Contents of Lecture 12 on Linux

- Minix
- Linux

Minix

- Minix was an educational operating system released in 1987.
- Initially it had about 12,000 lines of C code and 800 lines assembler.
- The author, Prof. Andrew Tanenbaum, rejected many requests for enhancements to keep it small.
- Quickly more than 40,000 persons used Minix.
- Starting with Minix 3.0 from 2004 its focus has changed to reliability instead of "educational".

First Linux Versions

- Linus used a Minix machine to develop his Linux 0.01 kernel and announced its existance firstly to the Minix community on August 25th, 1991.
- Originally Linux was written for the Intel 386.
- Linux is **not** written in ISO C but rather in GNU C, ie using GCC's extensions to ISO C.
- The kernel can be compiled at least by Intel's compiler in addition to GCC.

Linux Version 1.0 1994

- Linux version 1.0 was released in 1994 and consisted of 165,000 lines of code.
- Ext2 file system
- Memory mapped files
- Networking with sockets and TCP/IP
- Many additional device drivers

Linux Version 2.0 1996

- Linux version 2.0 was released in 1994 and consisted of 470,000 lines of code.
- Support for other architectures than X86, e.g. PowerPC
- Support for 64-bit architectures
- Support for multiprocessors, using a giant-lock

Kernel Overview

- System call interface
- Kernel components
  - Process management
  - Memory management
  - I/O
- Interrupts and dispatcher

Three Classes of Threads

- Real-time FIFO
- Real-time round robin
- Timesharing
- A thread has a priority in 0..139 with 0 being the top priority

Scheduling Real-Time Threads

- Linux provides two classes of **soft** real time threads with priorities in 0..99
- Linux implements the POSIX P1003.4 extension to UNIX.
- No deadlines can be specified
- Real-time FIFO have highest priority
- Only preemptable by other real-time FIFO threads with higher priority
- Real-time round-robin have a time-quanta
- Real-time round-robin are preemptable by the system clock.

Scheduling Timesharing Threads

- The time quantum is specified as the number of clock ticks the thread may execute
- A clock tick is 1 ms
- Each CPU has its own **runqueue**
- Each runqueue has two arrays of 140 elements
- There are two pointers **active** and **expired** which point to one of these arrays
- A thread whose time quantum is expired is moved to the array pointed to by **expired**
- When all threads are expired, the two pointers are swapped

Priorities

- The scheduler takes a thread from the lowest active queue
- Threads at a higher (ie lower number) priority are given a larger time quantum
- For example: a priority 100 thread may run 800 ms before it must wait for all other processes quanta to expire while a priority 139 only may run 5 ms

Scheduler Heuristics

- Each thread has a **static** and a **dynamic** priority
- The purposes of the dynamic priority are:
  - increase the priority of interactive threads
  - reduce the priority of CPU-bound threads
- The dynamic priority is a ”bonus” which is added to the static priority
- The dynamic priority is in the range -5..+5

Multiprocessor Scheduling

- As is commonly used (eg Mach 3.0 as we saw earlier) the kernel uses affinity scheduling
- There are system calls to specify that a thread should run on a particular CPU

Booting Linux

- After the Power-On-Self-Test the Master Boot Record (ie the first sector) of the boot device is read and the boot loader, e.g. GRUB, on the active partition is fetched which fetches the kernel.
- The startup code of the kernel is written in assembler and performs:
  - identifies the CPU type
  - disables interrupts
  - enables the MMU (memory management unit)
  - calls the **main** C function of the kernel
- **main** continues the initialization and allocates data structures and probes devices

Dynamically Loading Device Drivers

- Linux, and many other OSes including MS-DOS, can load device drivers dynamically while the kernel is running
- This is convenient but unpleasant in security sensitive environments such as banks
- At a bank which uses dynamically loaded device drivers, a system administrator might be able to insert a module which accidently (or intentionally) corrupts the system.

