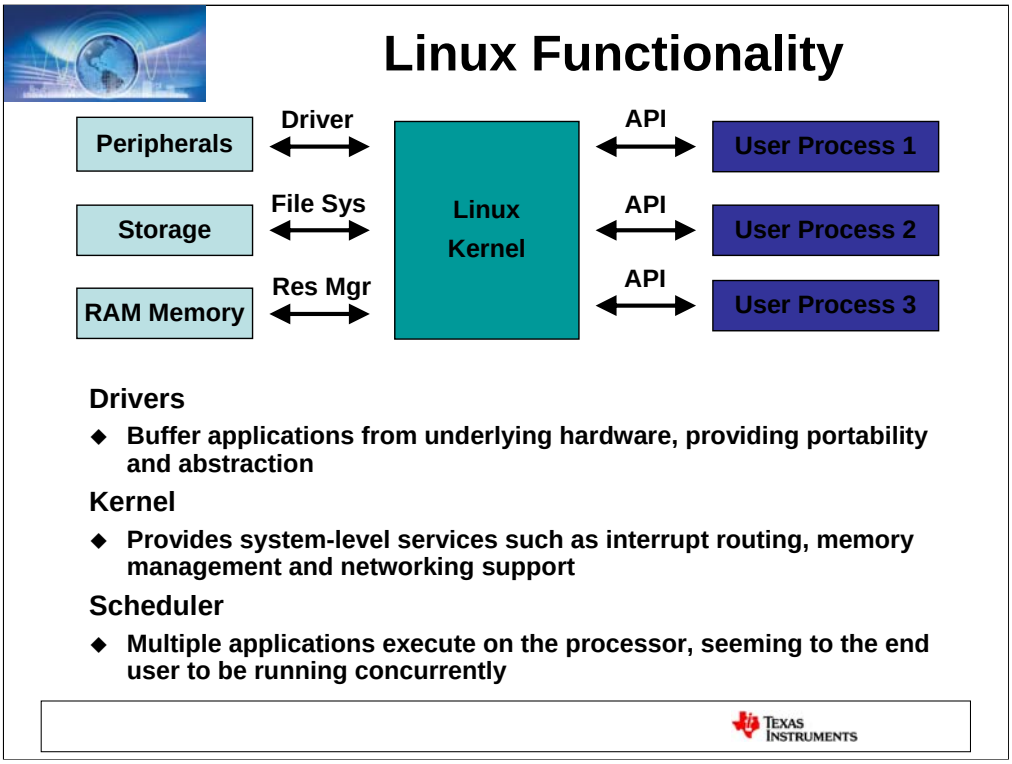
Introduction to Linux

Texas Instruments Tech Days

TTO
Technical Training Organization

Texas Instruments



MORE THAN JUST A SCHEDULER…

BIG VALUE: RESOURCE ABSTRACTION – HW SHARED VIA API SET

API DOESN'T CHANGE EVEN WHEN UNDERLYING HW CHANGES

EG: PORTING A PC BASED APP'N TO AN EMBEDDED SOLUTION – NEARLY SEAMLESS

VERY DIFFERENT THAN IN, SAY, A DSP SYSTEM…

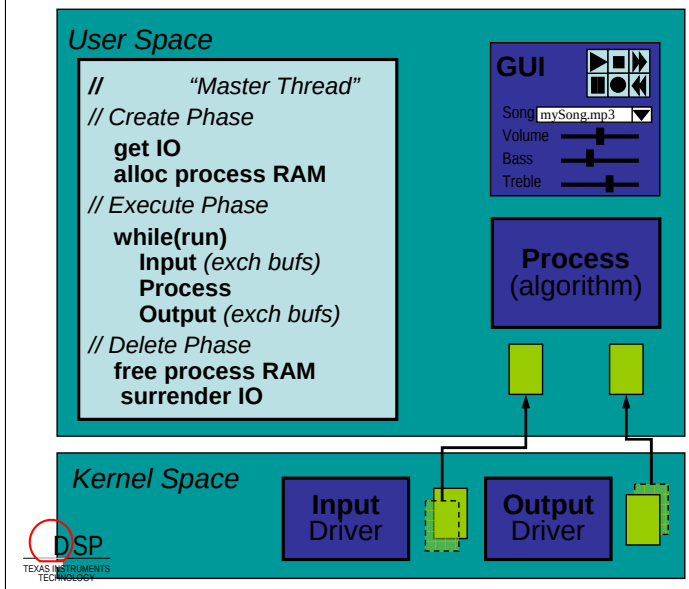
USER CODE UNCHANGED / DRIVER SWAPPING ONLY …

An operating system provides a common platform across many different machines on which user programs may run. User programs request resources such as memory and peripherals through a standard set of API (Application program interface, note User Programs are also called Application Programs in computerspeak) calls to the operating system.

By extracting the user interface, memory and peripherals away from the user program, this ensures that user programs can run on any machine supported by the O/S. Further, by managing these resources, an OS can provide a scalable environment in which many different programs may share common resources.

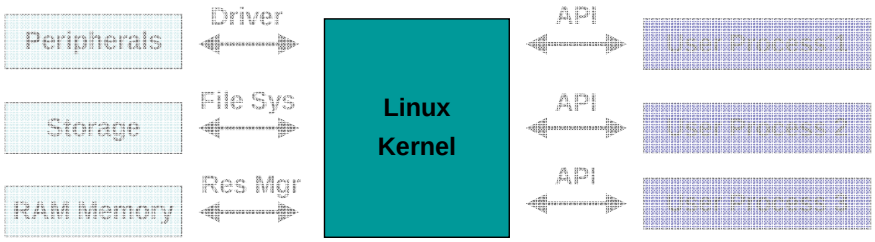
In addition to managing resources, GP O/S'es also manage user I/O, i.e. the user interface, which is why hitting ctrl-alt-delete will take you to a certain screen no matter what program you are in, and why all programs display within Windows on the Windows O/S.

Linux-based Solution



- **Application Host Software Pre-Ported to Linux**
  - Device drivers
  - DSP link
  - Sample code
- **Developers can leverage the abundance of available open source and third party software**
  - Networking software
  - Web browser
  - Signaling software, e.g., SIP
  - Electronic programming guide

Agenda



- **Kernel and Booting**
- **Drivers**
- **Scheduling**

Linux in Three Parts

❶ Bootloader

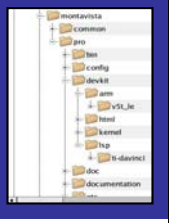
- Provides rudimentary h/w init
- Calls Linux kernel and passes boot arguments

❷ Kernel

- Initializes the system (and device)
- Manages system resources
- Provides services for user programs

❸ Filesystem

- Single filesystem (/ root)
- Stores all system files
- After init, kernel looks to filesystem for “what’s next”
- bootarg tells linux where to find root filesystem



Texas Instruments

Linux Boot Process

Power On

Boot Loader

U-Boot
- ARM assembly code
- Passes args to Linux (bootargs)

Linux Kernel

Linux Kernel — Initialize hardware

Initrd (optional) — Initial RAM Disk *(we don't use this)*

File System

Init Process — /sbin/init – 1st process exe by kernel

Login Prompt
- Login console
- Usually one of first prog's to run

Texas Instruments

Bootloader: Das U-Boot

- U-Boot resides in NOR flash memory, copies itself to DDR, and executes itself from DDR when the DaVinci EVM is powered on.
- In general, U-Boot performs the following functions:
  1. Initializes the processor
  2. Provides boot parameters to the Linux kernel
  3. Starts the Linux kernel

DSP
TEXAS INSTRUMENTS TECHNOLOGY

Texas Instruments

Configuring U-Boot

Common Uboot Commands:

- printenv - prints one or more uboot variables
- setenv - sets a uboot variable
- saveenv - save uboot variable(s)
- run - evaluate a uboot variable expression
- ping - (debug) use to see if Uboot can access NFS server

Common Uboot Variables:

