1 2015 Paper 2 Question 3

https://www.cl.cam.ac.uk/teaching/exams/pastpapers/y2015p2q3.pdf

(a) What is an access control list?

An access control list is a list of users and their priviliges for a specific file. The access control list is associated with each file allowing maximum flexibility. There are three priviliges: read, write and execute – each of which have a single bit. This allows different users to have different priviliges for the same file.

Initially on all accesses of files, the process trying to access does not have priviliges. This causes a segment fault. At this point we load in the access-control-list and check to see whether the process should have access. If it should then access is granted and if it should not then the process is terminated. Access can be granted or elevated only by the owner.

(b) What is a capability in the context of access control?

Capabilities are unique and unforgeable identifiers for files. If a process has it then the file knows that it should have the access granted by this capability. However, capabilities can be passed to other processes. This leaves the possibility of processes having access that they were not directly granted. Capabilities and what accesses they should give are stored in the files inode. This means that the priviliges they give can be rapidly updated.

(c) How is file access control in Unix implemented, and what simplifications are made over one of the general mechanisms you have described above?

UNIX uses access-control-lists. However, rather than giving access rights to users, it gives access rights to groups of users. This means that on large systems we do not have large amounts of duplicate data across files. There are three categories: owner, group and universe. The owner has full access rights to the file. Groups have whatever access rights the owner has assigned to their group. The universe is anyone who's not the owner and is not in a group. The universe typically has the lowest access level. Each file grants rights to the owner, one group and the universe. For each category there are three bits: Read, Write and Execute (RWX). This means that a fixed amount of bits are needed for each access control list, greatly simplifying the system: holding variable-length access-control lists for each system would make memory management much more complicated – variable length inodes. Some systems support more than one group – as long as the number of groups supported is less than a constant the system will work. Note that there are only 8 possible combinations of access and so having more than 8 groups per file makes no sense. Groups can be stored on other pages. So groups only need to be listed once. This decreases the amount of duplicate data and saves space.

Note that UNIX does not use capabilities: but Linux does.

(d) What is the principle of minimum privilege and which approach, access control lists or capabilities, lends itself better to supporting a minimum privilege regime?

The principle of minimum privilige means you give each process the minimum privilige that it needs to do whatever it should do. This way the process is physically unable to do most of the things it should not be able to do and if it tries to do them, it will raise a semgent fault – which can be dealt with accordingly.

Access Control Lists support the principle of minimum privilige better than capabilities.

The main reason for this is that capabilities are held by programs and can be passed around, if one rogue program gets a capability then the whole system can have the same capability even if it they were not directly given the capability. This is not the same with access control lists.

Since capabilities risk "leaking" access to files, I believe that access-control-lists better conform to the principle of minimum privilige.

2 2014 Paper 2 Question 3

https://www.cl.cam.ac.uk/teaching/exams/pastpapers/y2014p2q3.pdf

One goal of a multiuser operating system is to protect each user's information and activity from damage caused by accidental or deliberate actions of other users of the system.

(a) Describe a mechanism that operating systems use to reduce the opportunity for a user process to prevent another user's process from making progress. In your description include any particular hardware features that are relied upon.

Pre-emptive scheduling will interrupt every process eventually. A good pre-emptive scheduler will assign the process which has just been moved from the running state to the ready state a lower priority. This prevents denial of service. Pre-emptive scheduling relies upon hardware support for context switches.

(b) Describe two alternative mechanisms that operating systems could use to reduce the opportunity for a user process to access or corrupt the information being used by another user's process. In your descriptions include any particular hardware features that are relied upon.

If we use virtual addressing then processes are physically incapable of accessing other processes address spaces. This relies upon hardware support for traversing page tables and hardware support for converting virtual addresses to physical addresses.

We must also empty the TLB after context-switching processes. This relies on hardware support from the TLB to rapidly remove all data.

(c) Describe how an operating system might attempt to ensure that long-term user information (that is, information which exists beyond process execution) is not interfered with or misused by other users. Your description should be clear about when actions are performed and the resources they consume.

We can give files access rights to ensure that long term information is not interfered with. Each file should have an owner, group and universe access rights. These can be chosen so that users files give full access to the users processes and restricted access to other users processes. For example giving the owner RWX (read-write-execute), giving no groups access and giving the universe no access.

When a process tries to access a new file, it will cause a segment fault (since the process has not been granted access – although it may have sufficient priviliges). At this point, the CPU can check the access bits to see whether the process trying to access the file has sufficient priviliges. If it does then access is granted and tha page is loaded into memory.

(d) To what extent are the mechanisms described above useful in single user systems?

The mechanisms described above are still useful in single user systems. The difference is that now rather than preventing different users corrupting other users files we are preventing processes corrupting processes being run by the same user and preventing processes occupying all the CPU time. The measures described in part (a) will ensure that tasks cannot deny service to other processes.

If we did not have (a) then the below code would loop infinitely and prevent all other programs from running.

```
void main()
{
        start:
        goto start;
}
```

Virtual memory (discussed in (b)) is still necessary since we do not want processes to interfere with other processes address space irrelevant of whether the other process belongs to another user or not. An analagous argument can be made regarding flushing the TLB.

Access rights are still relevant in a single-user system. You do not with every process to have full read-write-execute access on all programs.

(e) How do operating systems ensure that they are not themselves overly-restricted by these mechanisms?

The Operating System is a high priority task and so is not interrupted by the scheduler when running. The Operating System does not need to read-write or execute user-files and so the permissions for those are not relevant and will not restrict the Operating System.

We have high hardware support for all of these processes. This means that execution is not slowed down significantly.

