Embedded Linux

Babak Kia
Adjunct Professor
Boston University
College of Engineering
Email: bkia *-at-* bu.edu

ENG SC757 - Advanced Microprocessor Design

Linux

- Linux has come a long way since its humble beginnings in 1991
- Today Linux supports a very wide range of platforms, from Embedded Systems based on ARM, PowerPC, Intel, and Hitachi microprocessors to name a few, all the way up to workstations, servers, and clusters
- It also served as a launch pad for the open source movement, and consequently lead to great interest from academia and business alike

What is Linux?

- Linux is free open source operating system which is fully featured, portable, and extremely versatile
- It runs on everything from PDAs to the largest Mainframes
- Unlike traditional proprietary software, Linux is developed by a multitude of developers across the world
- People often (and mistakenly) use the term Linux to refer to one of three disparate concepts:
  - A Linux Distribution
  - A Linux System
  - The Linux Kernel
- Our focus is primarily on the Linux Kernel, and therefore the term Linux refers to the Kernel itself

Why Linux?

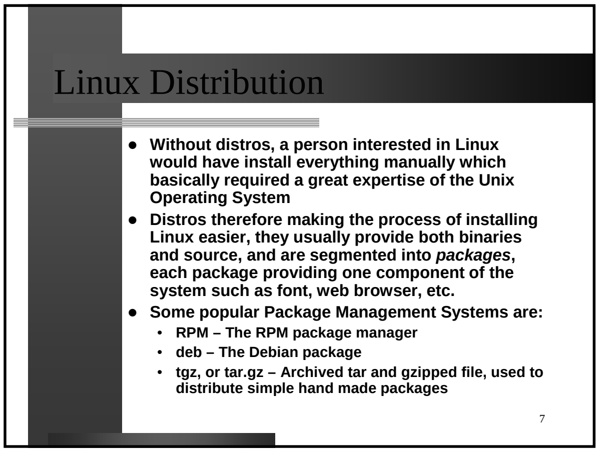
- Due to its open source nature, Linux has a *highly qualified* code base
- The Kernel can be *very small*, it could fit onto a single 1.4MB floppy disk drive, while including all the fundamental operating system tasks!
- It is *highly portable*, it is available for almost every microprocessor system in existence today
- It is *highly supported*, it draws on the open source community across the globe for both development and support
- It supports a *multi-user* environment with a built in capability to concurrently execute applications belonging to 2 or more users
- Supports *multi-processor* systems
- Well documented. The source code is available!

What is uClinux

- uClinux is not Linux, it is a variant of it which runs on processors which lack memory management
- Without memory management, there is no differentiation between *user space* and *kernel space*, and therefore all applications run at Privilege Level 0
- Without memory management, all code runs in a flat memory space, and therefore doesn't require a virtual memory subsystem
- Therefore in this configuration all processes have direct access to memory and I/O resources, and device drivers are not necessary

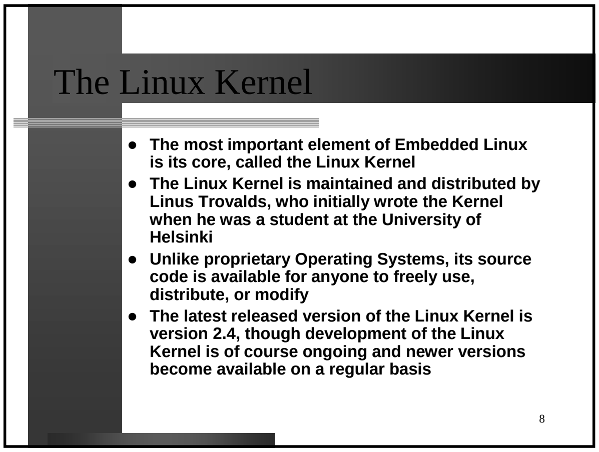
What is a Linux Distribution?

