EngSci Year 3 Winter 2022 Notes

Brian Chen

Division of Engineering Science
University of Toronto
https://chenbrian.ca

brianchen.chen@mail.utoronto.ca

Contents

Section 1

CSC473: Advanced Algorithms

Subsection 1.1

Global Min-Cut

Given an undirected, unweighted, and connected graph $G = (V, E)$, **return** the smallest set of edges that disconnects G

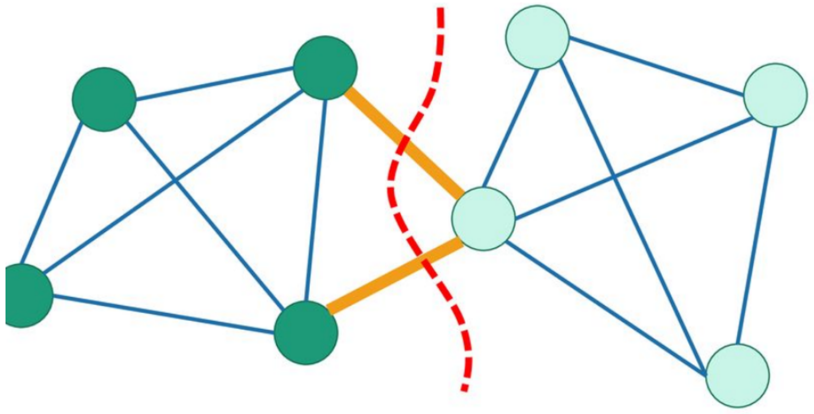

Figure 1. Example of global min-cut. Note that the global min-cut is not necessarily unique

An example of where this may be useful is in computer networks where we can measure the resiliency of a network by how many cuts must be made before a vertex (or many) get disconnected

Lemma 1 If the min cut is of size $\geq k$, then G is k-edge-connected

It may be more convenient to return a set of vertices instead

Definition 1

$$S, T \subseteq V, S \cap T = \varnothing \tag{1.1}$$

$$E(S, T) = \{(u, v) \in E : u \in S, v \in T\} \tag{1.2}$$

The global min-cut is to output $S \subseteq V$ such that $S \neq \varnothing, S \neq V$, such that $E(S, V \setminus S)$ is minimized.

Comment Note that the min-cut-max-flow problem is somewhat of a dual to the global min-cut problem; the min-cut-max-flow problem imposes a few more constraints than the global min-cut algorithm i.e. having a directed and weighted graph as well as the notion of a source or sink.

- **Input:** Directed, weighted, and connected $G = (V, E)$, $s \in V, t \in V$
- **Output** : S such that $s \in S, t \notin S$ such that $|E(S, V \setminus S)|$ is minimized

We can kind of intuitively see that the global min-cut can be taken to the minimum of all max-flows across the graph. So we can take the max-flow solution and then reduce it to find the global min cut.

Question: how many times will we have to run max-flow to solve the global min-cut problem? Naively, we may fix t to be an arbitrary node, then try every other $s \neq t$ to find the $s - t$ min-cut to get the best global min-cut.

We know from previous courses that the Edmonds-Karp max-flow algorithm will run in $O(nm^2) = O(n^5)$, which makes our global min-cut algorithm $O(n^6)$. However, there is a paper recently published which gives an algorithm for min-cut in nearly linear time, i.e $O(m^{1-O(1)}) = O(n^2)$ which gives a global min-cut runtime of $O(n^3)$.

A randomized algorithm will be presented that solves this problem in $O(n^2 \log^2 n)$

Section 2

ECE568 Computer Security

Subsection 2.1

Section 3

ECE353 Operating Systems

Subsection 3.1

Kernel Mode

3.1.1 ISAs and Permissions

There are a number of ISAs in use today; x86 (amd64), aarch64 (arm64), and risc-v are common ones. For purposes of this course we will study largely arm systems but will touch on the other two as well.

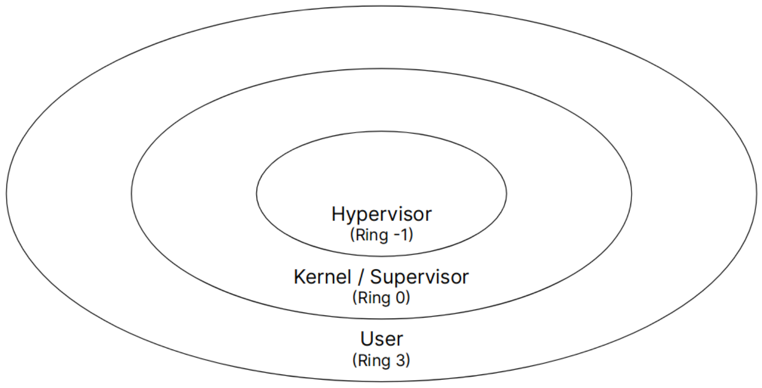

Figure 2. x86 Instruction access rings. Each ring can access instructions in its outer rings.

The kernel runs in, well, Kernel mode. **System calls** offer an interface between user and kernel mode[1].