- You can create whatever variables you want, though some are defined either by Linux or common style
  - bootcmd - where Linux kernel should boot from
  - bootargs - string passed when booting Linux kernel
    e.g. tells Linux where to find the root filesystem
  - serverip - IP address of root file system for NFS boot
  - nfspath - Location on serverip for root filesystem

Texas Instruments

Boot Variations

Mode	IP	Linux Kernel	Root Filesystem
1.	dhcp	Flash	HDD
2.	dhcp	Flash	NFS
3.	dhcp	TFTP	HDD
4.	dhcp	TFTP	NFS

U-Boot's bootargs variable specifies where to find the root filesystem

HDD

```
setenv bootargs console=ttyS0,115200n8
noinitrd rw ip=dhcp root=/dev/hda1, nolock
mem=120M
```

NFS

```
setenv bootargs console=ttyS0,115200n8
noinitrd rw ip=dhcp root=/dev/nfs
nfsroot=$(serverip):$(nfspath),nolock
mem=120M
```

Texas Instruments

Configuring U-Boot
Kernel from Flash, Filesystem from HDD

```
[rs232]# baudrate 115200
[rs232]# setenv stdin serial
[rs232]# setenv stdout serial
[rs232]# setenv stderr serial
[rs232]# setenv bootdelay 3
[rs232]# setenv bootfile uImage
[rs232]# setenv serverip 192.168.2.101
[rs232]# setenv nfspath /home/user/workdir/filesys
[rs232]# setenv bootcmd bootm 0x2050000
[rs232]# setenv bootargs console=ttyS0,115200n8
     noinitrd rw ip=dhcp root=/dev/hda1, nolock
     mem=120M
[rs232]# saveenv
```

Configuring U-Boot
Kernel from Flash, Filesystem from NFS

```
[rs232]# baudrate 115200
[rs232]# setenv stdin serial
[rs232]# setenv stdout serial
[rs232]# setenv stderr serial
[rs232]# setenv bootdelay 3
[rs232]# setenv bootfile uImage
[rs232]# setenv serverip 192.168.2.101
[rs232]# setenv nfspath /home/user/workdir/filesys
[rs232]# setenv bootcmd bootm 0x2050000
[rs232]# setenv bootargs console=ttyS0,115200n8
     noinitrd rw ip=dhcp root=/dev/nfs
     nfsroot=$(serverip):$(nfspath),nolock
     mem=120M
[rs232]# saveenv
```

Texas Instruments

Configuring U-Boot
Kernel via TFTP, Filesystem from NFS

```
[rs232]# baudrate 115200
[rs232]# setenv stdin serial
[rs232]# setenv stdout serial
[rs232]# setenv stderr serial
[rs232]# setenv bootdelay 3
[rs232]# setenv bootfile uImage
[rs232]# setenv serverip 192.168.2.101
[rs232]# setenv nfspath /home/user/workdir/filesys
[rs232]# setenv bootcmd 'dhcp;bootm'
[rs232]# setenv bootargs console=ttyS0,115200n8
     noinitrd rw ip=dhcp root=/dev/nfs
     nfsroot=$(serverip):$(nfspath),nolock
     mem=120M
[rs232]# saveenv
```

Texas Instruments

Booting with Static IP Addresses

Mode	IP	Linux Kernel	Root Filesystem
1.	dhcp	Flash	HDD
2.	dhcp	Flash	NFS
3.	dhcp	TFTP	HDD
4.	dhcp	TFTP	NFS
5.	static	Flash	HDD
6.	static	Flash	NFS
7.	static	TFTP	HDD
8.	static	TFTP	NFS

Texas Instruments

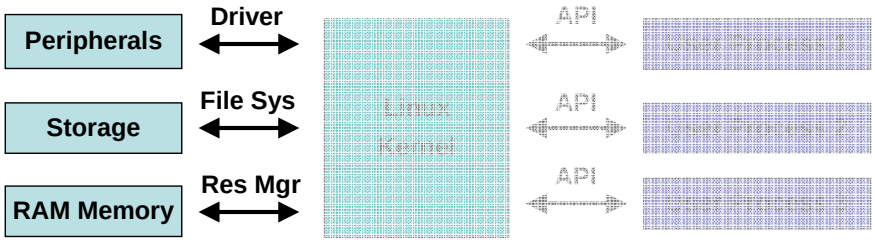
U-Booting : Static vs Dynamic IP

- Everywhere we previously had dhcp must now reference the static ip addresses
- This example creates a variable called myip and used it in place of the previous dhcp entries in *bootargs*

```
[rs232]# setenv serverip 192.168.13.120
[rs232]# setenv ipaddr 192.168.13.121
[rs232]# setenv gateway 192.168.13.97
[rs232]# setenv netmask 255.255.255.224
[rs232]# setenv dns1 156.117.126.7
[rs232]# setenv dns2 157.170.1.5
[rs232]# setenv myip $(ipaddr):$(gateway):$(netmask):$(dns1):$(dns2)::off
[rs232]# setenv nfspath /home/user/workdir/filesys
[rs232]# setenv bootcmd bootm 0x2050000
[rs232]# setenv bootargs console=ttyS0,115200n8 noinitrd rw
    ip=$(myip) root=/dev/nfs nfsroot=$(serverip):$(nfspath)
    ,nolock mem=120M $(videocfg)
[rs232]# saveenv
```

Texas Instruments

Agenda



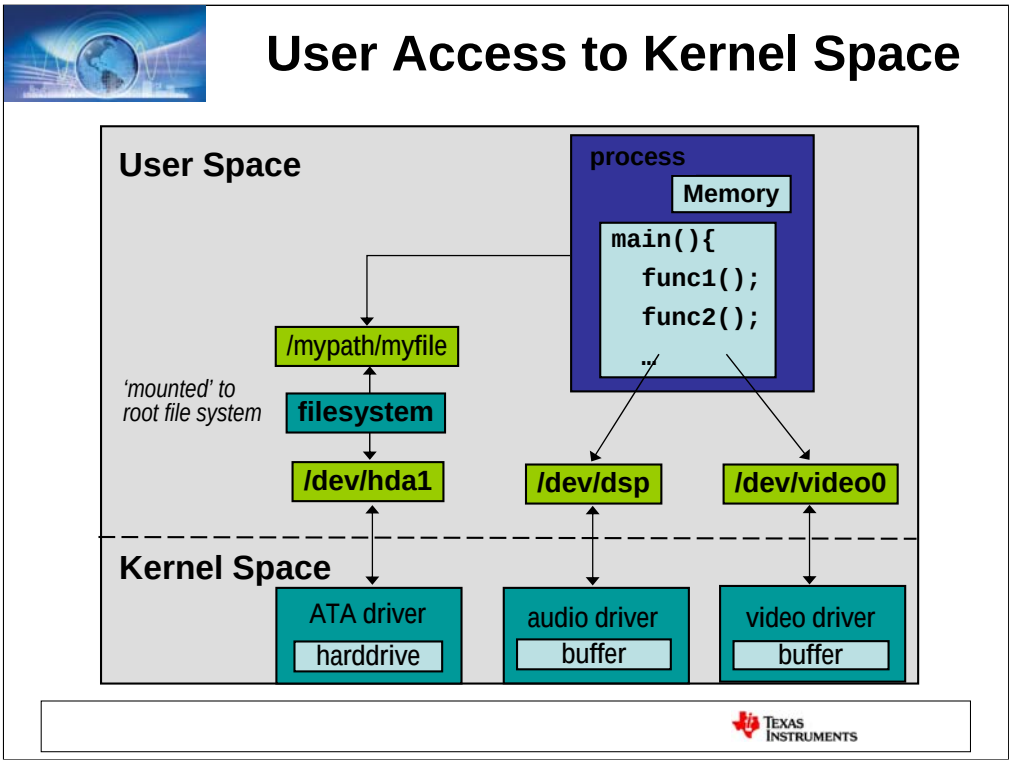
- Kernel and Booting
- Drivers
- Scheduling

Example Linux Drivers

- **GPIO**
- **Serial - UART, I2C, SPI**
- **Storage - ATA, NAND, MMC**
- **Audio - ALSA Audio driver**
- **Video - V4L2 for Video Capture/Display**
- **Video - FBDev for On-Screen Display**
- **Network - 10/100 Ethernet (EMAC/CPMAC)**
- **USB - Mass storage - Host and Gadget drivers**
- **Boot - Das UBoot** (open source Linux boot-loader)

DSP
TEXAS INSTRUMENTS TECHNOLOGY

Texas Instruments



PROTECTION OF THE SYSTEM – MMU – EACH APP HAS IT'S OWN MEMORY AND CANNOT TOUCH BEYOND – BY ACCIDENT OR INTENTION

HOW DOES A PROC ACCESS PERIPHS THEN?

