Introduction

This research paper is divided into three parts:

1. How UNIX views process, user and program.
2. I/O system
3. UNIX file system

Process control:

In the UNIX system, a user executes programs in an environment called a user process. It encompasses the program code, data, stack, and registers necessary for the program's execution. Whenever a system function is required, the user process calls the system as a subroutine and at this point there is a distinct switch of environment. The user process **may** execute from a read-only text segment, which is shared by all processes executing the same code. This provides efficiency benefits as the original copy resides on disk so you don't need to swap it back to disk. Also this benefit saves a lot of primary memory space, but this optimization comes into play only when you have more than enough memory to load multiple processes concurrently as simultaneous execution of processes is uncommon. So this becomes ironic as a feature designed to reduce memory usage comes into play most significantly when there is already plenty of memory to spare.
The info of read-only text segments is stored in a text table which is nothing but a table that initially stores the location of the segment in disk, when the segment is loaded then further entries like primary memory location and count of process sharing this segment is added.The entry will be removed when the count value becomes zero. A user process also has private read-write data in its segment. And it's very rare that the system will store its data in the user data segment. Lastly you also have a process table too.

1.1 Process creation:

Processes are created using fork. **Child process does not share primary mem and writable data segment(s), the only things that are shared with the parent is read-only text segment and list of all the files opened by the parent before fork.** A process executes by changing the current text and data from the segment for new text and data however this doesn’t change the process, it’s just that old data is lost.

1.2 Swapping

The major data associated with a process are swapped to and from secondary memory, as needed. Usually the **user data segment** is stored in a contiguous location to reduce **swapping latency.** If the process grows then a new piece of primary mem is allocated,content of old mem is copied, old mem is freed and the tables are updated. If enough space is not available in the primary mem then the process is swapped onto disk. There is a separate process in the kernel, the **swapping process**, that simply swaps the other processes in/out of primary memory.
There are two algorithm involved in swapping process:

I. **Which processes that are swapped out are to be swapped in?:** The one with the longest time out is swapped in first.(decided by secondary storage residence)

II. **Which processes are to be swapped out of primary mem?:** Processes which are waiting for slow events(not running or waiting for disk I/O) are swapped out.

1.3 Synchronization and Scheduling

Process synchronization is accomplished by having processes wait for events. Events are represented by arbitrary integers. Similarly, signaling an event on which many processes are waiting will wake all of them up. Race conditions may occur due to the lack of memory association. The scheduling algorithm selects the highest priority process, favoring system processes over user processes. **But the scheduling algorithm has a negative feedback character. If a process uses its high priority to hog the computer, its priority will drop. At the same time, if a low-priority process is ignored for a long time, its priority will rise.**

I/O System:

The I/O system is broken into two completely separate systems:

a. the block I/O system
b. character I/O system.

Devices are characterized by a major device number, a minor device number, and a class (block or character).
Major device no: The major number identifies the driver associated with the device(floppy disk).
Minor device no: It is used to identify the specific device(i.e., the first floppy would have minor 0, the second would be 1).
Device Class (Block or Character):
Devices are broadly classified into two types: "block" and "character."

Block devices are typically used for storage (like hard drives), and data is read or written in blocks.Character devices are used for input or output character by character (like keyboards or printers).

2.1 Block I/O system

The model block I/O device consists of randomly addressed, secondary memory blocks of 512 bytes each. The blocks are uniformly addressed 0, 1, . . . up to the size of the device. The block device driver has the job of emulating this model on a physical device.**The block I/O devices are accessed through a layer of buffering software.** The system maintains a list of buffers (typically between 10 and 70) each assigned a device name and a device address. This buffer pool constitutes a data cache for the block devices. **On a read request, the cache is searched for the desired block. If the block is found, the data are made available to the requester without any physical I/O. If the block is not in the cache, the least recently used block in the cache is renamed, the correct device driver is called to fill up the renamed buffer, and then the data is made available.**

2.2 Character I/O system

All the devices which do not fall in the **block I/O model** are character I/O systems. Ex: paper tape and line printers. I/O requests from the user are sent to the device driver essentially unaltered however the implementation of these requests is up to the device driver.

2.3 Disk Drivers

Disk drivers are implemented with a queue of transaction records. **Each record has some data associated with it like, read/write flag, transfer byte count, primary mem address and secondary mem address.** Swapping is achieved by passing the above info to the swapping device driver. Block I/O is also implemented by passing the same info along with the request to fill and empty the buffers. The character I/O interface to the disk drivers creates a transaction record that points directly into the user area. This allows for a direct connection between the transaction record and the user's data, making it easier to associate the transaction with a particular user or process. **By implementing the general disk driver, it is possible to use the disk as a block device, a character device, and a swap device.**

2.4 Character list **(Just go through this)**

A character list is a queue of characters. One routine puts a character on a queue. Another gets a character from a queue. Storage for all queues in the system comes from a single common pool. Putting a character on a queue will allocate space from the common pool and link the character. Getting a character from a queue returns the corresponding space to the pool.

The File system:

In UNIX a file is a 1D array of bytes. **Directories are simple files that user cannot write.** The UNIX file system is a disk data structure accessed completely through the block I/O system. The file system breaks the disk into four self-identifying regions. The first block (address 0) is unused by the file system. It is left aside for booting procedures. The second block (address 1) contains the so-called ‘‘super-block.’’ This block, among other things, contains the size of the disk and the boundaries of the other regions. Next comes the i-list, a list of file definitions. Each file definition is a 64-byte structure, called an i-node. The offset of a particular i-node within the i-list is called its i-number. The combination of device name (major and minor numbers) and i-number serves to uniquely name a particular file. After the i-list, and to the end of the disk, come free storage blocks that are available for the contents of files. **The free space on a disk is maintained by a linked list of available disk blocks.** I-node has 13 disk addresses, 10 are direct blocks, and the rest are indirect, double indirect and triple indirect block each.**A logical directory hierarchy is added to this flat physical structure simply by adding a new type of file, the directory.**

3.1 File system implementation.

Since in the UNIX file system the i-node defines a file, the implementation of the file system is based around access to the i-node. So the system maintains the list of all active i-nodes. As a

new file is accessed, the system locates the corresponding i-node, allocates an i-node table entry, and reads the i-node into primary memory. When the last access to the i-node goes away, the table entry is copied back to the secondary store i-list and the table entry is freed. **The user process accesses the file system with certain primitives. The most common of these are open, create, read, write, seek, and close. There is a type of unnamed FIFO file called a pipe. Implementation of pipes consists of implied seeks before each read or write in order to implement first-in-first-out.**

3.2 Mounted file systems:

The UNIX file system starts with some designated block device, and the root of this structure is the root of the UNIX file system. A second formatted block device may be mounted at any leaf of the current hierarchy. This logically extends the current hierarchy. To implement mount **a mount table is maintained that contains pairs of designated leaf i-nodes and block devices**. When converting a path name into an i-node, a check is made **to see if the new i-node is a designated leaf. If it is,** the i-node of the root of the block device replaces it. Allocation of space for a file is taken from the free pool on the device on which the file lives. Thus a file system consisting of many mounted devices does not have a common pool of free secondary storage space. This separation of space on different devices is necessary to allow easy unmounting of a device.