- A *Linux distribution* is a collection of software components, including the Linux Kernel itself, as well as the GNU toolchain (compiler, linker, etc.), and a number of free and open source software such as Emacs, X11, FTP, etc.
- There are many companies involved in creating distributions, such as RedHat, SUSE, and Mandriva, and there are many community projects dedicated to creating distributions (or *distros*) such as Debian and Gentoo Linux
- There are over 300 active Linux distribution projects in existence today



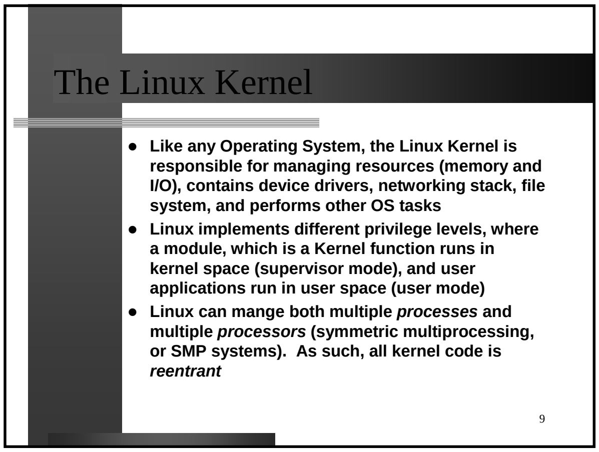
Linux Distribution

- Without distros, a person interested in Linux would have install everything manually which basically required a great expertise of the Unix Operating System
- Distros therefore making the process of installing Linux easier, they usually provide both binaries and source, and are segmented into *packages*, each package providing one component of the system such as font, web browser, etc.
- Some popular Package Management Systems are:
  - RPM – The RPM package manager
  - deb – The Debian package
  - tgz, or tar.gz – Archived tar and gzipped file, used to distribute simple hand made packages



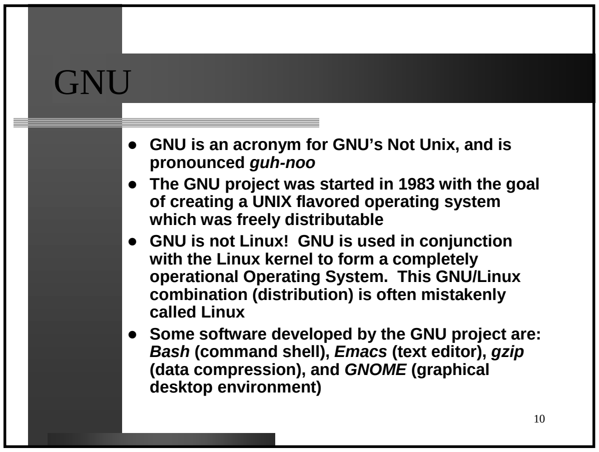
The Linux Kernel

- The most important element of Embedded Linux is its core, called the Linux Kernel
- The Linux Kernel is maintained and distributed by Linus Trovalds, who initially wrote the Kernel when he was a student at the University of Helsinki
- Unlike proprietary Operating Systems, its source code is available for anyone to freely use, distribute, or modify
- The latest released version of the Linux Kernel is version 2.4, though development of the Linux Kernel is of course ongoing and newer versions become available on a regular basis



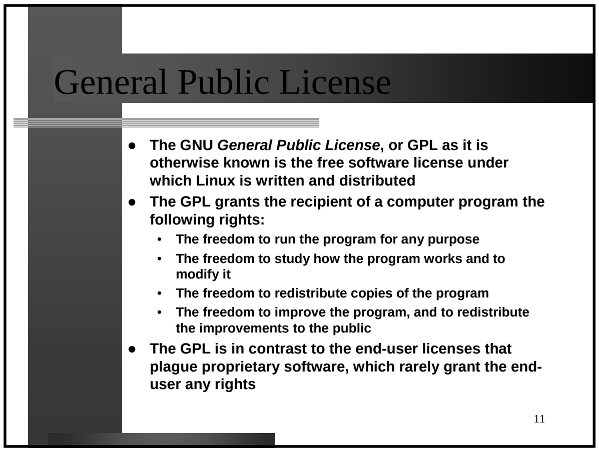
The Linux Kernel

- Like any Operating System, the Linux Kernel is responsible for managing resources (memory and I/O), contains device drivers, networking stack, file system, and performs other OS tasks
- Linux implements different privilege levels, where a module, which is a Kernel function runs in kernel space (supervisor mode), and user applications run in user space (user mode)
- Linux can mange both multiple *processes* and multiple *processors* (symmetric multiprocessing, or SMP systems). As such, all kernel code is ***reentrant***



