LINUX:

HISTORY:

1. 1991 Linus Torvalds writes a small (i.e. minimal functionality and only supported Minix file system) self-contained kernel for the 80396.

   Linus originally started hacking the kernel as a pet project, inspired by his interest in Minix, a small UNIX system developed by Andy Tanenbaum. He set out to create, in his own words, "a better Minix than Minix". And after some time of working on this project by himself, he made this posting to comp.os.minix:

   > “Do you pine for the nice days of Minix-1.1, when men were men and wrote their own device drivers? Are you without a nice project and just dying to cut your teeth on a OS you can try to modify for your needs? Are you finding it frustrating when everything works on Minix? No more all-nighters to get a nifty program working? Then this post might be just for you.
   >
   > As I mentioned a month ago, I'm working on a free version of a Minix-lookalike for AT-386 computers. It has finally reached the stage where it's even usable (though may not be depending on what you want), and I am willing to put out the sources for wider distribution. It is just version 0.02...but I've successfully run bash, gcc, gnu-make, gnu-sed, compress, etc. under it.”

2. 1994 Linux (v1.0) adds networking, a better file system, SCSI support, paging support for swap files, floating point emulation, system V styled interprocess communication, and some support for dynamically loadable kernel modules.

3. 1995 Linux (v1.2) adds support for more hardware, support for the 80386’s virtual 8086 mode (allows DOS emulation), and improved/expanded network support.

4. 1996 Linux (v2.0) adds support for non-intel machines, for SMP, improved TCP/IP performance, additional network protocols, improved memory management, better support for

dynamically loaded kernel modules, and support for kernel level threads.

- Versions r.x.y, where x is an even number, are stable versions, and only bug fixes will be applied as y is incremented.

  So from version 2.0.2 to 2.0.3, there were only bug fixes, and no new features.

- Versions r.x.y, where x is an odd number, are beta-quality releases for developers only, may be unstable and may crash, and are having new features added to them all the time.

  So version 2.1.127 would be an unstable development version.

- From time to time, as the current development kernel stabilizes, it will be frozen as the new “stable” kernel, and development will continue on a new development version of the kernel.

  So version 2.2.y would be the next stable release kernel?

DESIGN GOALS:

- The design of LINUX can be attributed to:

  1. The first version was designed and built by one person. Later versions include components that were submitted by "anyone".

  2. LINUX is governed by the General Public License. Therefore, no proprietary code/algorithms is/was used.

  - Both of the above suggest a hodge-podge of an operating system using less than state-of-the-art methods.

- The design goals where:

  1. To provide an UNIX-like operating system for PC's.

  2. Initially, LINUX was concerned with squeezing as much UNIX functionality as possible from the PC.

  3. Speed and efficiency of the OS.

  4. Standardization is also important – and becoming more so as LINUX is ported to additional platforms.

  5. Be POSIX compliant.

OS DESIGN:

- The LINUX system is composed of 3 types of code:

  1. The kernel – A single monolithic binary program. Responsible for maintaining information about the state of the system and for management of the system resources

  2. System libraries – Define a standard set of functions through which applications can interact with the kernel. Part of the intent of the libraries is to provide the user/application with OS functionality without giving the user/application root-level privileges.

  3. System utilities – Provide specialized management tasks.

- LINUX uses dynamically loaded Kernel modules – sections of the kernel which can be loaded or unloaded on demand (on demand kerneling – my term ☺).

  1. Kernel modules run in privileged mode.

  2. Kernel modules ease the task of developing drivers for new hardware (a common occurrence with PC's).

  3. Kernel modules allow one to tailor a LINUX system to a given PC.

OS DESIGN (cont):

- LINUX used 3 techniques to manage dynamic kernel modules.

  1. Module management – Requests a region of kernel memory for the module, loads the module into the kernel memory, and updates the module's symbol table (adjusts it to match/reference the kernel).

  2. Driver registration – When a module is loaded, the kernel calls its start-up code. This code, in turn, tells the kernel what (new) features it is providing. When a module is removed, the kernel calls its shut-down code. This code tells the kernel what features are "going away."

  3. Conflict resolution – Since LINUX was originally intended to run on PC's, it is possible that several modules may provide interfaces to the same hardware. Device conflict resolution is managed by a first-come-first-serve device reservation method. If a device has already been reserved by another module, it is up to the rejected module to decide how to proceed.

PROCESS CONTROL:

- Process control under LINUX is similar to that of UNIX. A new process is created with the FORK system call and a new program is run with the EXECVE system call.

- UNIX processes typically include 2 types of information: The process identity, environment, and the process context.

  1. UNIX process identity includes the process ID (PID) and process credentials (owner). LINUX adds a third item, the process personality (which flavor of UNIX to conform to).

  2. The UNIX process environment is similar to the LINUX process environment.

  3. The UNIX process context is similar to the LINUX process context.

- Normal kernel code is nonpreemptive (by other processes). If another process does attempt an interrupt, it is serviced after the kernel code completes.

