UNIX:

HISTORY:

1. 1969 UNIX developed (Ken Thompson).

2. 1972 UNIX rewritten using C (Dennis Ritchie).

3. 1976 UNIX (v6) released for commercial use.

4. 1978 UNIX (v7) released (ancestor of most UNIXs).

5. 1978 The UNIX Support Group (USG) assumes control of UNIX

6. 1978 3BSD (Berkeley Software Distributions) adds virtual memory, demand paging, and page replacement to UNIX (v7)/(32v).

7. 1979 4BSD adds support for internet protocols, improved terminal line editing features, the C shell, VI, and Pascal and LISP compilers.

8. 1980 Xenix (MicroSoft) derived from UNIX (v7).

9. 1982 System III adds some real-time features and a set of software tools to UNIX (v7).

10. 1983 System V improves on system III.

11. 1983 4.2BSD improves on 4BSD.

12. 1984 System V (v.2.4) adds virtual memory with copy-on-write paging and shared memory.

13. 1984 Sun OS derived from 4.2BSD.

14. 1988 4.3BSD improves networking and TCP/IP performance.

15. 1993 4.4BSD includes X.25 networking, POSIX compliant interface, a new file system, a version of NFS, enhanced security, and improved kernel structuring. This was the last release from BSD.

16. 1993 Solaris 2 derived from System V (v.4.4).

17. 1993 Novell purchases Unix System Labs (originally USG).

DESIGN GOALS:

- The unique design of UNIX can be attributed to:

  1. The first version was designed and built by two people (the small number of people involved led to its coherence – like Pascal).

  2. The people who built UNIX also used it (which mean it is simple to understand).

- The design goals where:

  1. Keep the operating system simple.

  2. Keep the operating system general. For example:

     a) The same system calls are used to read (or write) files, devices, and interprocess message buffers.

     b) The same naming, aliasing, and access protection mechanisms apply to data, directories, and devices.

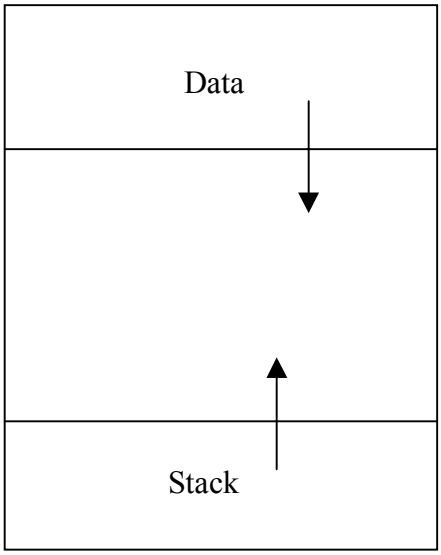
     c) The same mechanism is used to trap software interrupts and processor traps.

  3. Create an environment where large tasks can be created from existing small tasks (using pipes and filters).

PROCESS CONTROL:

- Processed are represented by various control blocks which are stored in kernel space (protected space). This includes the process control block (sometimes referred to as an image).

- The storage component is divided into 3 parts: the text (program code), stack, and data. The stack and data sections are stored in the same address space. The text segment is stored in a different region (segment) of memory.



CPU SCHEDULING:

- CPU scheduling is a time-shared priority scheme (if all processes have the same priority the approach reduces to a round-robin scheme).

- The scheduler uses negative feedback to dynamically adjust process priorities (the more time a process accumulates the lower its priority goes and vise versa).

- Process aging is used to prevent starvation.

- The time quantum on newer versions of UNIX is 0.1 second. Priorities are recomputed every second.

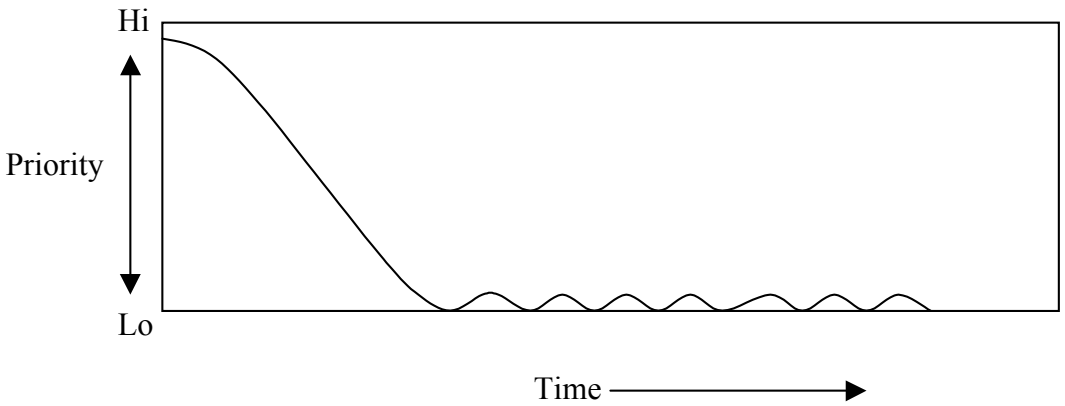
- Processes cannot preempt other processes. Once a process has the CPU it keeps it until its time quantum is up or it requires I/O.