The First Processes

- After initialization the kernel "manually" creates the first process (manually since it cannot create a process using fork yet)
- Process 0 then mounts the root file system and programs the real time clock
- It then forks to create the **init** process and the **page daemon** process
- The init process either enters single-user mode by forking a shell or multiuser, and forks the program **getty** for each attached terminal
- Getty makes initializations and then prints **login:**. After somebody has written a user name it execs the **login** program which reads and checks the password
- If the password is correct login execs the shell specified in **/etc/passwd** for the user

Linux Memory Management

- A process has the three usual UNIX segments:
  - Text
  - Data
  - Stack
- Recall that data with static storage duration is by default initialized to zero
- Also recall that such data is initialized when the program is started — to avoid writing lots of zeroes in the executable file
- The kernel has a **zero page** which all page table entries of such data pages refer to, and which is copied-on-write when such data is modified

Linux Memory Zones

- Three types of physical memory zones:
  - Pages that can be used for DMA: **ZONE_DMA**
  - Normal pages: **ZONE_NORMAL**
  - Temporarily mapped pages at high addresses: **ZONE_HIGHMEM**
- The details of each zone is specific to the architecture and differs between X86 and Power for instance
- On X86 DMA uses the first 16 MB
- The zones use the buddy memory allocator described earlier and allocate a number of full pages, ie the queue at k holds blocks of 2^k pages

Linux Memory Usage

- Three kinds of usage:
  - By the kernel: always in memory
  - Memory map: also always in memory: see below
  - The rest: including text, data, stack, page tables

Data Structures

- A 32 byte **page descriptor** contains
  - a pointer to the address space the page belongs to,
  - pred and succ pointers to make up a list (eg of free pages in the buddy allocator)
  - some other attributes
- **mem_map** is an array of page descriptors (less than 1% of all memory)
- The mem_map is equivalent to the **coremap** in Lab 3.
- A **node descriptor** is used for **NUMA** multiprocessors (NUMA = non-uniform memory access time, ie a interconnection topology that is more complex than a bus)
- The purpose of the node descriptor is to help the kernel to allocate memory close to where a thread will execute to reduce cache miss performance penalties

More on Data Structures

- Earlier kernel versions used three-level page table
- With 2.6.10 a four-level page table was introduced
- Recall that the kernel itself is always in RAM
- To allocate memory for a kernel module (which can be of any size) the buddy allocator is very useful

Page and Paging Caches

- The **page cache** holds file blocks which either
  - have recently been used, or
  - predicted to soon to be needed
- The size of the page cache is determined dynamically
- The page cache competes for pages just as the processes' address spaces
- With this approach the Linux kernel can allocate pages to where they are most useful
- The **paging cache** is the set of user pages which are on their way to the swap (but may be reclaimed before being written)

The Slab Allocator

- Recall that the buddy allocator easily causes internal fragmentation: if you need 33 pages, you must request 64 pages
- The **Slab allocator** allocates memory using the buddy allocator
- Each slab is used for a certain type of object (with the same size)
- When an object is needed from a slab, it is removed
- When all objects taken from a slab are deallocated, the slab is returned (and the memory can be used eg for some other object type)

The **kmalloc** and **vmalloc** Functions

- When the kernel needs some memory it can request them using **kmalloc** which itself is implemented upon the slab allocator
- The **vmalloc** allocator is used for allocating virtual addresses which not need to be contiguous in RAM

Address Spaces

- The address space of a process consists of a number of memory areas
- All pages in a memory area are consecutive and have the same protection
- Examples of regions are: text, data, stack, a memory mapped file
- Each area is described by a **vm_area_struct**
- The areas of a process form a linked list sorted by virtual address
- When the number of areas exceeds 32, a tree is used instead of a list
- Attributes of an area include: pageable or not, growth direction (down for stack and up for data), read/write protection, and private/shared

Private vs Shared Areas

- An area can be either private to a process or shared with others
- At fork, the kernel copies the list of memory areas but the child's refer to the same page tables
- For copy-on-write, the pages are marked as readonly and when a write occurs the kernel sees that the memory area has write permission (ie if it has that) and copies the page and page table and mark the entries as read/write
- **Swap** or **backing store** for a memory areas depends on which areas it is

Backing Store

- The **text** areas uses the executable file as backing store, ie when a text page is paged out it is simply dropped on the floor and when needed next time it is fetched from the executable file
- a **memory mapped file** not too surprisingly uses the file
- Stack and data areas use swap space when a page is paged out
- The swap space is allocated when needed
- To find the swap space an attribute in the memory area is used
- On top of the list of memory areas a struct **mm_struct** holds additional information about all memory in an address space