- Normal kernel code can only be interrupted by:

  1. Interrupts – Which require the kernel code to wait for the interrupt service routine to finish.

  2. Page faults – Which require the kernel code to wait (and other processes can execute) for the page transfer.

  3. Kernel call to the scheduler – Which may interrupt the kernel by forcing a rescheduling.

- The critical sections of interrupt service routines are protected by, sort of, disabling interrupts. LINUX splits the interrupts into 2 sections:

  1. Top-half – A normal interrupt service routine (only interruptible by higher priority interrupts).

  2. Bottom-half – Interrupt operations that are not time critical and therefore interruptible. For example, in keyboard interrupt, the change of LED lights is done in its bottom half not in its interrupt (top-half). In the network drivers, the received buffer is copied to the upper network layer by its bottom half as well.

- This design allows the high priority code of interrupts to be handled fast, the lesser priority code of interrupts to be handled when possible, and allows all of this to within the kernel.

CPU SCHEDULING:

- Three schedulers: One is a time-sharing algorithm. Two are a "soft" real-time schedulers. Part of a process's identity is the scheduling class, which defines which scheduler to use for that process.

1. Time sharing: This algorithm is a credit based algorithm. Each process is given a certain number of credits. When a new task is to be run, the process with the most credits runs. As a process runs, every time a timer interrupt occurs, the process losses one credit. When its credit count reaches 0, it is suspended and another process is chosen to run.

   If no runnable processes have any credits, Linux recredits all (even the non-runnable ones) of the processes according to:

   ```
   credits = (held_credits / 2) + priority
   ```

   This method favors processes with high priority and processes that have not run in a long time (typically processes which are I/O bound).

2. Real-time: To be POSIX compliant, two real-time algorithms are implemented, FIFO and round-robin.

   - The FIFO real time scheduler always runs the oldest process with the highest priority first. This process runs to completion or until the process generates an I/O interrupt.

   - The round-robin scheduler manages real-time processes of equal priority with a round-robin scheduler.

MEMORY MANAGEMENT:

- PHYSICAL MEMORY – The primary memory manager is the page-allocator, which is responsible for allocating and freeing memory pages. The allocator uses a buddy heap to track available memory (smallest allocatable block of memory is a physical page).
    - All kernel memory requests/needs are static (allocated at boot-up) or are dynamic by a page allocator (the kernel has its own pool of memory).
    - kmalloc – A variable length memory allocator.
        1. Memory is permanently allocated (until explicitly freed).
        2. Memory allocation is an atomic (uninterruptable) operation. Memory is either allocated or not (if none is available).
- VIRTUAL MEMORY – The virtual memory manager is responsible for maintaining the address space available to each process. It creates pages of virtual memory on demand and manages the loading and swapping of these pages from and to the disk.
    - Several types of virtual memory are implemented:
        1. Backed by nothing - ***Demand zero*** memory. Memory that is backed by nothing (initially filled with zeros).
        2. Backed by file – Memory that is backed by a file. Read and writes to this memory are synced to the file on the drive.
        3. Private – Memory that is "local" to a process. Can be read or write memory.
        4. Shared – Memory that is "global" to all/several processes. Can be read or write memory.
    - When a new process is started (execve) a new virtual address space is created for that process. In the case of a fork the process shares its parent's memory.

- LINUX uses paging methods for memory and swap space management (kernel pages are never swapped).

- LINUX uses a modified version of the second-chance page replacement algorithm.

- Paging supports "swap" devices and "swap" files (much slower). The mechanism uses a best-fit algorithm to write out pages to contiguous disk blocks.

- Technically, LINUX does not swap at all. LINUX uses the disk swap space for the storage of pages (which is ***paging*** not ***swapping***).

- Not all of the kernel's virtual memory space is reserved exclusively for the kernel.

  - vmalloc – A kernel virtual memory space allocator.

    1. Allocates an arbitrary number of physical pages and maps them into a single region of kernel memory.

    2. Allows allocation of large contiguous blocks even if there are not enough free adjacent memory pages to meet the request.

- See: http://victoria.uci.agh.edu.pl /doc/khg/chapter2.7.html for more details.

PROGRAM EXECUTION:

- Since the format for executable files under LINUX has changed (from a.out to ELF), LINUX maintains several possible binary loader programs. When an ***exec*** call is made each loader routine is tested to see if it can load the program.

- When a program is loaded, only the virtual page table is set up for the program. Only when a page fault is generated, is anything loaded into memory (pure on demand paging).

