20HT – 1DV512 – Operating Systems
Individual Assignment 1

1

Student: Sami Mwanje

ID/mail: mm223kk@student.lnu.se

Assignment date: 2020-11-15

Hand in date: 2021-06-18

1. **Which programming languages were used to implement the Unix kernel? Why?** The first version of the UNIX was created by Ken Thompson with the support of Dennis Ritchie and some other members at Bell Laboratories 1969. The programming language originally used for the UNIX kernel was the Assembly language. After some years and version of work with the UNIX operating system Ritchie and Thompson rewrote the UNIX-programming language to C. The programming language C was especially developed to support the UNIX operating system. Using C instead of Assembly made the UNIX operating system easier to implement and more flexible for further enchantments. Today the Unix kernel contains about 10 000 lines of C code and 1000 of Assembly code. To understand why the whole UNIX-code has not been rewritten to C we must look closer at the 1000 Assembly lines, 800 of these 1000 Assembly code lines contains hardware instructions that are not possible to rewrite to C.[23]

2. **Is the Unix kernel a complete operating system? How would you compare these concepts?** An operating system manages software, hardware and provides services for programs running on the system. Looking closer at the UNIX kernel we can see that it manages all of these. First UNIX was a research tool but with time it became more of an own operating system. Today there are even UNIX-based operating system such as Linux. Just like in today's operating systems users can execute programs in an environment (shell), which is the whole meaning of an OS. Even system and user processes can be found inside the UNIX operating system. The process that the UNIX-kernel goes through to execute a user/system process can differ from modern operating systems, still the UNIX-kernel does what is required in order to be called a complete operating system. It even controls and coordinates the computer's hardware in order to protect and control system resources for the applications running on the system. [45]

3. **If you decided to re-implement the Unix kernel using the modern programming languages and tools, which language(s) would you select and why?** I would prefer to re-implement the Unix kernel using Java as programming language. When trying to implement the Unix kernel using Java, I would probably notice that it would be a more complicated task then implementing using C and Assembly. Primarily Java needs a JVM (Java Virtual Machine) that translates each bytecode to the current platforms language in order to run. With short words, JVM is a simulated computer running on the system, therefore there must exist an operating system that runs on the current machine in order to run the JVM over it. This will then contribute to a loss of processing power and efficiency. A solution that can be applied is using hardware that is designed to run an operating system implemented with Java. As we can see the re-implementation will be more complicated when using a modern programming language as for example Java. Therefore, I would probably go back to using C and assembly as these programming languages are more closely related to modern computers hardware and platforms.

4. **Are new running processes (programs) started from scratch in Unix? What is their relationship to the already running processes?** No, newly running processes (programs) are not started from scratch in Unix. Every new process is created with a system function called **fork()**. Fork is the first phase in order to create a new process. The process that makes the fork()-call to the system makes an exact copy of itself and split into two processes. The process that made the fork()-call is now called the parent process and the new duplicate is now called the child process. The child process has a copy of the address space of the parent process, this allows the parent process to easily communicate with the child process. After the fork()-call is done the child process has new data (program) loaded into it using the system call **exec(),** and the parent processes does another system call named **wait()**. The **wait()**

call makes the parent process to keep waiting until the child has finished the **exec()** function. After **exec()** function has replaced the child memory space with the new program (text/data), the child can go a separate way from its parent process. However, all the child processes to a parent must first be terminated with the function **exit()** before the parent process can be terminated. There is no detectable sharing of primary memory between the child and the parent process, though if the parent and the child process has the same read-only segments, only one of these will be loaded into the primary memory. The current read-only segment comes from a text table with the location of the current read-only segment in the secondary memory. There is a flag/count in the primary memory which has the number of current process sharing the current segment.

5. **What are main classes of input-output devices supported by Unix? To which class would a Solid-State Drive (SSD) belong? How about a Brain-Computer Interface (BCI)?** All devices on a UNIX-system are split into two separate systems (classes), block I/O devices and character I/O devices. This split is often based on how and how often the devices are going to be used. Block devices are typically devices that allow random access from the system/user. These are fixed-sized blocks of data such as hard disks, floppy, flash memories, CD-ROMs and Blu-rays. Character I/O devices are typically mice, keyboards and line printers. The biggest difference between these and the I/O block is that they require to be accessed serially. A Solid-State-Drive (SSD) would fit in the block class because of that it obviously requires blocks, and the block class is typically made for file systems. A closer look proves that a disk drive may take use of both the character class and block class. When speaking of an SSDs tremendously speed it may arguably not fit in any general category. A solution to this is setting up a device that have its own buffers or borrow buffers from the I/O block. Another device that can take use of this solution is the Brain-Computer-Interface(BCI). It would probably be connected to the computer using a USB. Mainly it would fit in the character I/O class because of its non-block and no-file system character. The problem here is that the BCI would be dependent of high-speed transmission which a character I/O class may not handle smoothly. Therefore,

the second solution may come in place here. Where the BCI either have its own buffer or "borrow" block I/O buffers for a while.

