Linux Kernel Module Programming

Anandkumar

June 5, 2021

Introduction

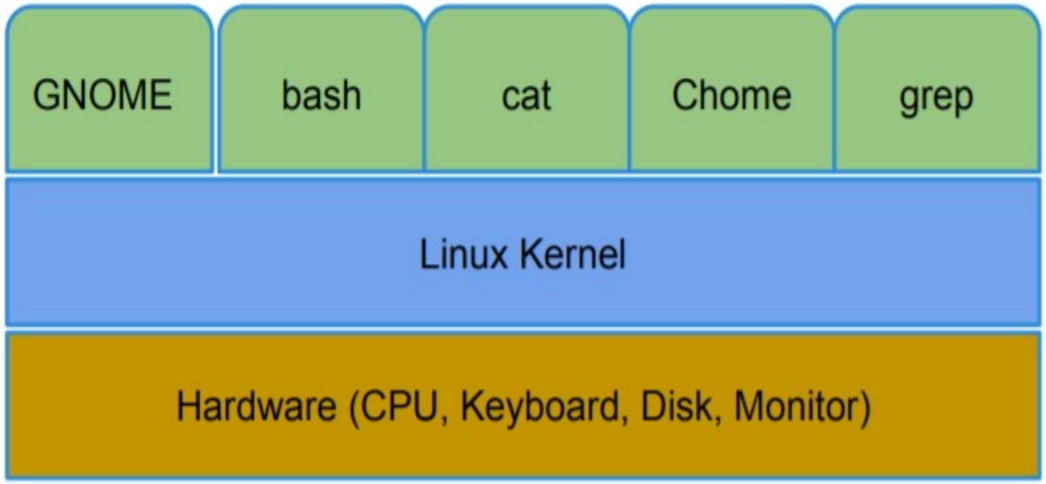
- Similar to Java Servlets with predefined structure and function names
- Code that executes as part of the Linux kernel
- Extends the capabilities and sometimes might modify the behavior of the kernel

Introduction Cont…

- Common C functions are not available to modules (such as printf, scanf, strcat, or system)
- Instead you must use kernel defined functions (/proc/ksyms)

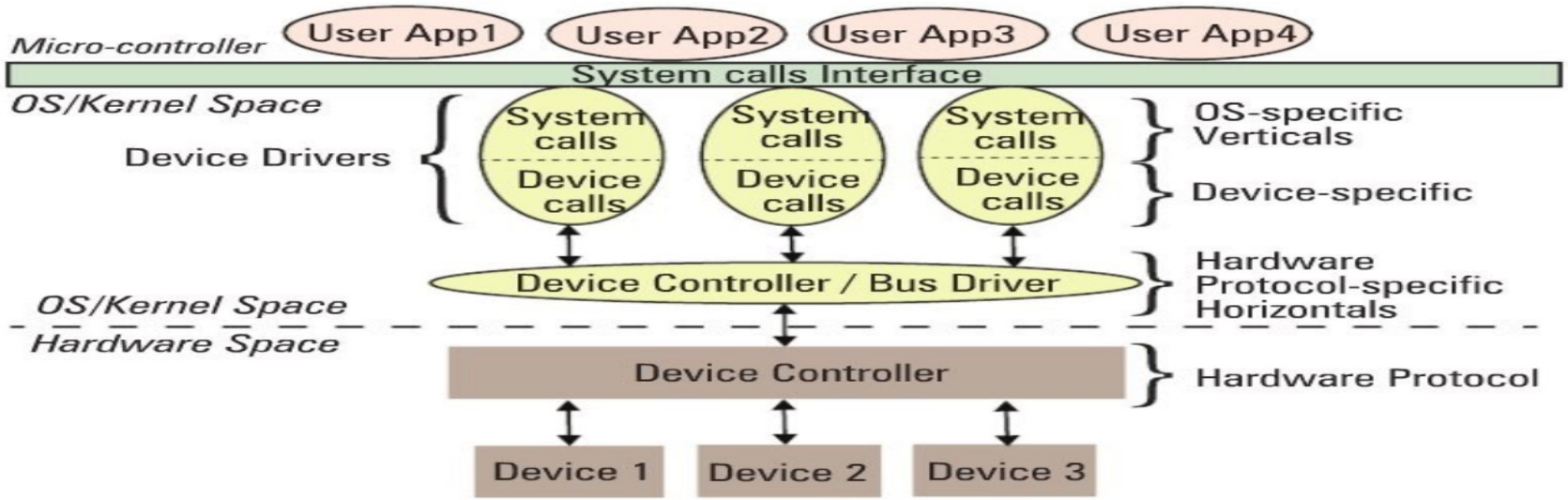
What Is the Kernel?