3 2011 Paper 2 Question 4

https://www.cl.cam.ac.uk/teaching/exams/pastpapers/y2011p2q4.pdf

(a) In the context of memory management:

(i) What is the address binding problem?

The problem of deciding *where* the physical addresses of variables in programs should be.

(ii) The address binding problem can be solved at compile time, load time or run time. For each case, explain what form the solution takes, and give one advantage and one disadvantage.

Program C:

```c
#include <stdio.h>

void main()
{
        int i = 0;
        scanf("%d", &i);
        if (i == 0)
        {
                int j = 1;
                int k = 2;
                int l = 3;
        }
}
```

- If we decide where to allocate memory at compile time, then we can run algorithms determining the optimal place to allocate it. This means that our program will be faster than other methods (although the compile time is far longer). Note also that we only run this algorithm once.

  However, if we decide where to allocate memory at compile time then we have to allocate all the memory the program *might* need throughout execution. This is necessary even if the program never uses some of these variables; so if we allocated memory at compile time, when running program C, we would allocate memory for j, k and l even if $i \neq 0$ at execution. If we decide where to allocate memory at compile time then we have to use absolute addresses. This is impractical in general purpose systems: we will not be able to run two programs which expect to use the same absolute addresses.

- If we allocate memory at load-time then we do not suffer from "processes expecting to use the same addresses".

  However, we must parse through the whole program and decide where we should place every variable which could exist in the program. This is very expensive and a large overhead before running any program. While we can run algorithms to make memory locations chosen more optimal, this has to be done every call. If we ran Program C, note we would still have to allocate memory for j, k, l.

- Allocating memory at runtime means we do not have to allocate any memory unless we use it. If we ran program C, then we would not have to allocate memory for j, k or l. With hardware support, we can automatically translate between program addresses and real addresses. This means that we do not have to change the addresses in the program itself at runtime.

  This scheme also does not suffer from "multiple programs expecting to use the same memory" either and we cna dynamically allocate more memory when needed. However we cannot run advanced optimisation algorithms to choose the best location for the memory addresses.

(iii) Under which circumstances do external and internal fragmentation occur? How can each be handled?

External fragmentation happens when placing data of different sizes straight into memory. An example of this is segmentation – where many different sized data leave many small gaps between them which are too small to be used. This can be lessened by using a heuristic method of choosing where to put data: ie best-fit or worst-fit. External fragmentation can be handled by compaction: moving data around in memory to make the spaces contiguous.

Internal fragmentation is caused by placing data of many sizes into uniformly sized blocks of data. Some of these blocks will be too large for the data placed inside them rendering part of the block unusable. This can be handled by using smaller blocks. Although using smaller blocks means more memory accesses: there is a trade-off.

(iv) What is the purpose of a translation lookaside buffer (TLB)?

Caching pages:

Memory accesses are long and slow while the TLB is far faster to read from. TLB's have expensive and fast hardware so that searching the TLB is significantly faster than searching memory. Modern TLB's can have a 98% hit rate – meaning that 98% of pages we try to find will be in the TLB. This dramatically speeds up memory accesses.

(b) Describe how UNIX handles user authentication.

On startup, you load the kernel from disk and start executing it. This then runs the login program. This copies some programs and forks some stuff. Each process then outputs login and waits for the user to input their username. It then takes a password and checks whether it hashes to the same value as the password stored on disk.

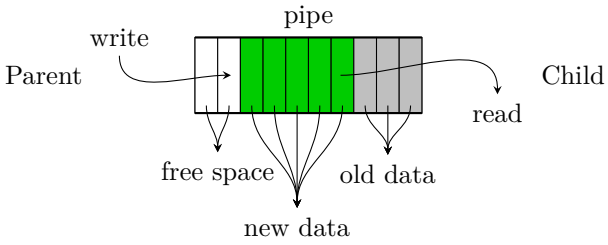
4 2009 Paper 2 Question 3

https://www.cl.cam.ac.uk/teaching/exams/pastpapers/y2009p2q3.pdf

(b) In the context of the UNIX operating system:

(i) What is a pipe? What is it used for? How does it operate? Use a diagram to illustrate your answer.

An ordinary pipe is a special type of FIFO file which parent processes share with child processes. This allows the parent to write into one end of the pipe and the child to read from the other end of the pipe. This is a very basic type of inter-process communication. Note that unnamed pipes are unidirectional: the parent can send data to the child *or* the child can send data to the parent. To achieve bidirectional data transmission we simply use two pipes.



Note there is another type of pipe: named pipes (FIFO's in UNIX).
Named pipes are also FIFO files – except they are bidirectional; they can be written to and read from by many different processes without parent-child dependencies; and they are persistent: name pipes are not automatically deleted after the processes stop communicating – they must be explicitly deleted.

(ii) What is the shell? Describe its operation in pseudo-code, giving special emphasis to any system calls invoked.

A shell is a command interpreter. It is a program which allows users to manipulate the file store, read from and write to and execute files. Shells allow users to use relative paths. This makes manipulation of the file system easier.

Here is pseudo-code describing what a shell does:

```
#include <all>
/*
this is not an actual library: rather it signifies that we should
include all libraries in the path and builtin libraries etc.
In a real system many of these would be loaded when called
to be more efficient -- however the effect is the same.
*/

void main()
{
        string instruction;
        function(string[]) command;
        string[] args;
        bool successful;
        interpreter:
        printf(">");
        scanf("%s", &instruction);
        command = get_function_to_execute(instruction);
        if (command == NULL)
        {
                printf("No such command");
                goto interpreter;
        }
        args = get_arguments(instruction);
        if (fork() == 0){
                execve(command, args);
                exit();
        }
        goto interpreter;
}
```