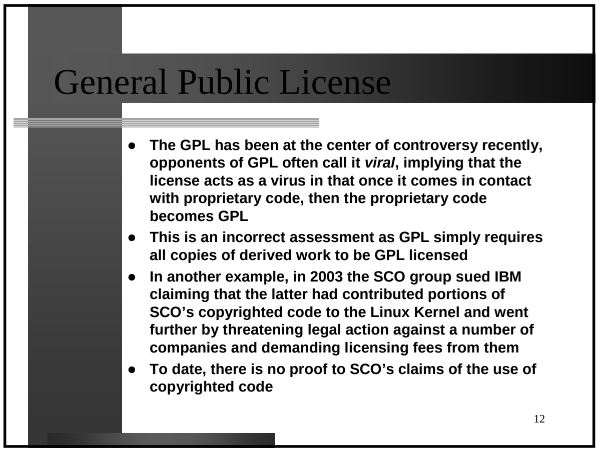
GNU

- GNU is an acronym for GNU's Not Unix, and is pronounced *guh-noo*
- The GNU project was started in 1983 with the goal of creating a UNIX flavored operating system which was freely distributable
- GNU is not Linux! GNU is used in conjunction with the Linux kernel to form a completely operational Operating System. This GNU/Linux combination (distribution) is often mistakenly called Linux
- Some software developed by the GNU project are: *Bash* (command shell), *Emacs* (text editor), *gzip* (data compression), and *GNOME* (graphical desktop environment)



General Public License

- The GNU *General Public License*, or GPL as it is otherwise known is the free software license under which Linux is written and distributed
- The GPL grants the recipient of a computer program the following rights:
  - The freedom to run the program for any purpose
  - The freedom to study how the program works and to modify it
  - The freedom to redistribute copies of the program
  - The freedom to improve the program, and to redistribute the improvements to the public
- The GPL is in contrast to the end-user licenses that plague proprietary software, which rarely grant the end-user any rights



General Public License

- The GPL has been at the center of controversy recently, opponents of GPL often call it *viral*, implying that the license acts as a virus in that once it comes in contact with proprietary code, then the proprietary code becomes GPL
- This is an incorrect assessment as GPL simply requires all copies of derived work to be GPL licensed
- In another example, in 2003 the SCO group sued IBM claiming that the latter had contributed portions of SCO's copyrighted code to the Linux Kernel and went further by threatening legal action against a number of companies and demanding licensing fees from them
- To date, there is no proof to SCO's claims of the use of copyrighted code

Other Licensing Models

- GPL is not the only licensing model available
- Some licenses such as BSD permit distribution of a modified BSD-based code as proprietary software
- The difference between GPL and BSD licenses is legal mechanism known as *copyleft*, invented by Richard Stallman (initiator of GNU project and founder of Free Software Foundation)
- Copyleft requires that derivative works of a GPL-licensed application also be covered by the GLP license

The Copyleft

- The right to redistribute GPL-based code is granted only if the licensee includes the source code in the redistribution (including all modifications!)
- The redistributed copies themselves are required to include and be licensed under GPL in a mechanism known as copyleft
- Copyleft actually derives its legal impact from the fact that the program is copyrighted!
- Under a copyright, a licensee does not have the right to modify or redistribute the code unless under the terms outlined in copyleft
- Therefore copyleft uses copyright law to accomplish an almost opposite effect – granting modification and redistribution rights

The GNU Toolchain

- Linux relies on the GNU development toolchain
- A *toolchain* is series of programming tools (assembler, compiler, linker, etc.) which are used to create another computer program
- The tools are used sequentially, or in a *chain*, in such a way that the output of one program becomes the input of another one, hence the term toolchain

The GNU Toolchain

- The *GNU toolchain* is an overall term given to the series of programming tools developed by the GNU project
- The projects include:
  - GNU *make* – Build and compilation automation
  - GNU Compiler Collection (*GCC*) – Compilers for several programming languages
  - GNU *Binutils* – Linker, assembler, and other tools
  - GNU Debugger (*GDB*) – Interactive debugger
- Other related projects are:
  - GNU C Library – A standard C library
  - CVS – Concurrent Version System