Very simple answer: it's a program that makes your hardware look and feel like an OS to other programs

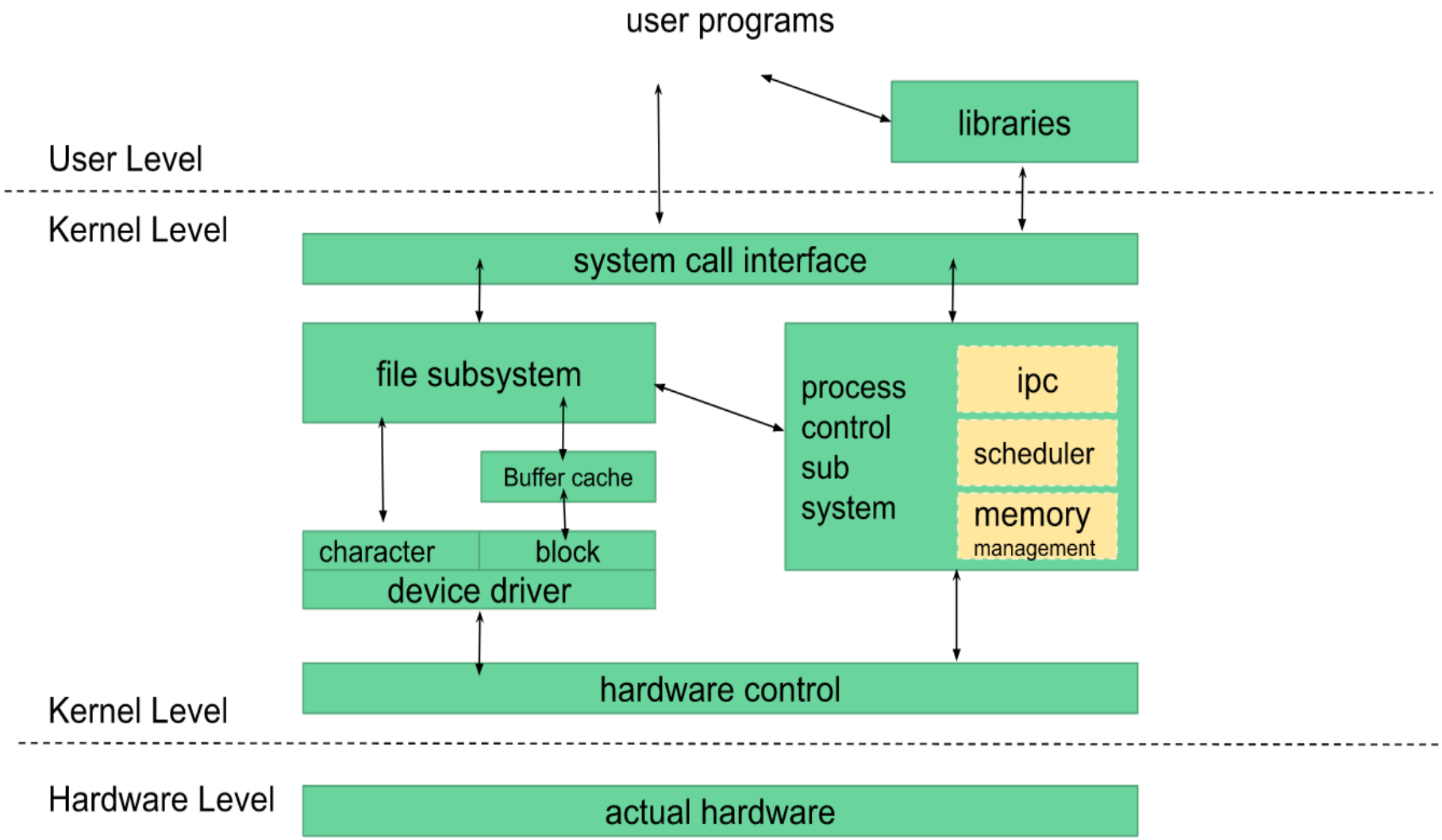


UserSpace Vs KernelSpace

- The Concept of kernel space and user space is all about memory and access rights.
- It is a feature of modern CPU, allowing it to operate either in privileged or unprivileged mode.
- One ,may consider kernel to be privileged and user apps are restricted.