VIA 'DEVICE NODES'

READS FROM PORTS ARE AS THOUGH THEY WERE FILES ON A DRIVE (ABSTRACTION)

TWO TYPES OF DRIVERS:

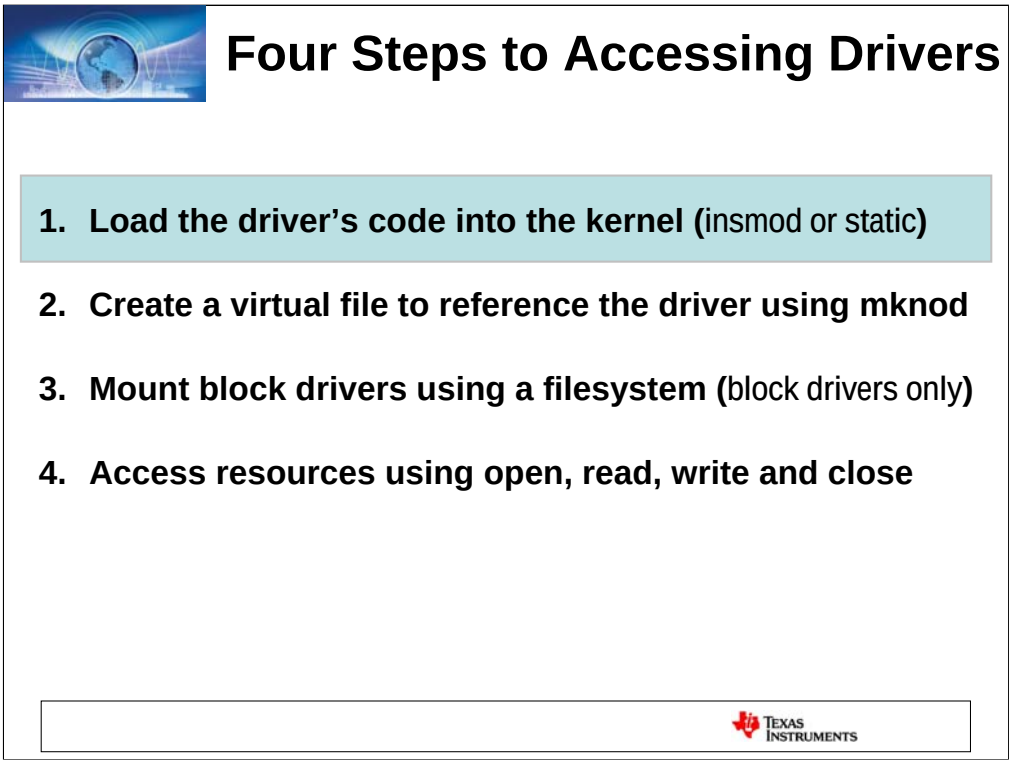
CHARACTER (STREAM) DRIVERS

BLOCK DRIVERS

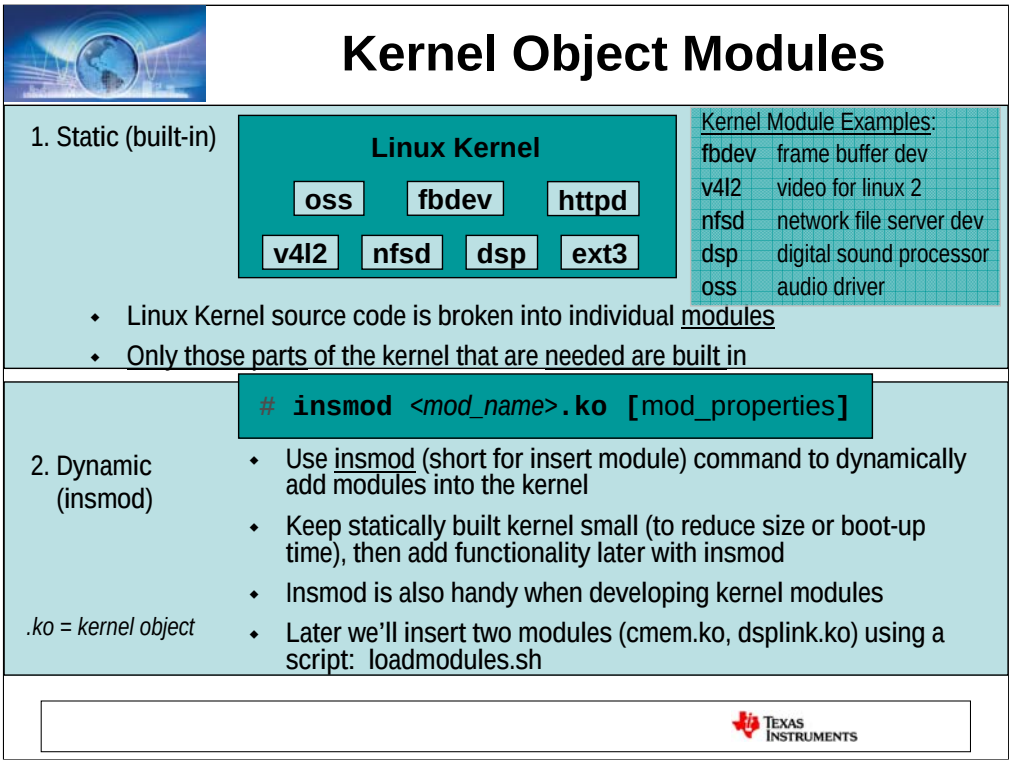
BLOCK DRIVERS ALLOW RANDOM ACCESS ANYWHERE IN PERIPH (EG: HDD)

BLOCK DRIVERS ARE MOUNTED INTO THE ROOT FILE SYSTEM

ACCESSED NOT BY THEIR NODE ID BUT INSTEAD BY THEIR FILE NAMES (ABOVE)



RANDOM ACCESS REQUIRES FILE SYSTEM TO FIND GIVEN COMPONENT ON PERIPH.



LIKE OTHER OSs

PEOPLE LIKE A GIVEN NUMBER OF SVCS, BUT NOT ANY GREATER BUILD SIZE THAN NECESSARY

THUS, LINUX IS A MODULAR OS, BUILDING ONLY THE SVCS AND COMPONENTS YOU REQUIRE

DAEMONS – ALWAYS RUNNING IN BKGND…

WAITING FOR NEW ACTIVITY – EG: INTERNET TCP/IP

FILE SYSTEM – WHICH ONES DO *YOU* WANT?

