Basic OS

Shahzeb Asif

June 17, 2017

References

ThinkOS

A lot of these notes are based on the book at: ThinkOS

GCC

Compiler

The compiler is the tool that takes our code and turns it into machine code that the processor can read.

GCC is a widely used C compiler. Like most other compilers, it does quite a lot more between input and output.

Compilers like GCC will commonly perform the following steps:

- **Preprocessing** - allows you to put in ”settings” for the compiler.
- **Parsing**
- **Semantics** - checking whether all types declared are used correctly.
- **Generation** - creates the machine code.
- **Linking** - link object code from libraries.
- **Optimization** - optimizes code to be very efficient.

Compilers are very complicated pieces of software.

Note that **object code** is just machine code for a part waiting to be linked.

Flags

Here are a few common, useful GCC flags:

- `-o <name>` creates an executable with `<name>`.
- `-c` tells GCC to only create an object file and not an executable.
- `-O, -O1, -O2, ...` is for optimization from safe to riskier.
- `-S` will generate assembly code.

- `-E` gets all the other code required from libraries or other files.

Memory

Before talking about memory, it is necessary to be fully comfortable with binary.

Binary

Here's a very quick refresher:

- n bits can represent 2^n states.
- To convert from decimal to binary, just run through bit places from MSbit to the LSbit and put `1`s where they should go. For example:
  - To get 6_{10} in binary, three binary digits are enough.
  - `100` is too low so we go up.
  - `110` is 6_{10}.
  - After doing this a few times, you'll memorize values.
- To convert from binary to decimal is simple. $p_n * 2^n + p_{n-1} * 2^{n-1} + ... + p_1 * 2^1 + p_0 * 2^0$
  - p_n is the values (0, 1) at the nth place.

Another very useful thing to know is how **floating point** numbers are stored.

It's not too complex. Floating point numbers are usually just stored in something resembling scientific notation but for base 2. An example will help.

$8_{10} = 1 * 2^3 = 1.00 * 2^3$

That value, $1.00 * 2^3$ is stored in binary as: `|[1 - sign] [8 - exponent] [23 - coefficient]|`

But the exponent usually has a bias which gets added to the actual exponent. In 32-bit FP, it is 127_{10}.

And the coefficient only includes the digits after the radix.

That means the 32-bit floating point representation for 8_{10} is: `|0 10000010 00000000000000000000000|`

There are also 64-bit floating point numbers.

RAM

All programs have most of their data in RAM. It is important to distinguish RAM from HDDs or SSDs.

RAM is physically and functionally different from HDD/SSDs. For one, RAM is cleared anytime power is lost.

RAM is also much faster than disk storage.

But that's fairly low level stuff. When we're talking about OSes, something called virtual memory also exists. But before explaining virtual memory, we must explain processes.

Processes

Operating systems make heavy use of processes. A single process runs in its own little world. It has its own virtual memory and share of processor time.

Everything working in an OS is running its own process which probably stems from another parent process.

Using `ps` on a UNIX-y terminal gives a list of processes currently running. Using `-e` flag gives a list of all processes on the OS.

If you use `ps -e`, you'll get a large list back which will have `init` at the top. `init` is the first thing that starts when the OS first starts up.

Some other interesting processes will include:

- `kthreadd` which is responsible for creating ”threads”.
- `kworker` is a worker processes.

Virtual Memory

A process's memory will contain its machine code, static vairables, the **heap**, and the **stack**.

For sensible reasons, I picture lower addresses at the top and higher addresses at the bottom.

The machine code is placed at the very top of the memory. Next comes the static variables. Then the heap which grows downwards. The stack start at the very bottom and grows up.

These components are placed in a program's virtual memory which is all the memory a process can access. This is why a process runs independent of other processes. It literally can't access other processes' memory.

All virtual memory can be translated to physical memory. Usually, this is done by a MCU component called the MMU (memory management unit). The MMU is responsible for quickly translating a virtual address to a real one.

Without going into details, the MMU process can be described as converting relative addresses (virtual memory) to absolute addresses (physical memory).

It's also important to note that there is often much more virtual memory than physical memory.

Memory Management

We have already described how memory works in hardware and at what happens in an OS with memory but we've yet to talk about how memory is managed and tracked.

Tools

We use four main C functions for a whole lot of memory allocation.

- `malloc`
- `calloc`

- `realloc`
- `free`

`malloc` allocates a block of a certain size and returns a pointer to it.

`calloc` is like malloc but zeros each byte.

`realloc` copies a chunk from the old malloc'ed area to the new area.

`free` will free a previously allocated space.

Performance

Performance can be a little tricky to estimate for the memory management tools above. But generally, these guidelines should be useful:

- `malloc` does not depend on size.
- `calloc` depends on the size.
- `realloc` can be slow if the new size is larger than the old size.
- `free` is usually very quick.

Tags

Note that malloc'ed space also has tags attached to it. There is some overhead information that contains the allocated space's size, state, and some other details.

These tags do take up some space. When you malloc a block of size x, it actually takes up more than just x bytes.

These tags also allow the system to store all allocated blocks in a doubly linked list.

Caching

Caching is extremely important for performance in computer.

Blocks

Any time a byte is loaded from memory, it also gets stored in the CPU cache. But it gets stored as a **block** of a certain size unique to hardware. A few neighboring bytes may also be stored in the cache. Next time something needs to be read from memory, the cache will be checked first.

The reason for the cache is that it is much faster to access the cache than it is to access memory. This is how the cache speeds up performance.

Locality

To talk about performance, we must talk about locality first. **Spatial locality** is when a program uses nearby data in physical memory. **Temporal locality** is when a program accesses the same data more than once.

Rates

The **hit rate** is the fraction of bytes that are already in the cache. The **miss rate** is the fraction of bytes that need to be read again from memory.

We can also improve the performance of our programs by keeping the hit rate in mind.

If we decide to use data in ways that allow it to have both types of locality, then it will be faster than one with neither.

For example, iterating over an array of `unit8` will likely have both spatial and temporal locality. The block size of the CPU will likely be much larger than 1 byte so reading consecutive bytes will have lots of spatial locality. And if you perform any actions on the data read, the program will retrieve the data from the cache instead of memory. This means the program will also have temporal locality.

File System

File systems are created to help store files. Files themselves are generally on drives. The drives can be very complicated so we won't go into details.

It is important to know that reading and writing usually doesn't occur in single byte operations but rather larger **blocks** consisting of many bytes.

Performance

IO is generally incredibly expensive compared to the other parts of a program.

This is because writing to an HDD is very slow compared to RAM. SSDs are faster but still slow compared to the processor.

To deal with this slow IO, OS and hardware use a few tricks:

- **Block transfers**.
- **Prefetching** loads a block before it is needed.
- **Buffering** means to write a file to memory only when completely done with it.
- **Caching** a block to internal memory.

These tricks can vastly improve the performance of IO without the user even being aware of it happening.

Metadata

OSs usually have their own way of organizing the data on a disk. This allows the OS to store data anywhere on the disk instead of just sequentially.

This is what the partition format refers to.

Basically, we store small structures that contain information like its owner, date modified, etc.

Files

One important thing to note in UNIX-y systems is that everything is a file. Everything.

This is because a file isn't just a text document, or an image. It's more of an abstraction for a bunch of data.

This has some cool side-effects. If you create a library that works with a file stream, it'll work with local files, terminal pipes, and even things like webcams and other external devices.