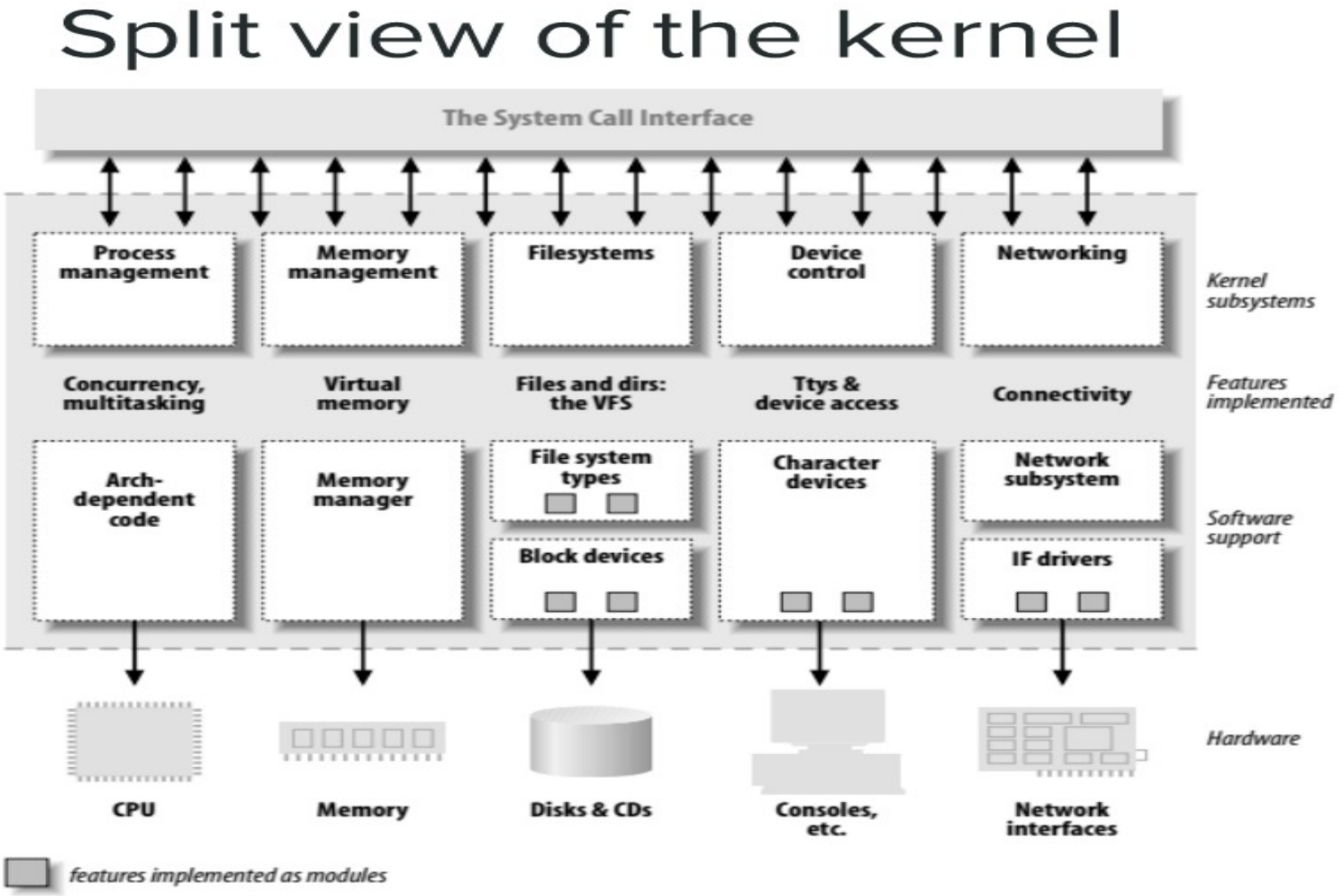


UserSpace Vs KernelSpace



Block Diagram of the System Kernel

Split view of kernel…



Classification of Drivers

- **Packet oriented (network drivers)**

  Network interface card, WiFi, Ethernet etc.