6. **Does the Unix file system design use disks C: and D: (as in DOS or Windows)? Can a Unix system have several disk drives attached at the same time?** A file is some related information defined by a creator. Files are mapped to physical driver by the operating system. For an operating system to understand how files are going to be accessed and manipulated by users and programs a **files system** is needed. Files systems must be designed so they can be accessed quickly and efficiently. In windows a mounted disk device will be represented as a letter C: D: and so on. In UNIX a mounted disk device will be represented as a folder/directory, in fact every device and file on the computer will be organized in a folder/directory on UNIX. The directories are organized in a tree-like structure which is referred to as "file system". Directories are equal to files that cannot be edited by users. The directory tree starts with a directory called "root" which is represented with a "/". All other "files/devices/directories" are "children/sub directories" to the root "/". This makes any installed disk able to appear anywhere in the files system and the user can install as many disks as he fits. In most of the UNIX-like systems the "folder" representing what is displayed as "C: ,D: ..." in Windows can be found in the directory "root/dev/device-id".

7. **What is an *i-node*? What are its contents and their purpose? Are i-nodes used for implementation of directories (if yes, how)?** The UNIX file system separates the disk in four parts (blocks). The first block is empty and addressed 0, the second block is called "super-block" and contains information like disk boundaries. This block is addressed 1. The third block contains a so called "i-list" and is addressed 2. The last block (blocks) is free for files etc. To answer what an "i-node"

Sami Mwanje
mm223kk@student.lnu.se

is we must first look at the block that contains an i-list. An “i-list” is a data structure that contains elements which are called i-nodes. Each i-node is defined with a 64-bit structure with of file/directory definitions. The “index” for each i-node is called an “i-number”, which can be accessed by the Unix kernel when accessing an i-nodes contents. I-nodes carries a various number of contents like user and group ID, addresses of the file-data on the disk, time when file was last modified etc. A typically i-node contain 15 pointers. 12 of these 15 pointers points to an address on the disk where the file data is located. If the file size is between 4 and 48 kB the data can mainly be accessed direct from the i-node and not from the data blocks. This tells us that a file is stored in two different locations. In the data block where the data of a file can be found, and in the i-node where the information about the file. The remaining 3 pointers points to indirect blocks which can be addresses that points to other addresses. The Unix users are not aware of the i-nodes or i-numbers. Only the content that the i-node carries. As said earlier I-nodes carries a various number of contents like user and group ID, addresses of the file-data on the disk, time when file was last modified etc. But the question is where is the name/path of a file stored? To answer this, we must first take a closer look on directories. Just like files the content of a directory can be in data blocks and has a i-node representation. Directories have a different structure from plain files, therefore there i-node type is not the same as the i-node type of a classic file. Unix early directories had 16 bytes available, which 14 of those 16 could be used for names of the files and the remaining two for i-number. Today we can find directories with 255 bytes which gives the user the option to use more advanced names. To clear things up we could say a directory is a file which holds the names and the path of the files linked to it.

8. **What are the primitive file system operations supported in Unix?** For the UNIX file system to communicate and execute system tasks some functions must be used. These are so called primitive operations (functions). The UNIX file system uses a various number of primitive operations, **open, create, seek,**

read, write and unlink. The difference between these primitives and user primitives is that these mostly affects the i-nodes. For example, **create** that creates a new i-node entry and **open** that converts a path name into a i-node table entry. The **unlink** primitive is used when removing a file. A file only exists if there is a directory pointing to its i-node. What unlink does is that it removes the pointer between a directory and a file, so when there is no pointer from a directory to the i-node(file) left, the file gets removed and does no longer exist in the file system.

[1] https://blog.desdelinux.net/wp-content/uploads/2018/02/unix.jpg
[2] Abraham-Silberschatz-Operating-System-Concepts-10th-2018, C.1 Unix History
[3] Thompson - UNIX Implementation (1978).pdf
[4] Abraham-Silberschatz-Operating-System-Concepts-10th-2018
[5] Thompson - UNIX Implementation (1978).pdf
These sources are used all over the assignment.