EXT3 – HDD SUPPORT… LATER

INSERTMODULE MODULE NAME . KERNEL OBJECT … PROPERTIES

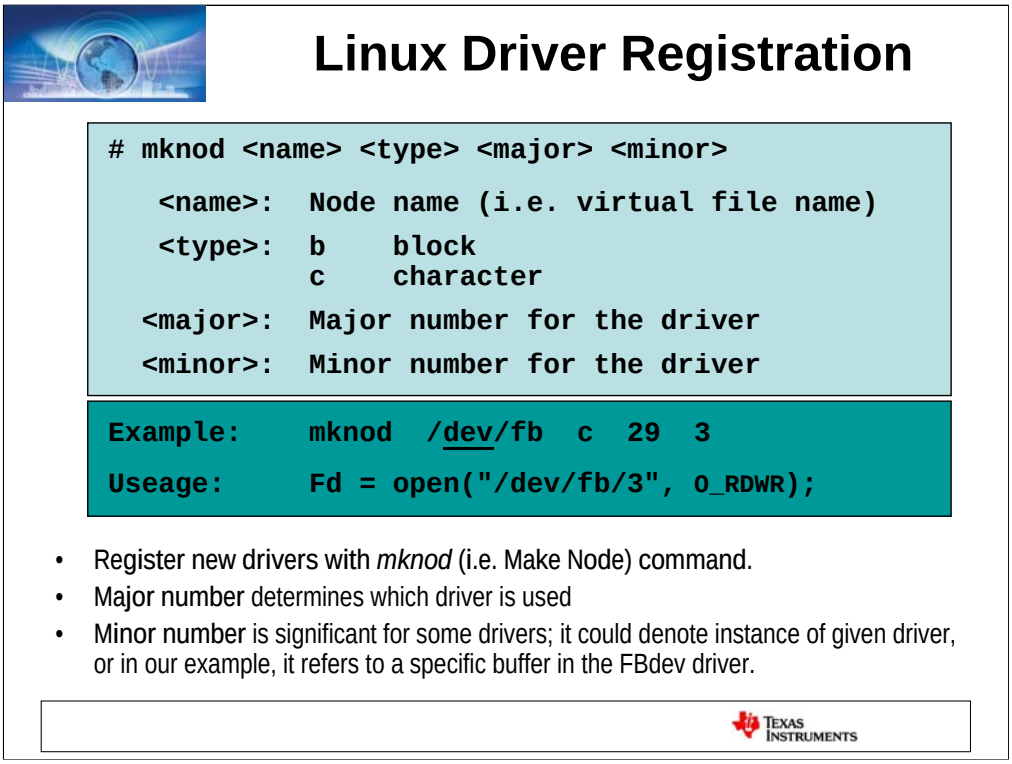
DRIVERS CAN BE STATICALLY BUILT INTO A GIVEN KERNEL OR DYNAMICALLY INVOKED, AS SEEN HERE…

Examining The Steps in More Detail…

1. **Load the driver's code into the kernel (insmod or static)**
2. **Create a virtual file to reference the driver using mknod**
3. **Mount block drivers using a filesystem (**block drivers only**)**
4. **Access resources using open, read, write and close**

Texas Instruments

MAKE NODE :



MAKE NODE :

ALL DEVICE NODES, BY LINUX ‘STYLE’ GO IN THE /DEV DIRECTORY OFF THE ROOT.

MYDEVICE – YOUR DRIVER NAME

TYPE – BLOCK CHAR, ETC

MAJOR/MINOR NUMBER – ENUMERATION OF DRIVERS BY TYPE AND EXAMPLE

A NUMBER OF LONG USED DRIVERS HAVE FIXED ‘WELL KNOWN’ MAJOR NUMBERS

OTHERS THE NUMBERS ARE CHOSEN AT AUTHORS DISCRETION AND REFERENCED BY USER.

MINOR NUMBER – USUALLY FOR ‘INSTANCE NUMBERS’ : EG: FOR 4 SERIAL DRIVERS, BUILD ONLY 1 SET OF CODE, BUT RECOGNIZE (AS DATA OBJECTS) A NUMBER OF CHANNELS OF THAT CODE.

Most device files will already be created and will be there ready to use after you install your Linux system. If by some chance you need to create one which is not provided then you should first try to use the **MAKEDEV** script. This script is usually located in /dev/MAKEDEV but might also have a copy (or a symbolic link) in /sbin/MAKEDEV. If it turns out not to be in your path then you will need to specify the path to it explicitly.

In general the command is used as: # **/dev/MAKEDEV -v ttyS0** create ttyS0 c 4 64 root:dialout 0660 This will create the device file /dev/ttyS0 with major node 4 and minor node 64 as a character device with access permissions 0660 with owner root and group dialout.

ttyS0 is a serial port. The major and minor node numbers are numbers understood by the kernel. The kernel refers to hardware devices as numbers, this would be very difficult for us to remember, so we use filenames. Access permissions of 0660 means read and write permission for the owner (root in this case) and read and write permission for members of the group (dialout in this case) with no access for anyone else.

Block and Character Drivers

Block Drivers:		
	/dev/hda	**ATA → harddrive, CF**
	/dev/ram	**external RAM**
Character Drivers:		
	/dev/dsp	**sound driver**
	/dev/video0	**v4l2 video driver**
	/dev/fb0	**frame buffer video driver**

- **Block drivers allow out of order access**
- **Block devices can be mounted into the filesystem**
- **Character drivers are read as streams in a FIFO order**
- **Networking drivers are special drivers**

Texas Instruments

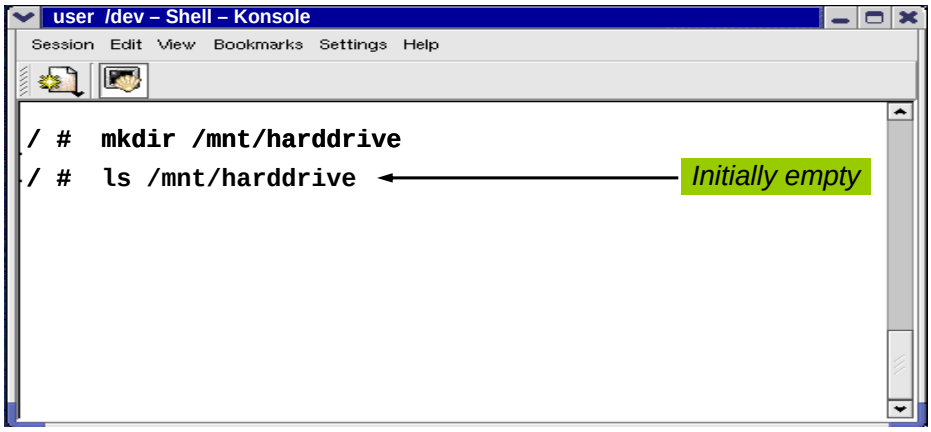
Examining The Steps in More Detail…

1. **Load the driver's code into the kernel (insmod or static)**
2. **Create a virtual file to reference the driver using mknod**
3. **Mount block drivers using a filesystem (**block drivers only**)**
4. **Access resources using open, read, write and close**

Texas Instruments

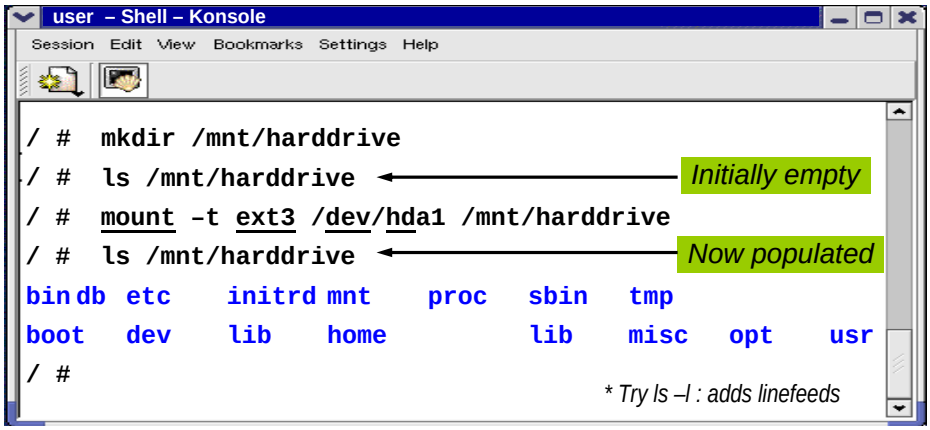
MAKE NODE :