- **Block oriented (block storage drivers)**

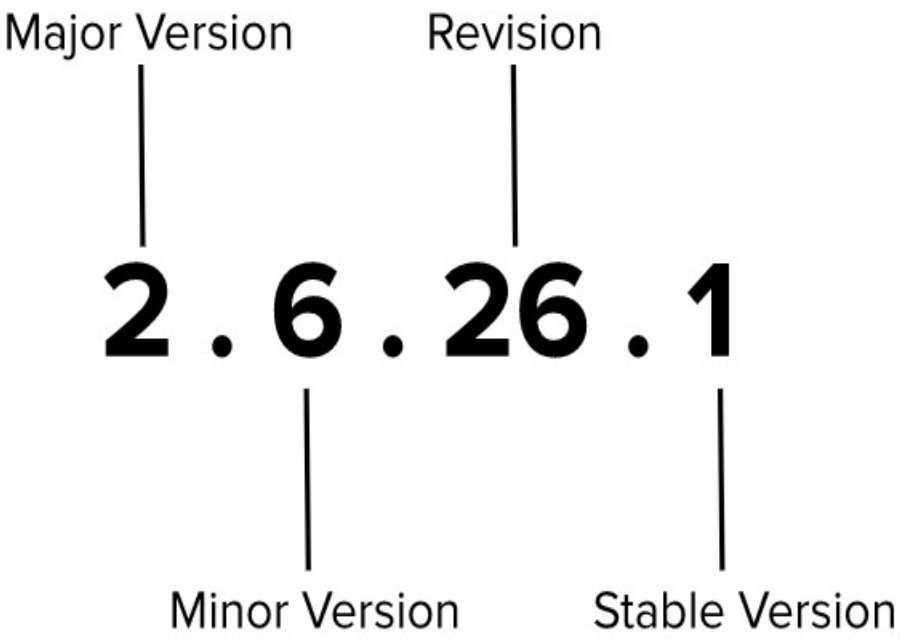
  File system driver to decode file formats & block storage device drivers

- **Byte oriented (character drivers)**

  Driver for everything else, I2C, TTY, VGA, Sound, SPI, SMBus etc.

- **Drivers for mixed data type device, like USB, PCI etc has characteristics of all of the above.**

Linux kernel versions



The kernel source tree

Browse full kernel source at https://github.com/torvalds/linux

directories in the root of kernel source tree

Directory	Description
arch	Architecture-specific source
block	Block I/O layer
crypto	Crypto API
Documentation	Kernel source documentation
drivers	Device drivers
firmware	Device firmware needed to use certain drivers
fs	The VFS and the individual filesystems
include	Kernel headers
init	Kernel boot and initialization
ipc	Interprocess communication code
kernel	Core subsystems, such as the scheduler
lib	Helper routines
mm	Memory management subsystem and the VM
net	Networking subsystem
samples	Sample, demonstrative code
scripts	Scripts used to build the kernel
security	Linux Security Module
sound	Sound subsystem
usr	Early user-space code (called initramfs)
tools	Tools helpful for developing Linux
virt	Virtualization infrastructure