CVS

- The Concurrent Versions System implements a version control system to keep track of all the work and changes in a set of files
- This enables developers from across the globe to collaborate on a project and as such as become a popular component of the open-source development community

CVS

- The way CVS works is as follows:
  - Any number of clients (developers) can check-out a full copy of a given project
  - One or more developers can work on the same copy of the code and then check-in their modifications
  - The CVS server automatically attempts to merge the different changes
  - If it is unsuccessful, for example in the case where two developers are trying to modify the same line of code, it rejects the second developer from updating the code, and directs the two developers to merge the code manually

Developing a Linux System

- There are three basic setup mechanisms which developers use to develop code for Linux
  - The Permanent Link Setup is where the host and the target are permanently connected together via an Ethernet cable for example. In this case a root file system can be NFS-mounted which prevents the need for constantly copying programs back and forth
  - The Removable Storage Setup is a situation where the code is created on the host, copied onto a removable storage device such as Compact Flash and transferred to the target
  - The Stand-alone Setup is a situation where the toolchain is contained on the target, as could be the case for creating embedded Linux on PC-based platforms

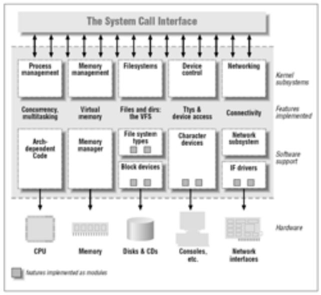
Starting up Linux

- From system power up to the time the system is up and running, there are three distinct steps the must be completed
  - *Bootloader* is the first piece of code which runs on the hardware and it is closely related to the type of platform on which it runs. There are many different types of bootloaders for Linux
  - *Kernel Startup Code* is the second stage of the boot process and it too differs greatly depending on the target platform
  - *Init* is the final process which further initializes the system

Linux Device Drivers

- Most Linux users are happily unaware of the complexities associated with the underlying hardware
- But every piece of the underlying hardware requires a device driver be written for it, and this is a job embedded system designers bravely undertake
- In the Linux Kernel there are many concurrent processes which tend to various system resources, such as memory, I/O, or the file system
- Though the Kernel can have any number of processes, it can basically be broken into the following groups:

Linux Device Drivers



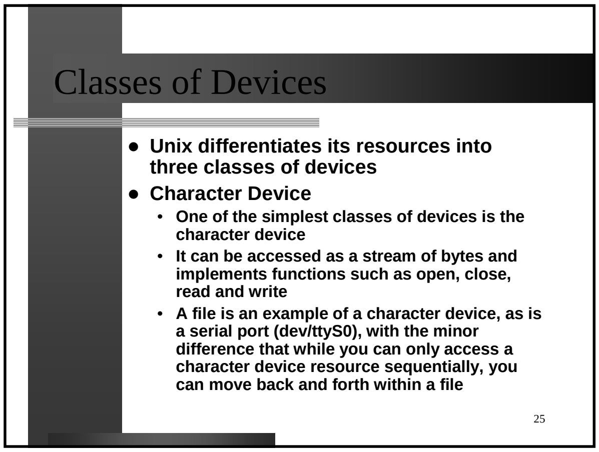
* Linux Device Drivers, Allesandro Rubini and Jonathan Corbet

Resource Management

- Process Management
  - It's the Kernels task to manage processes, to ensure that they can communicate with each other, and that they are scheduled, created and disposed of properly
- Memory Management
  - The Kernel is also responsible for handling memory resources, providing a virtual address space and memory management
- Filesystem
  - Filesystem is a major component of a UNIX (and Linux) operating system. Almost every resource in UNIX can be treated as a filesystem