Mounting Block Devices



- Mounting a block driver into the filesystem gives access to the files on the device as a new directory
- Easy manipulation of flash, hard drive, compact flash and other storage media
- Use mkfs.ext2, mkfs.jffs2, etc. to format a device with a given filesystem

Mounting Block Devices



- Mounting a block driver into the filesystem gives access to the files on the device
- You must mount to a mount point (i.e. empty directory) in the root filesystem
- Use mkfs.ext2, mkfs.jffs2, etc. to format a device with a given filesystem

Example Linux File Systems

Harddrive File systems:

ext2	Common general-purpose filesystem
ext3	Journaled extension to ext2
vfat	Windows “File Allocation Table” filesystem

Memory File systems:

jffs2	Journaling flash filesystem (NOR flash)
yaffs	yet another flash filesystem (NAND flash)
ramfs	Filesystem for RAM
cramfs	Compressed RAM filesystem

Network File systems:

nfs	Share a remote linux filesystem
smbfs	Share a remote Windows® filesystem

Texas Instruments

Examining The Steps in More Detail…

1. **Load the driver's code into the kernel (insmod or static)**
2. **Create a virtual file to reference the driver using mknod**
3. **Mount block drivers using a filesystem (**block drivers only**)**
4. **Access resources using open, read, write and close**

Texas Instruments

MAKE NODE :

Basic File I/O & Character Driver API

Basic Linux file I/O usage in user programs is via these API:

```
myFileFd = fopen("/mnt/harddrive/myfile", "rw");
fread ( aMyBuf, sizeof(int), len, myFileFd );
fwrite( aMyBuf, sizeof(int), len, myFileFd );
fclose( myFileFd );
```

Additionally, you can use fprintf() and fscanf() for more feature-rich file read/writes.

Simple drivers use the same format as files…

```
soundFd = open("/dev/dsp", O_RDWR);
read ( soundFd, aMyBuf, len );
write( soundFd, aMyBuf, len );
close( soundFd );
```

Additionally, drivers use I/O control (ioctl) commands to set driver characteristics

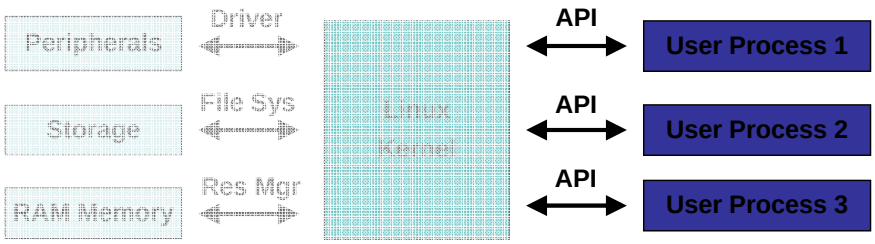
```
ioctl( soundFd, SNDCTL_DSP_SETFMT, &format );
```

- *Some Linux drivers (such as V4L2 and FBDEV video drivers) typically use* **`mmap`** *and ioctl commands instead of* `read` *and* `write` *that pass data by reference instead of by copy.*

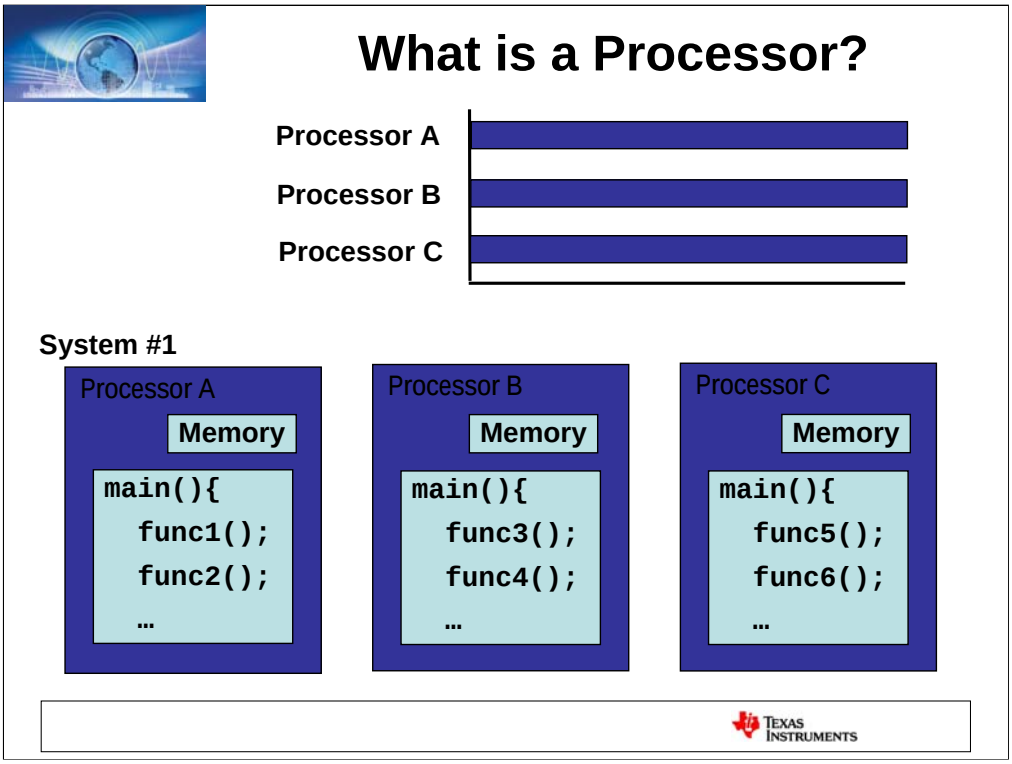
DSP TEXAS INSTRUMENTS TECHNOLOGY

Texas Instruments

Agenda



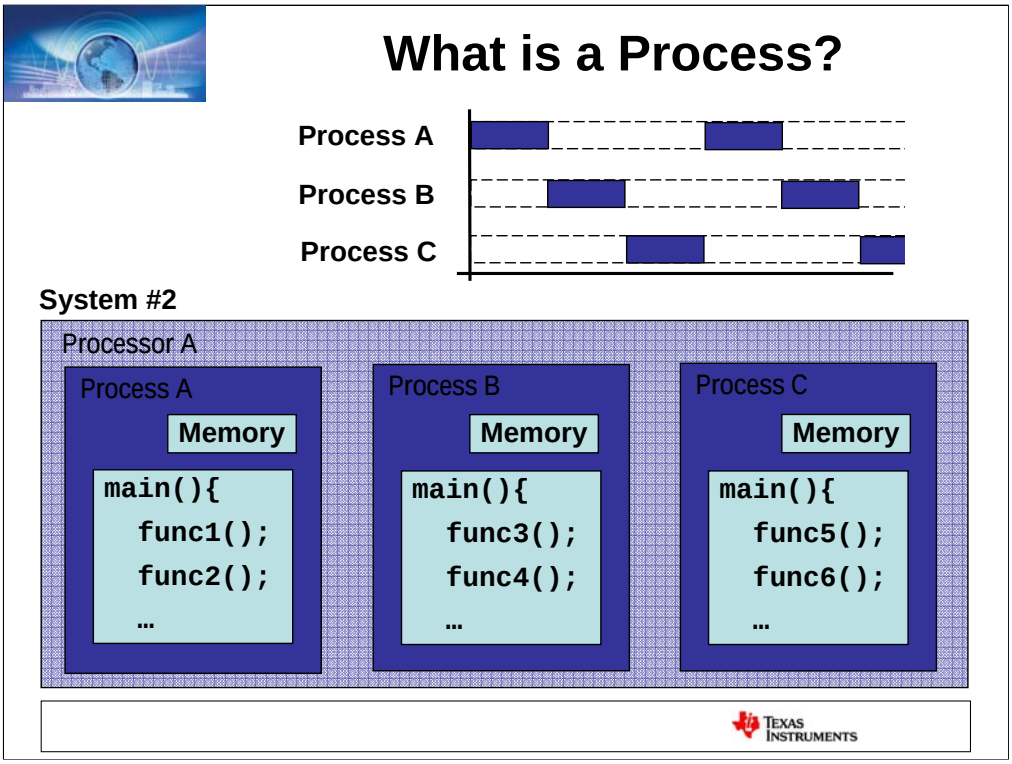
- **Kernel and Booting**
- **Drivers**
- **Scheduling**



When we executed our program in the previous section, we were really creating a Linux process and running our executable program within it. What do we mean when we say that we are running a process? What really is a process?