Standard Functions

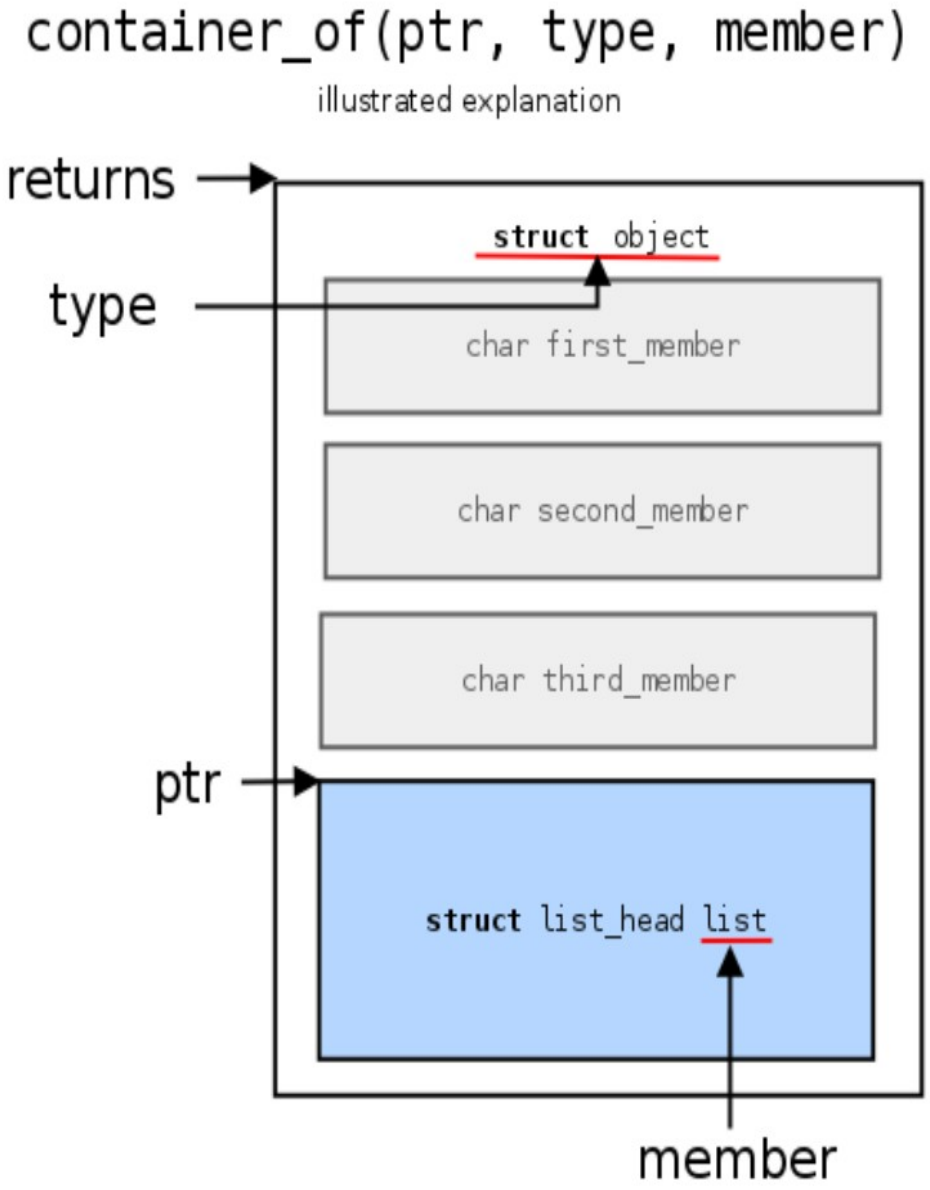
- printk ()
  - 8 Priority Levels: KERN_WARNING, KERN_ALERT
  - Kernel modules cannot call any functions in the C runtime library (e.g., ‘printf()’)
  - But similar kernel versions of important functions are provided (e.g., ‘printk()’)
  - Syntax and semantics are slightly different (e.g., priority and message-destination)
  - Capabilities may be somewhat restricted (e.g., printk() can’t show floating-point)

Standard Functions cont…

- Init function sets up variables, tells the kernel what the module provides and registers any necessary functions
- Cleanup function undoes changes made by init

container_of()

cast a member of a structure out to the containing structure



Standard Functions Cont…

- int init_module(void)
  - Called when the kernel loads your module.
  - Initialize all your stuff here.
  - Return 0 if all went well, negative if something blew up.
- void cleanup_module(void)
  - Called when the kernel unloads your module.
  - Free all your resources here.

A minimal module-template

```
#include <linux/module.h>
 int init_module( void )
 {
        // code here gets called during module installation
 }
 void cleanup_module( void )
 {
        // code here gets called during module removal
 }
MODULE_LICENSE(“GPL”);
```