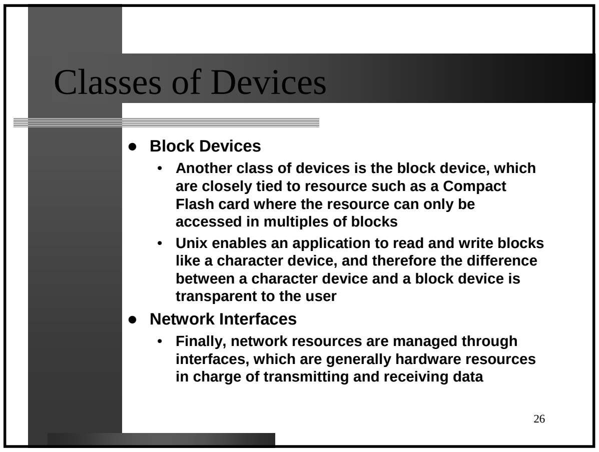
Resource Management

- Device Control
  - The Kernel implements a device driver for every hardware resource which is available on the system, ranging from hard drives to Timer modules
- Networking
  - Finally, the Kernel is responsible for providing a networking stack to the higher-level operating system functions



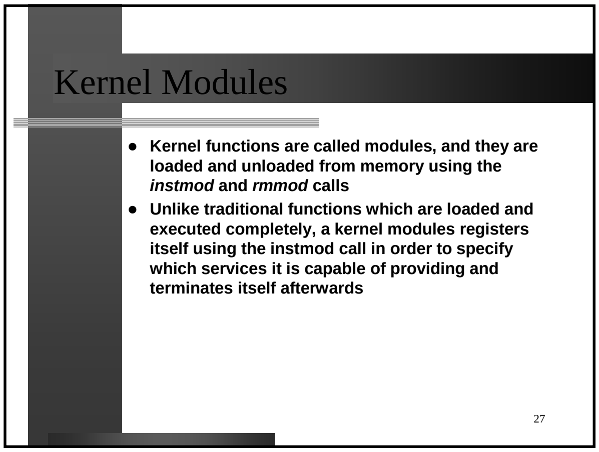
Classes of Devices

- Unix differentiates its resources into three classes of devices
- Character Device
  - One of the simplest classes of devices is the character device
  - It can be accessed as a stream of bytes and implements functions such as open, close, read and write
  - A file is an example of a character device, as is a serial port (dev/ttyS0), with the minor difference that while you can only access a character device resource sequentially, you can move back and forth within a file



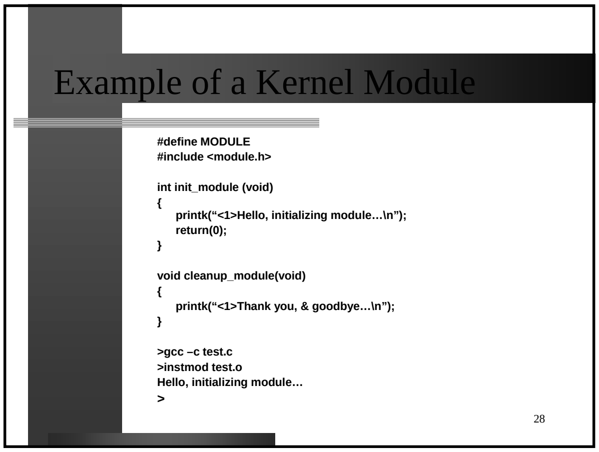
Classes of Devices

- Block Devices
  - Another class of devices is the block device, which are closely tied to resource such as a Compact Flash card where the resource can only be accessed in multiples of blocks
  - Unix enables an application to read and write blocks like a character device, and therefore the difference between a character device and a block device is transparent to the user
- Network Interfaces
  - Finally, network resources are managed through interfaces, which are generally hardware resources in charge of transmitting and receiving data