Let's back up for a moment and image a system which has three independent processors. Each processor, if it had no operating system, can only run a single program at a time, the entry point of which is the main() function. For instance, processor A in this example might run the program that we built in the previous section.

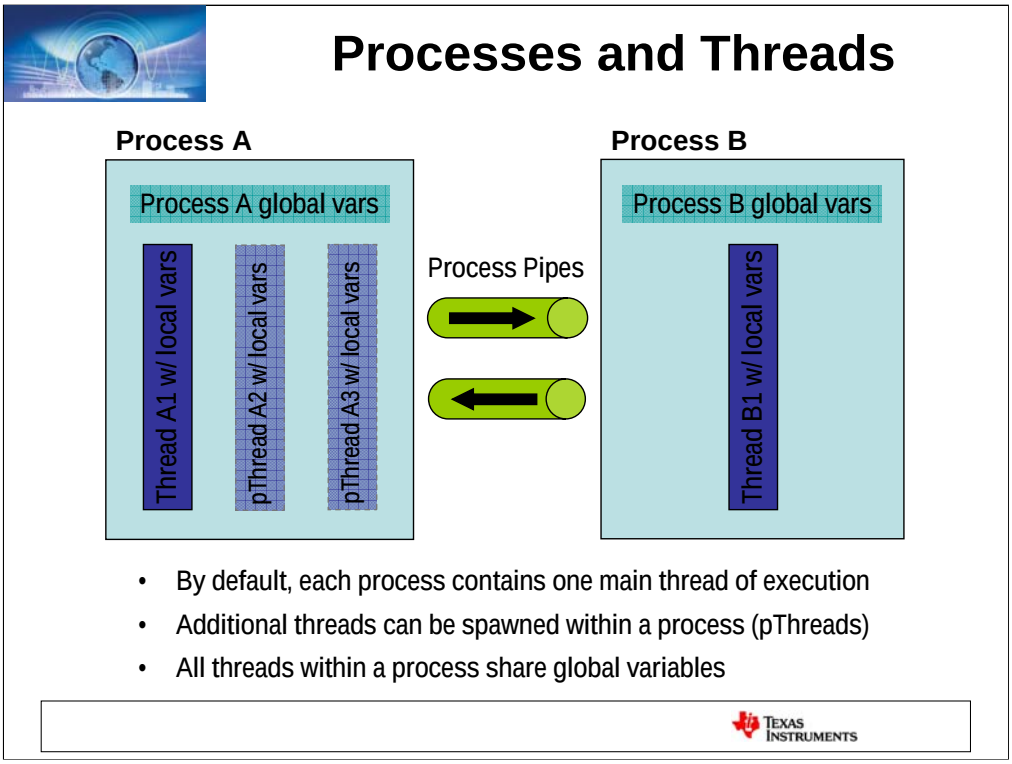
Each processor has a memory space, i.e. the internal and external memory that is physically connected to the processor. As shown in the bar diagram in the top left corner, the programs on these three processors run concurrently and independently, and the three memory spaces for the three processors are distinct, so that unless memory is shared physically through connections on the hardware board, one processor cannot access the memory used by another processor. This can be a nice feature because it provides insulation between the three processors. If the program on one processor has a rogue pointer which starts overwriting memory incorrectly, it may crash its own program, but since it cannot access the memory of the other two processors, it will at least not crash the programs which are running on them.



On this next slide, we show a similar system; however, in this system we have only one processor. If this processor is running an operating system such as Linux which supports multiple processes, we can take the three programs which in our previous example were running on three different processors and run them as three concurrent processes on the single processor.

The three processes have many characteristics corresponding to the three individual processors in the previous system. Firstly, each process encapsulates an executable program, each having its own main() function as an entry point. Furthermore, each process has its own memory space, which is separate from the memory spaces of other processes running on the system. (This feat is accomplished through hardware called the Memory Management Unit, and will not be discussed here.) So, as in our previous example, if the executable program running in process B develops a rogue pointer that begins overwriting random memory locations, it may bring down Process B, but cannot effect Processes A or C, which are isolated.

Notice that the difference in this system versus the previous systems is that the three processes, since they are now running on a single processor, cannot run concurrently. They now must be scheduled on the processor so that only one is running at a time, and the processor must be shared between the three. This is referred to as sheduling.



This slide shows an overall view of how processes and threads fit into the Linux operating system. As previously discussed, Linux may run multiple processes concurrently (at least appear to run concurrently, we already know they are actually time sliced). Each of these processes has its own memory space (note Process 1 global vars and Process 2 global vars, separate, each within their own process), but can communicate with each other through process pipes.

This much should be a review of the previous section. We have added, however, a new element of threads within each process. So process 1 has three threads running within it in this example system. Each of these threads can run concurrently (again, they are really time sliced, but appear to run concurrently), and each has its own entry point, so in those two ways they are similar to Processes.

The predominant difference between Processes and Threads is that the multiple threads within a process share the same Memory space (i.e. global variables, file descriptors, etc.), whereas the multiple Processes within the Linux O/S do not share the same Memory Space.

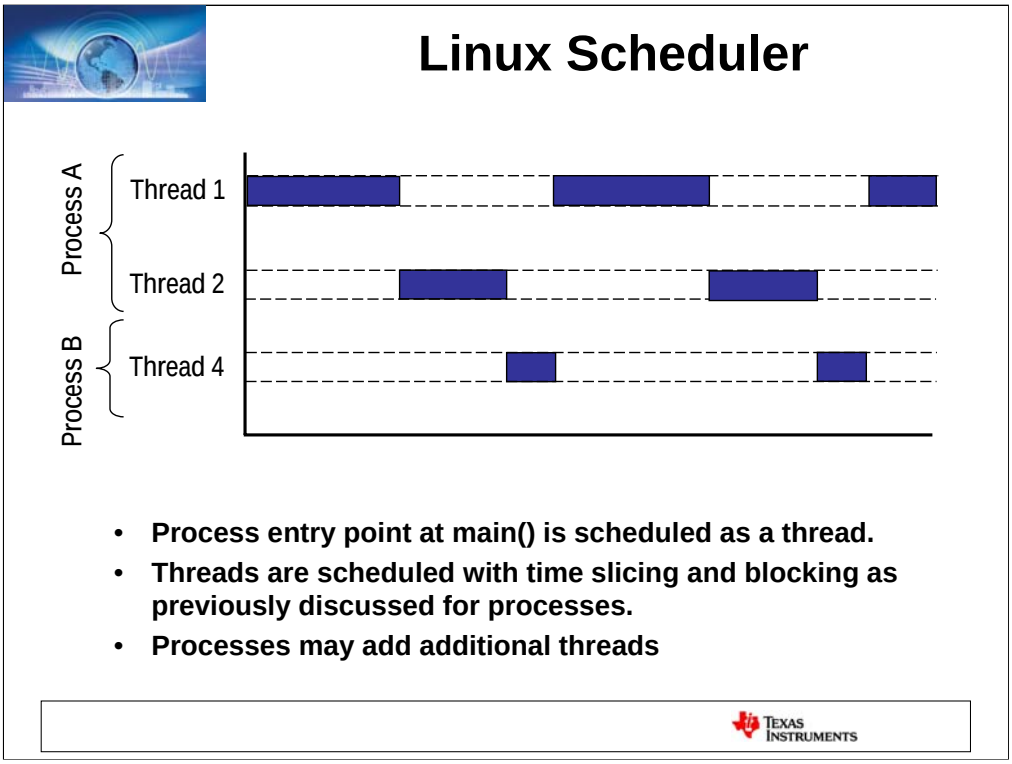
As a result, the overhead of creating new Threads within a process and the overhead of switching between threads in the same process is less than the overhead of creating new Processes and the overhead of switching between Processes. Furthermore, since Threads within the same process share the same Address space, they can pass large buffers with a simple pointer pass, whereas for processes to pass buffers of data over a process pipe requires copying of the entire buffer, a much less efficient process.

