Harry Langford
hjel2@cam.ac.uk

Operating Systems Supervision 4:

Some questions are cryptic. You need to figure out exactly what they are asking you about and answer that.

The Tannenbaum (2003) book was recommended and you should look through it. Although it better for "looking up concepts" rather than reading from front to back.

There is a second year course on distributed systems which builds on Operating Systems.

Access Control Lists and Capabilities:

An access control matrix is a very sparse boolean matrix which holds which permissions each subject (program) has for each file.

This can be simple RWX or more complicated like delete/append/access metadata (this is system specific).

Access Control lists are the columns of the access control list. (for each object which subjects).

Capabilities are the rows – which are stored with subjects.

An access-control list is more like an entry list.

A capability is like a key/keycard/ticket.

An example of a thing like capabilities are browser cookies (they hold identity information and access rights). In GitHub, access tokens are like capabilities.

It's easier to create fine-grained control with capabilities. Since you can request capabilities when needed and give them away when you no longer need them.

With capabilities it's easy to pass them onto other actors on the system. With access control lists this can't work. You have to change the access control list at the object. Capabilities are more dynamic.

File systems are more controlled – access goes by the operating system. This is not distributed. This is pretty good.

Capabilities must be non-forgable. This can be difficult.

Often capabilities are valid only for a certain time. IE you include a timestamp with the capability.

Capabilities are not necessarily hashes. You can encode data (ie deffie-hellman).

These security concepts exist in other concepts as well.

Principle of minimum privilige:

Capabilities are nicer for minimum-privilige.

Principle of minimum privilige comes in delegating subtasks.

ACL's are significantly simpler: they're not usually the best but they are easier than capabilities and this is why UNIX does not have capabilities: "UNIX is not really a success story when it comes to access rights".

There are multiple access groups in a system. But one file can only grant access rights to one single group in UNIX (since it only has 3 bits and we do **not** want variable length access rights). IE if I have teachers and students then I can't specify rights for both groups on one file – one will have to be universe.

Owners do not always have full access to the files. Although owners can always change the metadata to give themselves access. This allows users to prevent other processes from doing things to files they do not want.

A d bit at the start of an inode means that the file is a directory. Directories have different meanings of rwx.

Authentication

Learn how UNIX does authentication.

/etc/passwd is readable by anyone.

/etc/shadow is only readable by the root.

Protection

You protect CPU time by using a pre-emptive scheduler.

The second resource is anything stored in volatile memory. The only thing you need to know about this is “paging” and “segmentation” (base-limit solution).

Non-volatile data is protected using ACL’s.

You can protect the hard disk by only making hard disk accessible through system calls (don’t allow direct access).

The Operating System runs in Kernel mode and so has full access to do anything (bypassing authorisation).

Compile time:

Diadvantages:

- You can’t run the same program twice at the same time (since they’d both expect to use the same memory).
- You can’t run any two addresses if their memory addresses overlap at all.
- In order to compile a process successfully you have to know all the other processes and what memory they use.

Load Time:

Disadvantage:

- You have to load the entire process in (not just individual pages).

A TLB is a cache for page table entries to recently referenced pages.

If you try to read from an empty pipe then the system call will block until something has been written into it.

The same is true th eother way. The write will block until there is a space in the pipe.

A shell is a user interface that allows the user to run processes and interact with the operating system.

```
while True:
        print(prompt)
        input = read()
        cmd, args = parse(input)
        pid = fork()
        if pid == −1:
                print('error something went wrong')
        else if fork() == 0:
                execve(cmd, args)
        else:
                int status
                wait(pid, &status)
                // waits for a child process to die
```

If fork = -1, something went wrong

Fork returns a signed integer to the parent (the process id of the child). when you call wait you pass pid and this waits for the process with that pid to die.

Fork passes 0 to the child.

fork() creates a child process which “looks exactly like the paret process” and has the same virtual address space and data.

execve() flushes the entire virtual address space of the process and loads in the executable specified (in this case cmd) with arguments specified. This executable then starts executing.

fork, execve and wait are the system calls here.