Setup Environment

- Install Kernel Module Tools

  # sudo apt-get install kmod
- Install development tools and kernel headers

  # sudo apt-get install build-essential

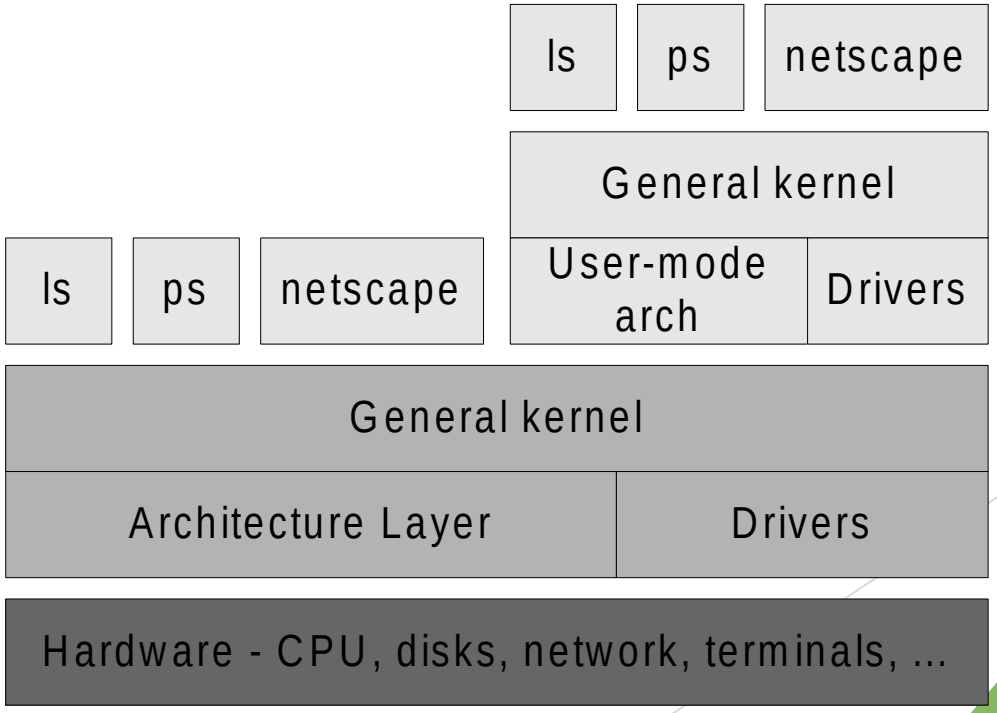
  # sudo apt-get install linux-headers-$(uname -r)

User-Mode Linux

- What is User-Model Linux?
  - Ported to run itself as a set of Linux (non-root user) process on itself.
  - Short form as UML
  - UML started by Jeff Dike since Feb 1999
  - Use simulated hardware i.e., services provided by host kernel.
  - Run (nearly) all applications and services available on the host architecture.

User-Mode Linux

- *Provides a self-contained environment:*
  - *Identical as hosting Linux kernel,*
  - *Processes have no access to host resources that were not explicitly provided*
- *Layer diagram:*



User-Mode Linux

- All UML devices are virtual; constructed from the abstractions provided by the host kernel.
- The UML devices:
  - Consoles and serial lines
    - Main console, virtual console and serial lines.
  - Block devices
    - Access to anything on the host which can be mounted, e.g. CDROM, disk partition
  - Network devices
    - Network access between UML to host OS/UML and between UML to host in outside network.

User-Mode Linux

- Focus of the presentation:
  - Applications
  - Design and Implementation
  - Some mentioned future works

User-Mode Linux Applications

- Kernel debugging
  - on top of software OS, not on separate test machine.
  - standard suite of process debugging tools such as gdb, gcov and gprof can be utilized.
- Prototyping
  - Realizing virtual network with a single physical network.
  - Testing software configuration.

User-Mode Linux Applications

- *Isolation*
  - *separating users/applications of virtual machines from each other and from the host.*
  - *purposes:*
    - *Against possibly hostile/untrusted processes,*
    - *performance control - allocation of resources (e.g. CPU, memory, disk space), avoidance of race*
- *Multiple environments*
  - *Especially for Linux applications, some incompatibility problems may exists and need to test.*
- *A Linux environment for other operating systems*
  - *Many Linux applications are free and open-source. With UML, those applications can run upon other hosting OS like MS Windows.*

User-Mode Linux Setup

- Install Dependency

  # sudo apt-get install build-essential flex bison

  # sudo apt-get install xz-utils wget

  # sudo apt-get install ca-certificates bc

  # sudo apt-get install libncurses-dev

- Downloading the Kernel and extract

  www.kernel.org

  # tar -xvzf linux-5.1.16.tar.gz

User-Mode Linux Setup

- *Configuring the Kernel*

  # make ARCH=um menuconfig

```
UML-specific Options:
  - Host filesystem
Networking support (enable this to get the submenu to show up):
  - Networking options:
    - TCP/IP Networking
UML Network devices:
  - Virtual network device
  - SLiRP transport
```