What threads give up in exchange for this reduced overhead, however, is the protection that Processes afford. Because threads exist in the same memory space, a rogue pointer in one thread will likely bring down all of the threads within the given process (though the threads in other processes will be insulated and will not be brought down.)

Threads vs Processes

	Processes	Threads
Memory protection	✓	✘
Ease of use	✓	✘
Start-up cycles	✘	✓
Context switch	✘	✓
Shared globals	**no**	**yes**
Environment	**program**	**function**

This slide is a visual of the speaker's notes on the previous slide, summarizing some of the pluses and minus of threads versus processes.



This slide is similar to the Linux Scheduler slide shown for Processes which did not show threads. The reason for this is actually due to bullet one above. Every process has a minimum of one thread, corresponding to the entry point at the main() function. You may think of this as an implicit thread since it does not have to be created using pthread_create, which is how additional threads are added into a process.

Because many Linux programmers do not use threads, a lot of literature shows the convenient simplification of individual processes being scheduled by Linux. In actuality, even on a system which consists of processes which only have one thread (the implicit main thread), it is really this implicit thread which is being scheduled, not the process.

For a system with some processes that have one thread and some processes with multiple threads, each thread is thrown into the mix and individually scheduled, so processes that “have no threads”, (i.e. which just have the one implicit main() thread) are scheduled as threads in the system. Processes with multiple threads simply have each of their individual threads scheduled individually on the system.

Scheduling Methodologies

Time-Slicing

- Scheduler shares processor run time between all threads with greater time for higher priority
- ✓ No threads completely starve
- ✓ Corrects for non-"good citizen" threads
- × Can't guarantee processor cycles, even to highest priority threads.
- × More context switching overhead

Realtime

- Higher priority threads must block for lower priority threads to run
- × Requires "good citizen" threads
- × Low priority threads may starve
- ✓ Lower priority threads never break high priority threads
- ✓ Lower context-switch overhead

Texas Instruments

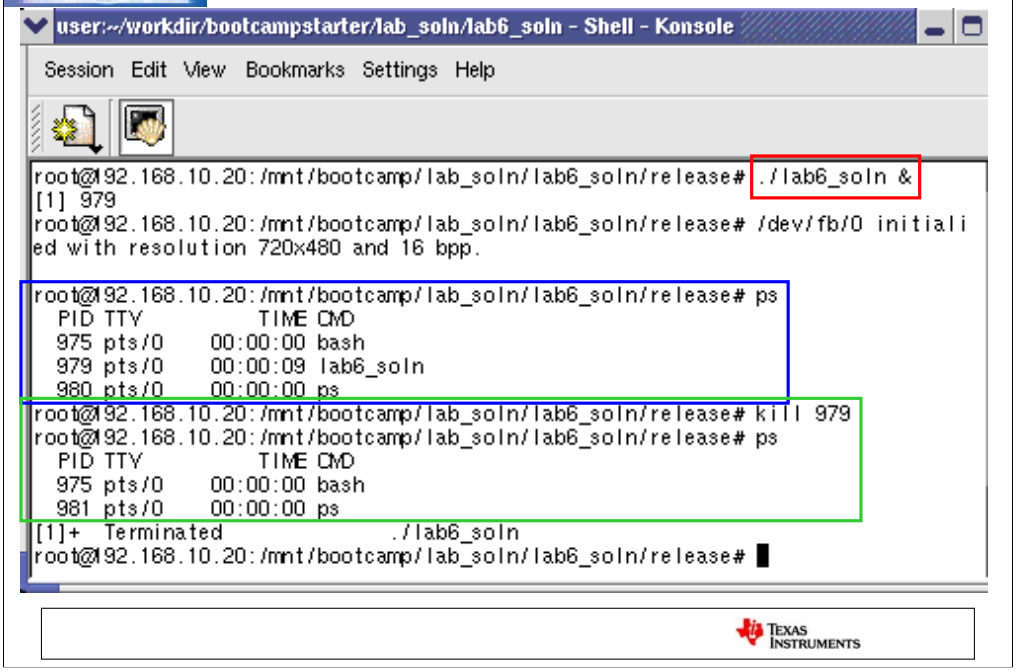
This slide compares the Time-slicing scheduling methodology used in Linux to an operating system such as BIOS which uses Blocking only, without time slicing.

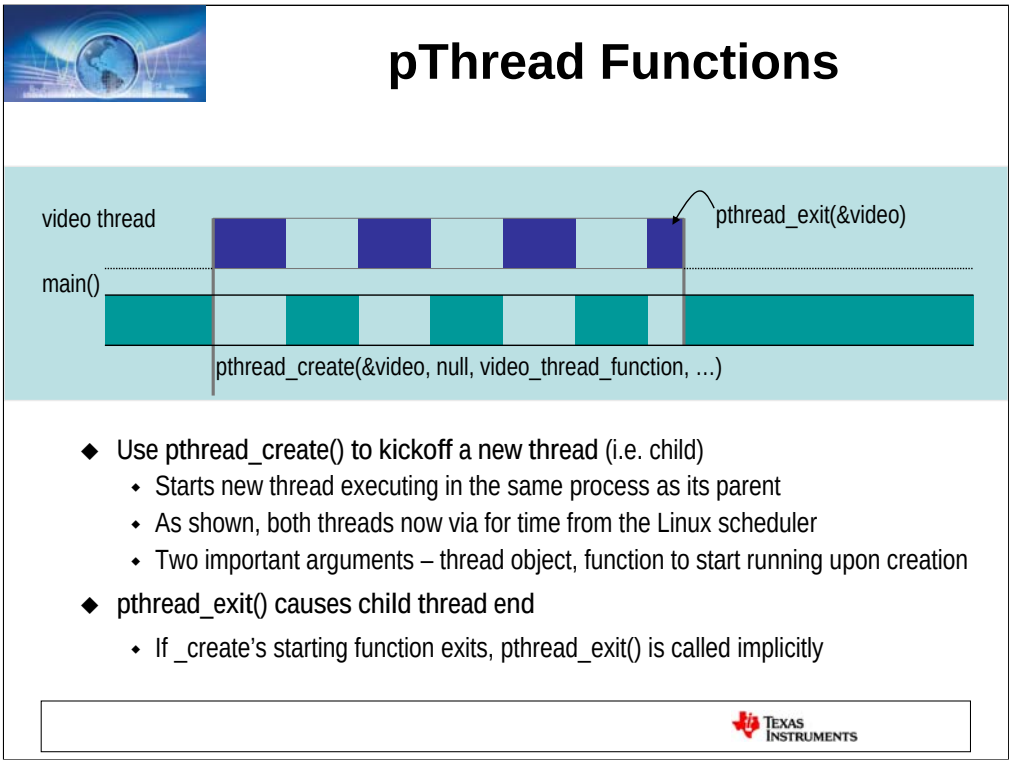
The advantage of the time slicing model is that it bends but doesn't break – i.e. even when the processor becomes overloaded, no threads completely starve. This is the typical functionality of the desktop systems that linux was developed for. When too many applications are opened, the system slows, but all of the applications continue to run.

The reason that BIOS does not use time slicing is that it is a true Real-Time operating system, developed specifically for DSPs. The main advantage of using Blocking only without time slicing is determinism. The highest priority threads in the system are guaranteed the processor bandwidth they need, therefore they are guaranteed to meet real-time, no matter how many lower priority threads are added into the system. The downside is that in an overloaded system, the lower priority threads may be given no CPU time at all, i.e. starvation. Because of this, it is very necessary that threads in a blocking-only scheduler, especially high priority threads, are good citizens and yield processor time by blocking themselves when they do not need processor time (as opposed to using polling or spin-loops), whereas a Time-Slicing scheduler such as linux will have performance loss due to non-good-citizen threads, but will not be as adversely effected.