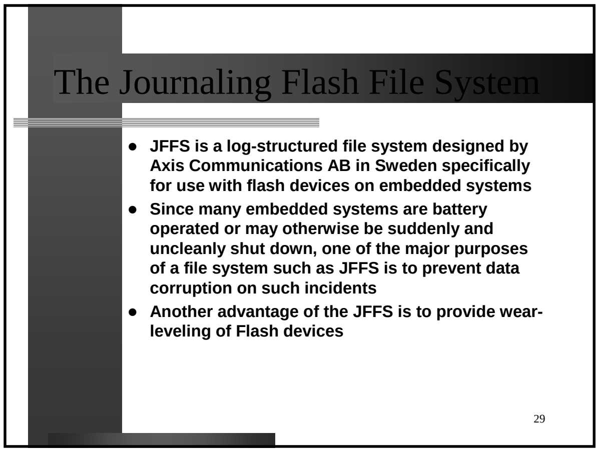
Kernel Modules

- Kernel functions are called modules, and they are loaded and unloaded from memory using the *instmod* and *rmmod* calls
- Unlike traditional functions which are loaded and executed completely, a kernel modules registers itself using the instmod call in order to specify which services it is capable of providing and terminates itself afterwards



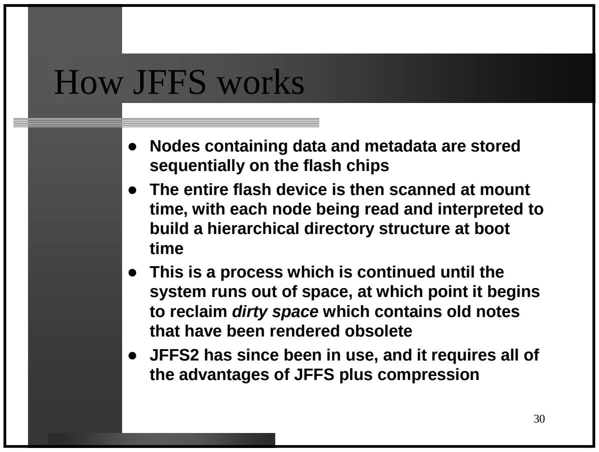
Example of a Kernel Module

```
#define MODULE
#include <module.h>

int init_module (void)
{
    printk(“<1>Hello, initializing module…\n”);
    return(0);
}

void cleanup_module(void)
{
    printk(“<1>Thank you, & goodbye…\n”);
}

>gcc –c test.c
>instmod test.o
Hello, initializing module…
>
```



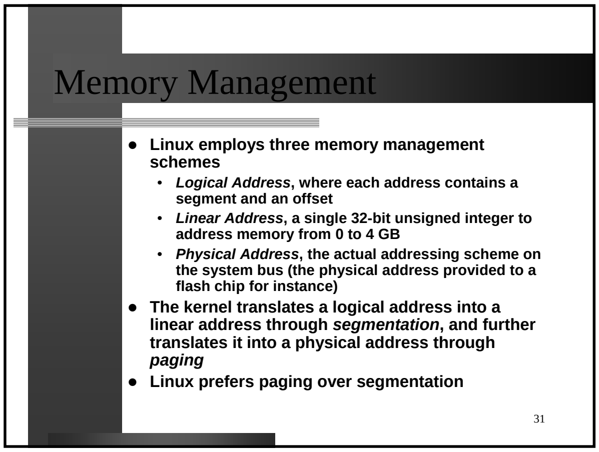
The Journaling Flash File System

- JFFS is a log-structured file system designed by Axis Communications AB in Sweden specifically for use with flash devices on embedded systems
- Since many embedded systems are battery operated or may otherwise be suddenly and uncleanly shut down, one of the major purposes of a file system such as JFFS is to prevent data corruption on such incidents
- Another advantage of the JFFS is to provide wear-leveling of Flash devices



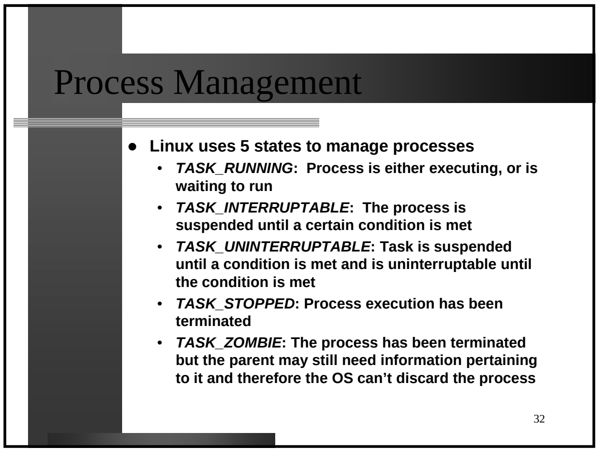
How JFFS works

- Nodes containing data and metadata are stored sequentially on the flash chips
- The entire flash device is then scanned at mount time, with each node being read and interpreted to build a hierarchical directory structure at boot time
- This is a process which is continued until the system runs out of space, at which point it begins to reclaim *dirty space* which contains old notes that have been rendered obsolete
- JFFS2 has since been in use, and it requires all of the advantages of JFFS plus compression