User-Mode Linux Setup

- Build Kernel

  # make ARCH=um

  # mkdir ../bin

  # cd ../bin

  # cp ../linux-5.1.16/linux .

- Setup Guest file system

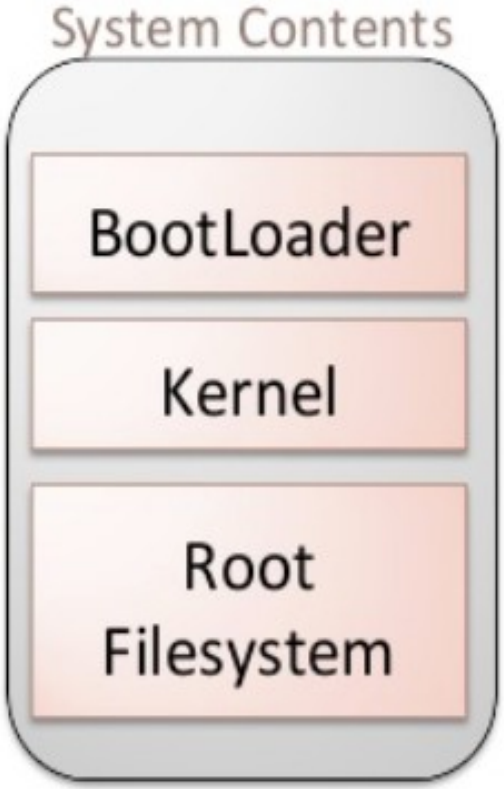
  # tar –xvzf alpine-rootfs.tgz

- Boot system

  # ./bin/linux rootfstype=hostfs rootflags=/home/test/uml/rootfs/ rw mem=64M init=/bin/sh

Rootfs

- The root filesystem, is mounted as /
- On embedded systems, this root filesystem contains all the libraries, applications and data of the system
- Therefore, building the root filesystem is one of the main tasks of integrating embedded Linux components into a device
- The kernel is usually kept separate



User-Mode Linux
Setup with custom rootfs

- Download busybox and untar

  https://busybox.net/downloads/busybox-1.33.1.tar.bz2

  # tar –xvjf busybox-1.33.1.tar.bz2

- Configure Busybox

  # make menuconfig

  Enable static build option

- Build Busybox

  # make

User-Mode Linux
Setup with custom rootfs

- Install Busybox

  # make install

- Using this rootfs to boot

  # ./bin/linux rootfstype=hostfs rootflags=/home/test/uml/rootfs_busybox/ rw mem=64M init=/bin/sh

Qemu Setup

- Install Dependency

  # sudo apt-get install qemu-system

- Download Cross compiler and install it

  # tar -xvf gcc-linaro-10.2.1-2021.02-x86_64_arm-linux-gnueabihf.tar.xz

- Downloading the Kernel and extract

  www.kernel.org

  # tar -xvzf linux-5.1.16.tar.gz

Qemu Setup

- Copy the default configuration file

  #cp arch/arm/configs/vexpress_defconfig .config

- Configuring the Kernel

  # make ARCH=arm menuconfig

- Cross compile Kernel

  # make ARCH=arm CROSS_COMPILE=/home/test/qemu/gcc-linaro-10.2.1-2021.02-x86_64_arm-linux-gnueabihf/bin/arm-linux-gnueabihf-

Qemu Setup

- Boot

```
# qemu-system-arm -M vexpress-a9 -kernel vmlinuz-3.2.0-4-vexpress -
    initrd initrd.img-3.2.0-4-vexpress -drive
    if=sd,file=debian_wheezy_armhf_standard.qcow2 -append
    "root=/dev/mmcblk0p2"
```

Kernel module command

- lsmod

  List Modules that Loaded Already
- insmod

  Insert Module into Kernel
- modinfo

  Display Module Info
- rmmod

  Remove Module from Kernel
- modprobe

  Add or Remove modules from the kernel

Example1

```
/* example1.c - The simplest kernel module. */
#include <linux/module.h> /* Needed by all modules */
#include <linux/kernel.h> /* Needed for KERN_ALERT */
#include <linux/init.h>   /* Needed for macros */

int example1_init(void) {
  printk(KERN_ALERT "Initing example1\n"); /* A non 0 return
   means init_module failed; module can't be loaded. */
  return 0;
}

void example1_stop(void) {
  printk(KERN_ALERT "Unloading example1\n");
}

module_init (example1_init);
module_exit (example1_stop);
```