Both methodologies have their place, and in most cases, an optimal configuration is one such as DaVinci with Linux on the ARM (where user interface will run) and BIOS on the DSP (where Real-time algorithms will run)

Launching a Process – Terminal





The process for a thread to create another thread is fairly simple.

Recall that the entry point for each process is main(), and that this entry point defines the implicit or originating thread of execution for a process. The main() thread may then create new threads by using the ptherad_create function call. Likewise, created threads can themselves create new threads using the same call.

The example above shows the main() thread, which exists at time 0, and at some time later uses pthread_create to spawn a new thread. The pthread_create function takes four arguments:

pthread_t *thrad – An empty thread object is passed by reference in the function call. This object must be declared before the function is called to allocate memory for it, but does not need to be initialized. The pthread_create function call initializes the object.

pthread_attr_t *attr – a pointer to an attributes structure for the thread to be created. Pass a NULL pointer to use default attributes.

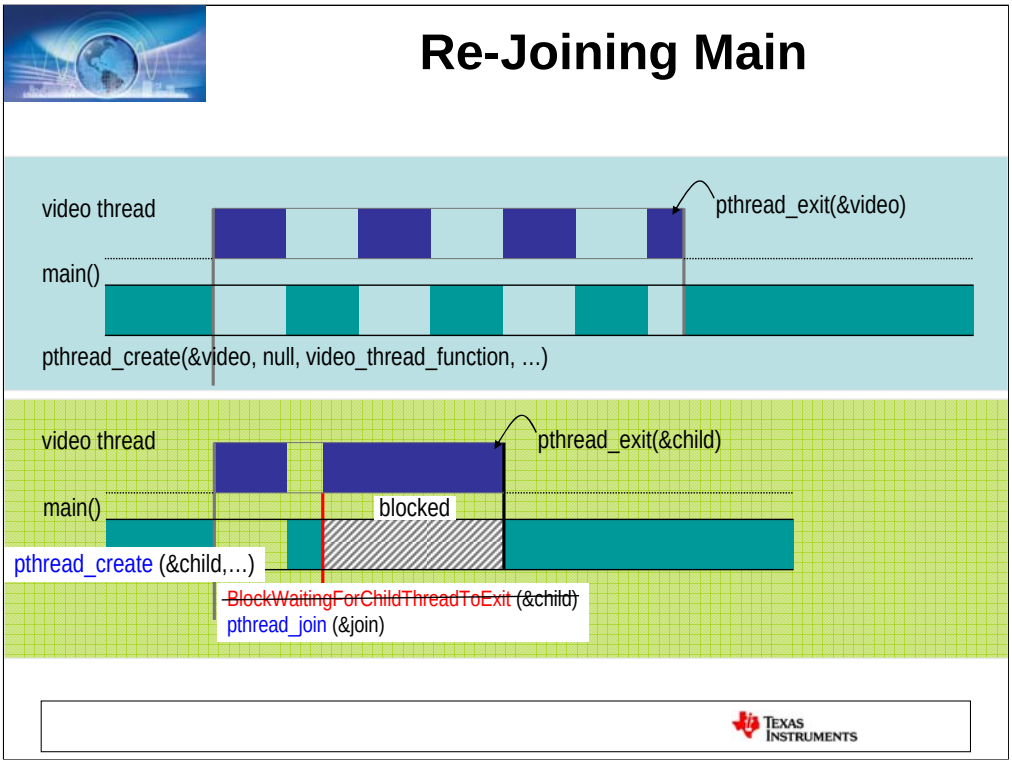
void *(*start_routine)(void *) – this is a pointer to a function. The function specified here is the entry point for the thread to be created. A thread function must have a single argument, a void pointer, and returns a void pointer. The use of void pointers allows a pointer of any type to be passed or returned, even a pointer to a structure, so that, in effect, this function can have any arguments or return values the programmer requires.

void *arg – this is the void pointer that will be passed to the start_routine function as its argument at the thread's entry point. Again, use of void * allows passing of a pointer to a structure so that any number of arguments may actually be passed.

In the example above, after the new thread is created the two run concurrently for some amount of time. (Again, recall that they are not truly running concurrently, but only appear to due to time slicing.) Some time later, the main thread calls pthread_join. The effect of this function call is to block the thread's execution until another thread has exited (in this case "new thread")

The pthread_join function call is simple, taking as its first argument the thread which must exit before the currently executing thread may resume. In this case that would be "new thread". The second argument is a pointer to a pointer, which is really just a void * passed by reference so that it may be modified. The void * is passed by reference and on return of ptherad_join will have the return value of the thread which just exited.

Which brings about the final function. In order to exit, i.e. terminate, a thread calls pthread_exit. In addition to terminating the thread, this function allows the terminating thread to pass a return value (which is a null *). This value is what is returned by pthread_join.



The process for a thread to create another thread is fairly simple.

Recall that the entry point for each process is main(), and that this entry point defines the implicit or originating thread of execution for a process. The main() thread may then create new threads by using the ptherad_create function call. Likewise, created threads can themselves create new threads using the same call.

The example above shows the main() thread, which exists at time 0, and at some time later uses pthread_create to spawn a new thread. The pthread_create function takes four arguments:

pthread_t *thrad – An empty thread object is passed by reference in the function call. This object must be declared before the function is called to allocate memory for it, but does not need to be initialized. The pthread_create function call initializes the object.

pthread_attr_t *attr – a pointer to an attributes structure for the thread to be created. Pass a NULL pointer to use default attributes.

void *(*start_routine)(void *) – this is a pointer to a function. The function specified here is the entry point for the thread to be created. A thread function must have a single argument, a void pointer, and returns a void pointer. The use of void pointers allows a pointer of any type to be passed or returned, even a pointer to a structure, so that, in effect, this function can have any arguments or return values the programmer requires.

void *arg – this is the void pointer that will be passed to the start_routine function as its argument at the thread's entry point. Again, use of void * allows passing of a pointer to a structure so that any number of arguments may actually be passed.

In the example above, after the new thread is created the two run concurrently for some amount of time. (Again, recall that they are not truly running concurrently, but only appear to due to time slicing.) Some time later, the main thread calls pthread_join. The effect of this function call is to block the thread's execution until another thread has exited (in this case "new thread")

The pthread_join function call is simple, taking as its first argument the thread which must exit before the currently executing thread may resume. In this case that would be "new thread". The second argument is a pointer to a pointer, which is really just a void * passed by reference so that it may be modified. The void * is passed by reference and on return of ptherad_join will have the return value of the thread which just exited.

Which brings about the final function. In order to exit, i.e. terminate, a thread calls pthread_exit. In addition to terminating the thread, this function allows the terminating thread to pass a return value (which is a null *). This value is what is returned by pthread_join.

TTO

Technical Training Organization

Texas Instruments

Backup

Linux Command Summary

File Management
- ls and ls –la
- cd
- cp
- mv
- pwd
- tar (create, extract tar and tar.gz files)
- chmod
- chown
- mkdir
- mount, umount (in general, what is "mounting" and how do you do it?)
- alias
- touch

Network
- /sbin/ifconfig, ifup, ifdown
- ping
- nfs (What is it? How to share a folder via NFS. Mounting via NFS.)

VMware Shared Folders
- /mnt/hgfs/<*shared name*>

Program Control
- <ctrl>-c
- ps
- kill

Linux Users
- root
- user
- su (... exit)

BASH
- What is BASH scripting
- What are environment variables
- How to set the PATH environment variable
- What is .bashrc? (like DOS autoexec.bat)
- man pages
- change command line prompt

Texas Instruments

Outline

- 1:Format Highlight List Item Here
- 2:Format Regular List Items Here

Texas Instruments