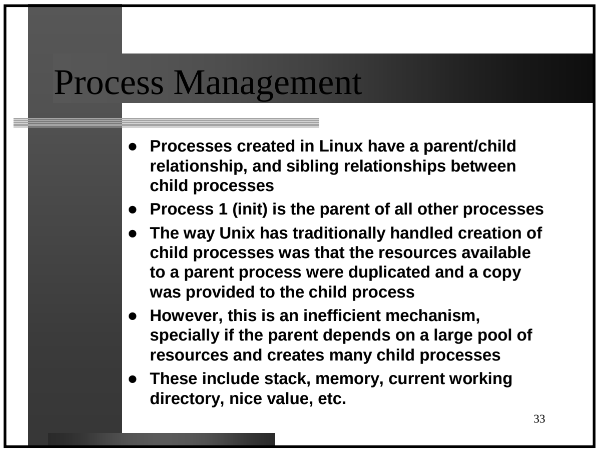
Memory Management

- Linux employs three memory management schemes
  - *Logical Address*, where each address contains a segment and an offset
  - *Linear Address*, a single 32-bit unsigned integer to address memory from 0 to 4 GB
  - *Physical Address*, the actual addressing scheme on the system bus (the physical address provided to a flash chip for instance)
- The kernel translates a logical address into a linear address through *segmentation*, and further translates it into a physical address through *paging*
- Linux prefers paging over segmentation



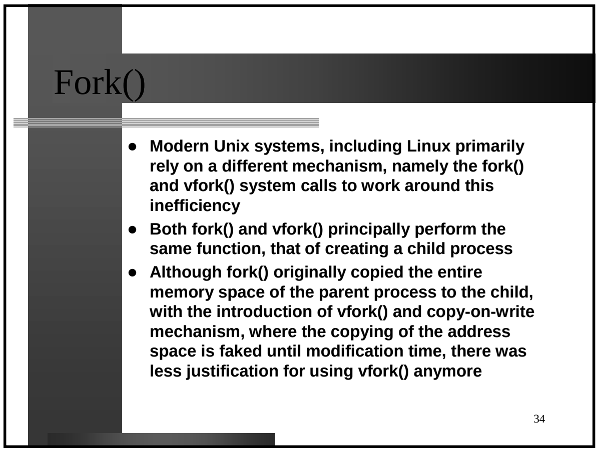
Process Management

- Linux uses 5 states to manage processes
  - *TASK_RUNNING*: Process is either executing, or is waiting to run
  - *TASK_INTERRUPTABLE*: The process is suspended until a certain condition is met
  - *TASK_UNINTERRUPTABLE*: Task is suspended until a condition is met and is uninterruptable until the condition is met
  - *TASK_STOPPED*: Process execution has been terminated
  - *TASK_ZOMBIE*: The process has been terminated but the parent may still need information pertaining to it and therefore the OS can't discard the process



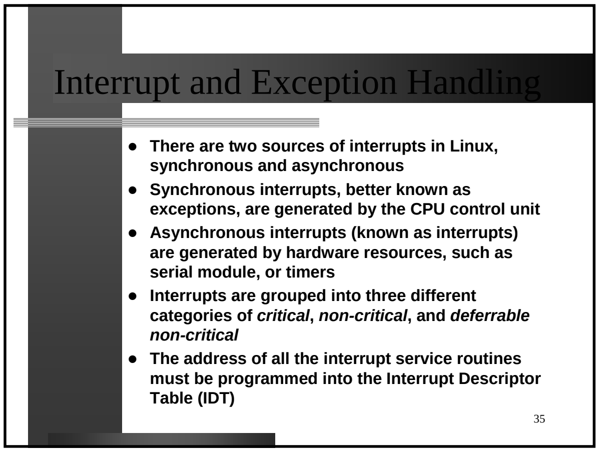
Process Management

- Processes created in Linux have a parent/child relationship, and sibling relationships between child processes
- Process 1 (init) is the parent of all other processes
- The way Unix has traditionally handled creation of child processes was that the resources available to a parent process were duplicated and a copy was provided to the child process
- However, this is an inefficient mechanism, specially if the parent depends on a large pool of resources and creates many child processes
- These include stack, memory, current working directory, nice value, etc.