- When a process requires I/O it goes to "sleep" and waits. When the system completes the I/O, it wakes the sleeping process (or processes), which is (are) then put back on the ready queue. When the process next runs, it collects the I/O data.

- Note that the effect of negative feedback and process aging could result in the following effect:

  1. A long running CPU hog program starts up with high priority.
  2. As it runs and accumulates CPU time, its priority continually drops.
  3. As its priority drops, other higher-priority processes begin to consume more and more of the CPU cycles until our process gets none.
  4. At this point due to aging, our process priority level is increased until our process again gets some CPU time.
  5. Go back to step 2.

- In a heavily loaded system, we can easily end up in a situation like:



Where a long running and CPU intensive process sinks to the bottom of the priority pool and waffles around down there with just enough priority to stay alive.

What if this process is also a memory hog? What would the effects of swapping and/or paging (caused by this process) have on the overall system performance?

Remember, to reduce the performance degradations caused by paging, that it is best to let a memory hog program finish first.

Is UNIX designed to handle these types of processes?

MEMORY MANAGEMENT - SWAPPING:

- Early versions of UNIX used contiguous allocation for memory and swap space.
  - A process larger than the physical memory could not run at all.
  - If a large process was scheduled to run, many other processes would have to be swapped out.
  - Memory and swap allocation was done "first-fit".
  - Swap decisions were made every 4 seconds by the swap scheduler (a medium-term scheduler). A process was more likely to be swapped out if it was idle, been in memory a long time, or was large. A process was more likely to be swapped in if it was small or had been in swap space a long time.
- Newer versions use paging methods for memory and swap space management. However, swapping (as implemented above) is still used as a secondary means of memory management.
- Most versions of UNIX use demand paging. Demand paging is done in a straightforward manner (sort of):
  1. When a memory section is needed that is not already in memory, a page fault occurs.
  2. If the page had been previously loaded, but was marked for removal (at the time it was no longer needed). It can be reused.
  3. In many cases most pages (belonging to a process) are loaded when the process loads (prefetching).
  4. If a page is to be loaded from disk, it must be locked in memory. This prevents some other processes' page fault from removing it before it can even be used.
  5. If a page is being used, it must also be locked in memory.

6. There are also checks to insure that the number of valid pages for a process do not fall too low (a global vs local page replacement check) as this would tend to increase the number of page faults generated.

7. There is also a check to prevent a process from gobbling up too much memory.

8. If the scheduler decides that the system is overloaded, whole processes will be swapped out to relieve the problem.

FILE SYSTEM:

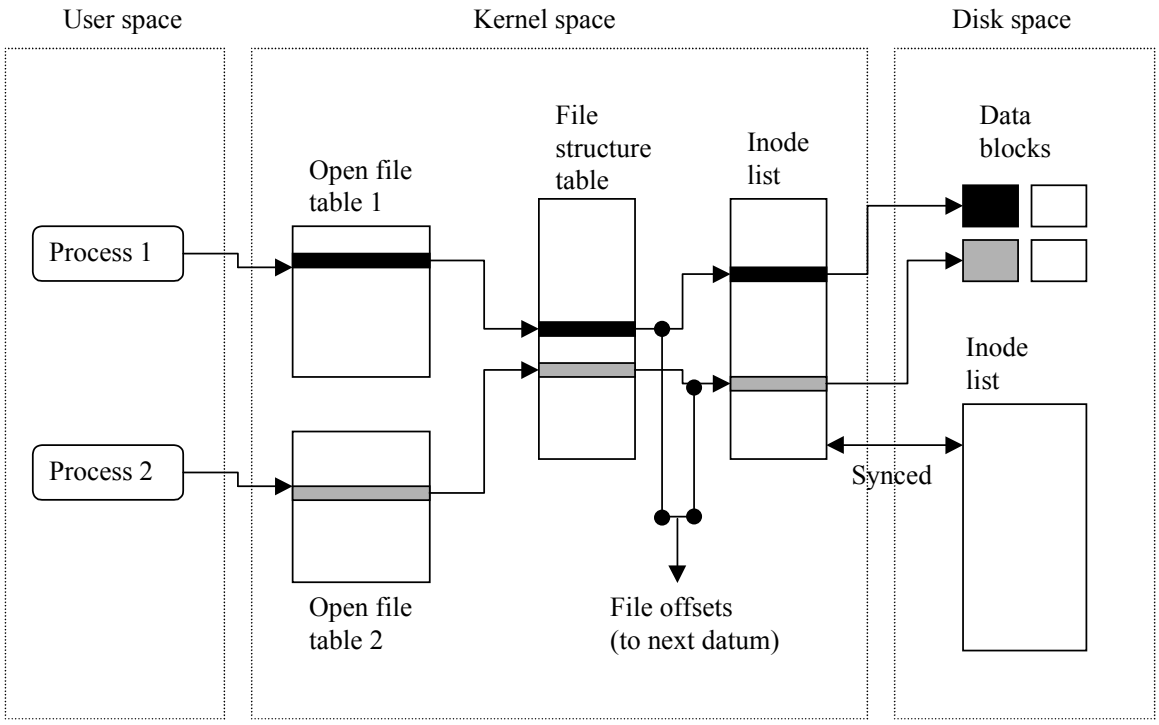
- UNIX uses Inodes to represent files. UNIX uses Inodes to represent directories (which are just files – with a "special" structure).

- UNIX uses a linear search to find files in the directory.

- UNIX treats hard links as directory entries.

- UNIX treats soft links as "pointers" to another location in the directory. When traversing a soft link, UNIX prevents infinite loops from occurring by counting the number of soft links encountered during the traversal (the limit is 8).

- UNIX uses kernel-level tables (1 table per process) to monitor open files. The table points to a file-structure table. The file-structure table points to the in-memory version of the Inode (for the file) and keeps track of the current location (to be read/wrote) in the file.



- The open file table has a fixed length. Therefore, there is some limit to the number of open files.

- UNIX file system can span many physical drives.

- To incorporate a new file system, it must be mounted (integrated into the root file system). The Inode of any file system which includes other mounted file systems has a bit set. This bit tells the system to search the mount table for the device number of the mounted device. The device number is used to find the “root” Inode for the mounted file system.

- UNIX records free blocks in a linked list. Free blocks are pushed into the front of the “free list” and removed from the front of the “free list” as needed. Thus, after time the file system will become fragmented.

- The superblock (contains the size of the disk, the boundaries of file systems, and the overall layout of the file system) of each mounted file system is kept in memory (for speedy accesses). Every 30 seconds the superblock is written to the disk (for longer term, non-volatile storage). If the system crashes, we must examine all of the blocks on all file systems on boot-up to rebuild the superblock. This problem still exists today.

- Variations of free block management and superblock management do exist. However, in some cases (4.2BSD) the variation decreases (wastes) drive capacity to improve on access rates or reliability.

I/O SYSTEM:

- The I/O system is designed to hide the peculiarities of I/O devices from the kernel.

- 3 types of I/O:

  1. Block devices – Block devices are directly addressable using a fixed block size (disks and tapes are block devices).

  2. Socket devices – Socket devices are network devices.

  3. Character devices – Character devices are everything else (terminals, line printers, /dev/mem, /dev/null, etc).

BLOCK BUFFER CACHE:

- Block devices use a block buffer cache.

  1. When a block is wanted from a block device, the block buffer cache is searched first.

  2. When a block is written to a block device, the block buffer cache is searched first. If the block is not found, a free block is used. In either case, dirty bits are used to indicate a change to the buffer.

  3. Periodically, blocks with dirty bits are force written to the block device.

- Increasing the size of the buffer cache, usually improves system performance.

- Use of the buffer cache also allows the disk system to better schedule (to minimize head movements) writes. However, reads are not affected.

IN A NUTSHELL:

- UNIX was not designed to be used on a large scale commercial computing type of machine. Hardware performance was not one of the design precepts.

- UNIX was designed to cater to multiprogrammed interactive processes. Not to long running, CPU hungry applications.