The system call ABI for x86 is as follows:

[1] *Linux has 451 total syscalls*

Note: API (application programming interface), ABI (Application Binary Interface). API abstracts communication interface (i.e. two ints), ABI is how to layout data, i.e. calling convention

Enter the kernel with a svc instruction, using registers for arguments:

- x8 — System call number
- x0 — 1st argument
- x1 — 2nd argument
- x2 — 3rd argument
- x3 — 4th argument
- x4 — 5th argument
- x5 — 6th argument

This ABI has some limitations; i.e. all arguments must be a register in size and so forth, which we generally circumvent by using pointers.

For example, the `write` syscall can look like:

```
ssize_t write(int fd, const void* buf, size_t count);
// writes bytes to a file descriptor
```

3.1.2 ELF (Executable and Linkable Format)

- Aways starts with 4 bytes: `0x7F, 'E', 'L', 'F'`
- Followed byte for 32 or 64 bit architecture
- Followed by 1 byte for endianness

`readelf` can be used to read ELF file headers.

For example, `readelf -a $(which cat)` produces (output truncated)

Most file formats have different starting signatures or magic numbers

```
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              DYN (Position-Independent
 ↪  Executable file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x32e0
  Start of program headers:          64 (bytes into file)
  Start of section headers:          33152 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         13
  Size of section headers:           64 (bytes)
  Number of section headers:         26
  Section header string table index: 25
```

strace can be used to trace systemcalls. For example let's look at the 168-byte hello-world example

This output is followed by information about the program and section headers

```
0x7F 0x45 0x4C 0x46 0x02 0x01 0x01 0x00 0x00 0x00 0x00 0x00 0x00 0x00
  ↪  0x00 0x00
0x02 0x00 0xB7 0x00 0x01 0x00 0x00 0x00 0x78 0x00 0x01 0x00 0x00 0x00
  ↪  0x00 0x00
0x40 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
  ↪  0x00 0x00
0x00 0x00 0x00 0x00 0x40 0x00 0x38 0x00 0x01 0x00 0x40 0x00 0x00 0x00
  ↪  0x00 0x00
0x01 0x00 0x00 0x00 0x05 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
  ↪  0x00 0x00
0x00 0x00 0x01 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x01 0x00 0x00 0x00
  ↪  0x00 0x00
0xA8 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0xA8 0x00 0x00 0x00 0x00 0x00
  ↪  0x00 0x00
0x00 0x10 0x00 0x00 0x00 0x00 0x00 0x00 0x08 0x08 0x80 0xD2 0x20 0x00
  ↪  0x80 0xD2
0x81 0x13 0x80 0xD2 0x21 0x00 0xA0 0xF2 0x82 0x01 0x80 0xD2 0x01 0x00
  ↪  0x00 0xD4
0xC8 0x0B 0x80 0xD2 0x00 0x00 0x80 0xD2 0x01 0x00 0x00 0xD4 0x48 0x65
  ↪  0x6C 0x6C
0x6F 0x20 0x77 0x6F 0x72 0x6C 0x64 0x0A
```

Listing 1: Note: This is for arm cpus

If we run this then we see that the program makes a write syscall as well as a exit_group

```
execve (" ./ hello_world " , [ " ./ hello_world " ] , 0 x7ffd0489de40
  ↪  /* 46 vars */ ) = 0
write (1 , " Hello world \ n " , 12) = 12
exit_group (0) = ?
+++ exited with 0 +++
```

Note that these strings are not null-terminated (null-termination is just a c thing) because we don't want to be unable to write strings with the null character to it.

```
execve("./hello_world_c", ["./hello_world_c"], 0x7ffcb3444f60 /* 46 vars */) = 0
brk(NULL)                               = 0x5636ab9ea000
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=149337, ...}) = 0
mmap(NULL, 149337, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f4d43846000
close(3)                                = 0
openat(AT_FDCWD, "/usr/lib/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0000C"..., 832) = 832
lseek(3, 792, SEEK_SET)                 = 792
read(3, "\4\0\0\0\24\0\0\0\3\0\0\0GNU\0\201\336\t\36\251c\324"..., 68) = 68
fstat(3, {st_mode=S_IFREG|0755, st_size=2136840, ...}) = 0
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0)
  = 0x7f4d43844000
lseek(3, 792, SEEK_SET)                 = 792
read(3, "\4\0\0\0\24\0\0\0\3\0\0\0GNU\0\201\336\t\36\251c\324"..., 68) = 68
lseek(3, 864, SEEK_SET)                 = 864
read(3, "\4\0\0\0\20\0\0\0\5\0\0\0GNU\0\2\0\0\300\4\0\0\0\3\0\0", 32) = 32
```

Figure 3. A c hello world would load the stdlib before printing...

3.1.3 Kernel

The kernel can be thought of as a long-running program with a ton of library code which executes on-demand. Monolithic kernels run all OS services in kernel mode, but micro kernels run the minimum amount of servers in kernel mode. Syscalls are slow so it can be useful to put things in the kernel space to make it faster. But there are security reasons against putting everything in kernel mode.