), the Stack Pointer (SP), and Instruction Register (IR). It executes a set of instructions, which are generally included in data handling (storing or moving values), arithmetic operations, and control flow (branching from ifs and so on). Not every one of these instructions take the same amount of time, for example reading the L1/L2 cache is slower than accessing the registers.

2.3 Memory



Main memory is located on chips outside the CPU. The program instructions and process data is kept here. Should this run out, then we might use the disk as a "swap" space to store this data, though that is much slower. Main memory (RAM) is volatile, and loses all data on power loss, whereas the disk will keep its data. The main memory also has much less memory than the hard disk. Through DMA the disk controller can write and read information directly from the main memory. There is also another type of memory, the cache which is stored directly on the CPU die. Due to being directly on the die, it is very fast, though very limited on space. There are typically 3 levels, L1, L2, and L3. Cache does not replace the main memory, it is just a fast "pool" to save things temporarily. Using a variable often dictates that we will need it again, and this time it is faster to access since it is in the cache. When a variable is moved to the cache, variables that are "near" it (think arrays) are also moved to the cache, since we will likely need them too.

2.4 Virtualisation

Virtualisation is the creation of a virtual version of computer hardware, storage devices, computer network resources, etc. A virtual machine (VM), is an emulation of the computer system. There can be many VMs on a single computer system, and each VM has its own resources and operating system.

Running many VMs comes at a severe performance overhead, since every VM needs to emulate an entire machine itself. To resolve this we may use containers. These are OS level virtualisation. A program inside a container (see Figure 2.) can only access the contents and devices assigned to the container, yet direct access to container files from the host is possible. Multiple container can run on a single host, and they all share a kernel with the other containers, and the host machine.

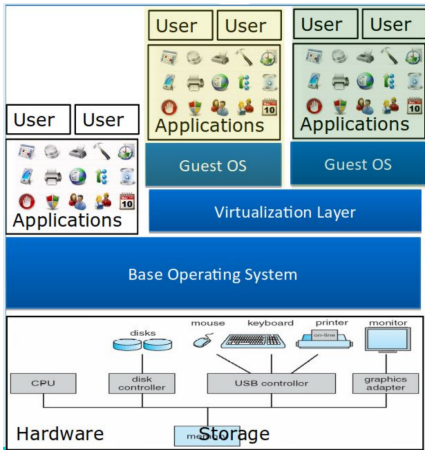


Figure 1: Virtualisation

2.5 Valgrind (our lord and saviour)

Valgrind is a framework debugging and profiling code on a linux system It detects memory leaks, incorrect usage of memory related functions (such as malloc, free, etc), out of bounds errors, and may hint resolution to segmentation faults. It is also recommended to add prints to aid debugging. However, sometimes the prints take time to be printed, due them being sent to a print buffer. In order to force the OS to print the contents of the print buffer, we may use the function

```
int fflush(FILE *_Nullable stream);
```

We may also use the function

```
void setbuf(FILE *restrict stream, char *restrict buf);
```

Setting the restrict value to 0 ensures that anything sent to the print buffer is immediately printed.
Valgrind does not see all memory problems, for example not initialising values. For this type of error, use the flag -Wall (and perhaps -Wextra should you so wish). Using -Werror will turn all the warnings into errors, and fail compilation when this happens. Using this flag is required.
The valgrind flag -fstack-protector-strong is a very useful flag, but it significantly slows the running of the program, so use with caution.

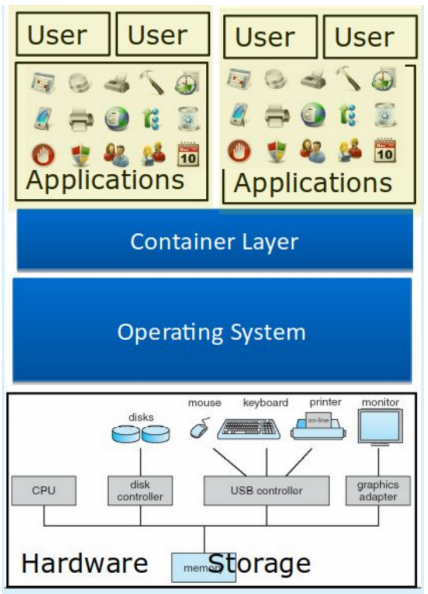


Figure 2: Containers

2.5.1 Example 1

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

void
foo (int n)
{
    int i;
    int *a = (int *)malloc (n * sizeof (int));
    a[0] = 1;
    printf ("a[0] = 1\n");
    a[1] = 1;
    printf ("a[1] = 1\n");
    for (i = 2; i += n; i++)
        {
            a[i] = a[i - 1] + a[i - 2];
            printf ("a[%d] = %d\n", i, a[i]);
        }
    free (a);
}