Paging in Linux

- A Linux process is in memory if both of the **u area** and the page tables are in memory, ie no pages are necessary since they will be fetched when needed
- When process 2, the **page daemon**, wakes up it checks if there are too few free pages and if so, it will look for pages to take from processes
- Taking a page means scheduling it for being written to the swap area
- The swap area is either the swap partition or a normal file used for swapping.
- A swap partition is faster due to no indirect blocks and more efficient I/O
- A swap area uses a bit map to keep track of free pages

PFRA: The Linux Page Frame Reclaiming Algorithm

- This is the page replacement algorithm in the Linux kernel
- There are four classes of pages for the page daemon:
  - **unreclaimable:** pages which can never be paged out
  - **swappable:** must be written to swap before being reused
  - **syncable:** must be written if they were modified
  - **discardable:** can be reclaimed directly
- The page daemon is called **kswapd**
- The init process (number 1) starts one page daemon for each memory node (recall a memory node is for NUMA architectures)

Page Daemon: One per Memory Node

- Usually about 32 pages are reclaimed each time it is woke up
- The page daemon reclaims easy pages first:
  - Discardable and unused pages are immediately moved to the zone's freelist
  - Not recently referenced pages with an assigned backing store using an approach similar to the clock algorithm
  - Shared pages which no process seems to be using
  - Ordinary not-shared user pages (ie modified or without backing store)
  - Other pages are skipped: if used in DMA transfer, shared and used, locked
- Shared pages require more work to page out since the page tables of all areas sharing the page must be updated

Active and Inactive Lists

- Each page is on either an **active** or **inactive** list
- When the page daemon finds a page whose referenced bit was zero that page is moved to the inactive list
- Instead of the normal clock algorithm's two states (referenced true or false), Linux uses four states combining the referenced and an active flag

I/O in Linux

- Linux is similar to other implementations of UNIX with respect to I/O
- Each I/O device is assigned a file name in **/dev**
- For example, a line printer might be called **/dev/lp** and can be accessed using normal system calls to open, write to, and be closed in order to write a file to it
- On some systems the following might work:

```
$ cp radix.c /dev/lp
```

- Devices are classified as either **block special files** or **character special files**

Special Files

- A block special file represents a device with addressable blocks such as a hard disk
- A character special file represents devices with a byte stream of input or output, such as a keyboard or network interface
- Each special file is assigned a **major** and **minor** device number
- The major device number identifies which device driver is used
- The minor device number identifies one of similar devices
- The system call **ioctl** controls numerous aspects of different special devices

The Generic Block Layer

- To avoid as many block special file accesses as possible, a cache of blocks is maintained
- Before the Linux version 2.2 kernel there were two separate page and block caches but they are now unified
- Disk writes therefore go to the cache and not to the disk
- Twice per minute are dirty blocks written to the disk

The Linux I/O Scheduler

- The disk I/O scheduler is a version of the scan algorithm
- For each disk there is a list sorted on block number
- When the scheduler is about to issue an I/O operation, requests for adjacent blocks are merged
- In addition to the main list, there are two more lists for ensuring that a read request must wait at most 0.5 s and a write request at most 5 s

Line Disciples

- Usually a process wants line editing
- A **tty_struct** contains information eg on how delete or newline characters should be interpreted
- This is called **cooked mode**
- Eg editors on the other hand want deal with every key stroke
- For this purpose a terminal can be set up in a so called **raw mode**

Linux Kernel Modules

- To avoid enforcing static linking of all device drivers, Linux can dynamically load kernel modules
- While usually used for device drivers, they can be used for any kernel extension such as a performance measurement tool
- To insert a module the command **insmod** is used
- The first step to do is to **link-edit** the module: the addresses of the symbols used by the module from the rest of the kernel must be set to the actual values (for the module, they are just undefined symbols before the module is inserted)

More about Inserting Kernel Modules

- After link-editing, the resources needed by the module must be examined to check that they are available (including interrupt request level)
- Interrupt vectors are set up to map the interrupt to code in the module
- A device driver table entry for a new major device number is set up to refer to code in the module
- Then the module is invoked to let it initialize itself
- After this, the module is a part of the running Linux kernel
- Other modern versions of UNIX have similar features