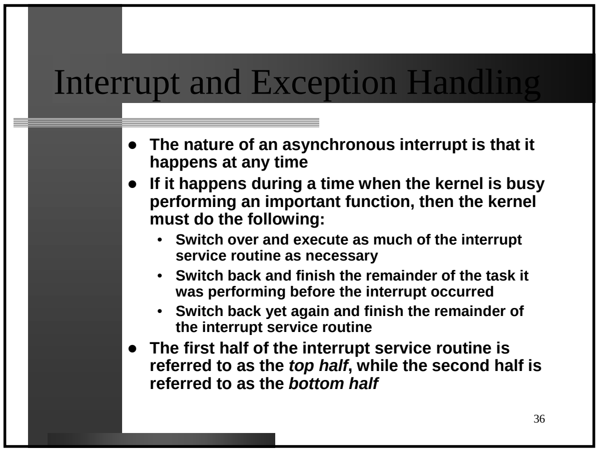
Fork()

- Modern Unix systems, including Linux primarily rely on a different mechanism, namely the fork() and vfork() system calls to work around this inefficiency
- Both fork() and vfork() principally perform the same function, that of creating a child process
- Although fork() originally copied the entire memory space of the parent process to the child, with the introduction of vfork() and copy-on-write mechanism, where the copying of the address space is faked until modification time, there was less justification for using vfork() anymore



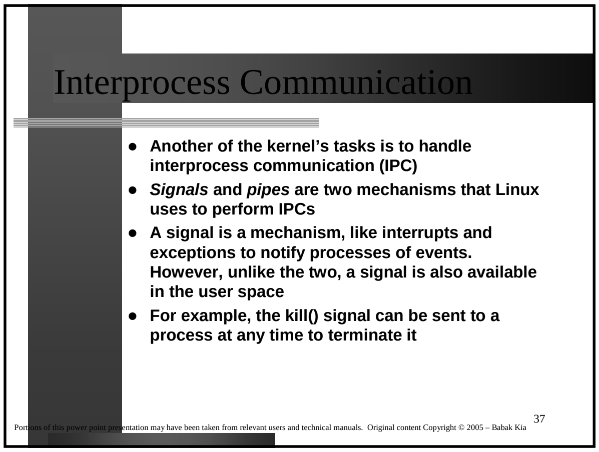
Interrupt and Exception Handling

- There are two sources of interrupts in Linux, synchronous and asynchronous
- Synchronous interrupts, better known as exceptions, are generated by the CPU control unit
- Asynchronous interrupts (known as interrupts) are generated by hardware resources, such as serial module, or timers
- Interrupts are grouped into three different categories of *critical*, *non-critical*, and *deferrable non-critical*
- The address of all the interrupt service routines must be programmed into the Interrupt Descriptor Table (IDT)



Interrupt and Exception Handling

- The nature of an asynchronous interrupt is that it happens at any time
- If it happens during a time when the kernel is busy performing an important function, then the kernel must do the following:
  - Switch over and execute as much of the interrupt service routine as necessary
  - Switch back and finish the remainder of the task it was performing before the interrupt occurred
  - Switch back yet again and finish the remainder of the interrupt service routine
- The first half of the interrupt service routine is referred to as the *top half*, while the second half is referred to as the *bottom half*



Interprocess Communication

- Another of the kernel’s tasks is to handle interprocess communication (IPC)
- *Signals* and *pipes* are two mechanisms that Linux uses to perform IPCs
- A signal is a mechanism, like interrupts and exceptions to notify processes of events. However, unlike the two, a signal is also available in the user space
- For example, the kill() signal can be sent to a process at any time to terminate it