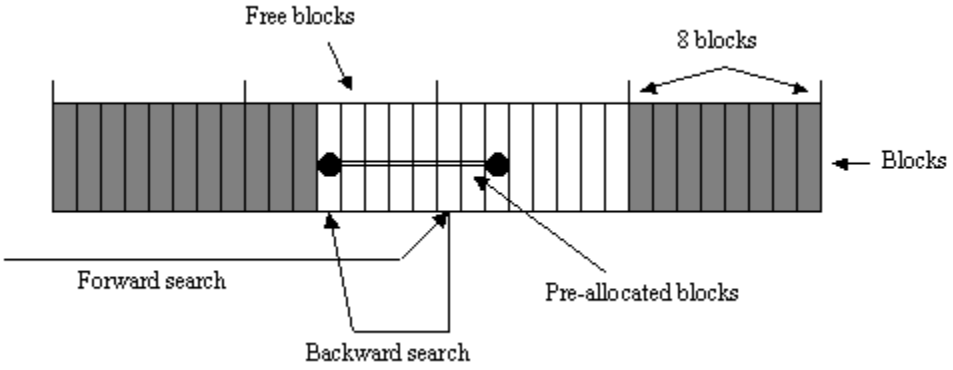
FILE SYSTEM:

- LINUX uses 2 file systems, *ext2fs (etc)* and *proc*.

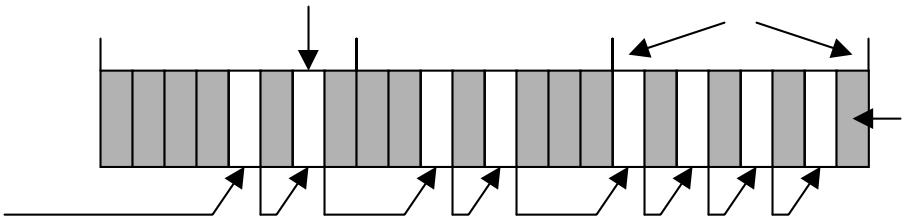
1. ext2fs – The standard disk file system (much like BSD's fast file system). Uses 1k (or 2k or 4K) blocks. For performance reasons, the disk scheduler always attempts to read/write blocks in clusters larger than 1 block.

    - When allocating space for a file, LINUX attempts to do a contiguous allocation (to reduce fragmentation). This is done by searching the entire free list for a free byte (8 contiguous free blocks).

        1. If a free byte is found, search backwards (from the start of the free byte) until a used block is found. Then preallocate the 8 blocks (following the used block) to the file (when the file is closed the unused blocks are returned to the free list).



        2. If one is not found, free blocks are sequentially allocated from the free list.



2. proc – The proc file system does not persistently store data at all, it is only used as an interface to some other functionality.

   - Each directory (in the proc file system) corresponds to a running process. The directory name corresponds to the process ID (PID).

   - In LINUX, the proc file system also contains extra directories and text files, which document the state of the system. This design allows the LINUX "ps" command to be implemented as a user-level process which simply scans the text files in the proc file system. In UNIX, "ps" is a privileged system process.

- **Additional LINUX file systems.**

  EXT3 - a minimal extension to the existing EXT2 filesystem to add journaling.

  EXT2 is a fairly well proven filesystem and there are a lot of users out there who have very large EXT2 filesystems. However, They take a long time to fsck if the filesystem gets corrupted. So the objective with EXT3 was this simple thing: availability. When something goes down in EXT3, we don't want to have to go through a fsck. We want to be able to reboot the machine instantly and have everything nice and consistent.

  XFS - On May 1 2001, SGI made available Release 1.0 of its high-end XFS file system for Linux. XFS, widely recognized as the industry-leading high-performance filesystem, provides rapid recovery from system crashes, high throughput, and the ability to support extremely large disk farms. XFS is the first journaled filesystem for Linux available today that has a proven track record

in production environments since late 1994. (http://www.oss.sgi.com/projects/xfs/index.html)

JFS - IBM's journaled file system technology, currently used in IBM enterprise servers, is designed for high-throughput server environments, key to running intranet and other high-performance e-business file servers. IBM is contributing this technology to the Linux open source community with the hope that some or all of it will be useful in bringing the best of journaling capabilities to the Linux operating system. (http://oss.software.ibm.com/developerworks/opensource/jfs/)

Reiserfs - Originally designed by Hans Reiser, Reiserfs carries the analogy between databases and filesystems to its logical conclusion. In essence, Reiserfs treats the entire disk partition as if it were a single database table. Directories, files, and file metadata are organized in an efficient data structure called a "balanced tree." This differs somewhat from the way in which traditional filesystems operate, but it offers large speed improvements for many applications, especially those which use lots of small files. (http://www.linuxplanet.com/linuxplanet/tutorials/2926/4/)

ETC – Several file systems are being developed for Linux, See: (http://linux.usc.edu/LDP/links/devel.html#kernel)

I/O SYSTEM:

- The LINUX I/O system is designed to look like the I/O system of UNIX.

- 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).

IN A NUTSHELL:

- LINUX was not designed to be used on a large scale commercial computing type of machine. Even though hardware performance was one of the design precepts, the underlying influence of UNIX cannot be forgotten (forgiven?).

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

- kmalloc and vmalloc do not have “man” pages. One has to read the code to figure out how to use these (probably because they were intended to be used only by kernel/device driver developers?).