int
main (int argc, char *argv[])
{
    foo (10);
    return 0;
```

```
}
```

This example has an out of bounds mistake, in that we access the 11th place in the memory in the for loop. This won't show up in most running of the program, but is caught by valgrind.

2.5.2 Example 2

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

void
foo (int n)
{
   int *a = (int *)malloc (n * sizeof (int));
   int i;
   for (i = 0; i < n; i++)
      {
         a[i] = i * i;
         printf ("a[%d] = %d\n", i, a[i]);
      }
}

int
main (int argc, char *argv[])
{
   foo (10);
   return 0;
}
```

In this example, there is a block of memory that is allocated, but not freed. This is once again caught by valgrind.

2.5.3 Example 3

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

typedef struct Foo
{
   int arr[3];
   int bar;
} Foo;

int
main (int argc, char *argv[])
{
   setbuf (stdout, 0);
   Foo t;
   int i;
   printf ("start\n");
   t.bar = 6;
   printf ("t.bar = %d\n", t.bar);
   for (i = 0; i < 3; i++)
      {
         t.arr[i] = i + 1;
      }
   t.arr[i] += t.arr[0] + t.arr[1];
   printf ("t.arr[2] = %d\n", t.arr[2]);
   printf ("t.bar = %d\n", t.bar);
   return 0;
}
```

In this example Foo.bar is overwritten at the end of the for loop. This will not be caught by valgrind, but will be caught by the flag -fstack-protector-strong. This is also a good time to use CLion memory watching.

2.5.4 Example 4

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

int
main (int argc, char *argv[])
{
    int arrLen = 10;
    int *arr;
    int i;
    arr = (int *)malloc (arrLen * sizeof (int));
    for (i = 0; i < arrLen; i++)
        {
            arr[i] = i;
        }
    free (arr);
    for (i = 0; i < arrLen; i++)
        {
            arr[i] = i * 2;
        }
    printf ("Done\n");
    return 0;
}
```

Use after free! Very bad. Valgrind will catch this error. This is closely related to the next listing:

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

int
compute (int len, int *arr)
{
    int sum = 0, i;
    for (i = 0; i < len; i++)
        {
            sum += arr[i];
        }
    free (arr);
    return sum;
}

int
main (int argc, char *argv[])
{
    int arrLen = 10, sum, i;
    int *arr;
    arr = (int *)malloc (arrLen * sizeof (int));
    for (i = 0; i < arrLen; i++)
        {
            arr[i] = i;
        }
    sum = compute (arrLen, arr);
    free (arr);
    printf ("sum = %d", sum);
    return 0;
}
```

Free after free! Again, this will be caught by valgrind, but is still a very bad error, and apparently commonly made in this course. Make sure to document your

functions properly, and to write in their documentation who is responsible for the memory used in them.

2.6 Debugging system calls

We have a few basic system calls:

```
int open(const char *pathname, int flags, ... /* mode_t mode */ );
```

```
ssize_t read(int fd, void buf[.count], size_t count);
```

```
ssize_t write(int fd, const void buf[.count], size_t count);
```

open returns a file pointer to the file specified by the pathname. If the specified file does not exist, it may optionally (if O_CREAT is specified in the flags) be created.
read reads from a file descriptor, such as that returned by open. It will try to read up to *count* bytes from the file descriptor, into the buffer. On success the number of bytes read is returned, where 0 indicates the end of the file. On error, -1 is returned and errno is set to indicate the error.
write writes up to count bytes from the buffer, starting at buf, to the file referred to by the file descriptor fd. We may watch these system calls in order to debug them by using strace. strace prints most of the system calls (or at least how they affect you) that the program makes. On success the number of bytes written is returned. On error, -1 is returned and errno is set to indicate the error.

2.7 Ex1

We will need to explore the memory hierarchy, fill up the cache to demonstrate the different latencies. We will be measuring the average latency time of a single reading operation from a large array. The array will be spread over the caches and the main memory.