Example 1 cont…

- **Accompany your module with a 1-line GNU Makefile:**
  - `obj-m += example1.o`
- **Run the magic make command:**
  - `make -C /lib/modules/$(uname -r)/build M=$(pwd) modulesProduces: example1.ko`
- **To manually load your module:**
  - `insmod example1.ko`
- **Where's our message?**
  - `dmesg`
- **To unload your module:**
  - `rmmod example1`

Handling Runtime Parameters

- module_param() , module_param_array() and module_param_string().

  The macro takes 3 arguments: the name of the variable, its type and permissions for the corresponding file in sysfs.

More Kernel Macros

- MODULE_LICENSE ()
- MODULE_AUTHOR ()
- MODULE_DESCRIPTION ()
- MODULE_SUPPORTED_DEVICE ()

Example 2

```
/*
 *  example2.c - Demonstrates command line argument passing to
 *  a module.
 */
#include <linux/module.h>
#include <linux/moduleparam.h>
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/stat.h>

MODULE_LICENSE("GPL");
MODULE_AUTHOR("test");
MODULE_DESCRIPTION ("A module to teach module writing");
MODULE_SUPPORTED_DEVICE ("The mind");

static short int myshort = 1;
static int myint = 420;
```

Example 2

```
static long int mylong = 9999;
static char *mystring = "blah";
static int myintArray[2] = { -1, -1 };

/*
 * module_param(foo, int, 0000)
 * The first param is the parameters name
 * The second param is it's data type
 * The final argument is the permissions bits,
 * for exposing parameters in sysfs (if non-zero) at a later
 * stage.
 */
module_param(myshort, short, S_IRUSR | S_IWUSR | S_IRGRP |
   S_IWGRP);
MODULE_PARM_DESC(myshort, "A short integer");
module_param(myint, int, S_IRUSR | S_IWUSR | S_IRGRP |
   S_IROTH);
```

Example 2

```
MODULE_PARM_DESC(myint, "An integer");
module_param(mylong, long, S_IRUSR);
MODULE_PARM_DESC(mylong, "A long integer");
module_param(mystring, charp, 0000);
MODULE_PARM_DESC(mystring, "A character string");
static int arr_argc = 0;

module_param_array(myintArray, int, &arr_argc, 0000);
MODULE_PARM_DESC(myintArray, "An array of integers");
```

Example 2

```
static int __init example2_init(void)
{
   int i;
   printk(KERN_INFO "Entering Example2\n============\n");
   printk(KERN_INFO "myshort is a short integer: %hd\n",
   myshort);
   printk(KERN_INFO "myint is an integer: %d\n", myint);
   printk(KERN_INFO "mylong is a long integer: %ld\n",
   mylong);
   printk(KERN_INFO "mystring is a string: %s\n", mystring);

   for (i = 0; i < (sizeof myintArray / sizeof (int)); i++)

   {
      printk(KERN_INFO "myintArray[%d] = %d\n", i,
   myintArray[i]);
   }
   printk(KERN_INFO "got %d arguments for myintArray.\n",
   arr_argc);
   return 0;
}
```

Example 2

```
static void __exit example2_exit(void)
{
   printk(KERN_INFO "Exiting Example2\n");
}

module_init(example2_init);
module_exit(example2_exit);
```

Module v0.2 cont..

Try to compile on your own and run it??

Example 2 cont…

- **Accompany your module with a 1-line GNU Makefile:**
  - **obj-m += example2.o**
- **Run the magic make command:**
  - **make -C /lib/modules/$(uname -r)/build M=$(pwd) modulesProduces: example2.ko**
- **To manually load your module:**
  - **insmod example2.ko**
- **Where’s our message?**
  - **dmesg**
- **To unload your module:**
  - **rmmod example2**

UML Kernel module building

- **Enable module support in kernel**

- **Build kernel module**

```
# make -C /home/test/uml2/linux-5.1.16/ M=$
  (pwd) ARCH=um
```

- **Copy to Rootfs**

- **Boot uml**

```
# ./linux-5.1.16/linux  rootfstype=hostfs
  rootflags=/home/test/uml2/rootfs_busybox/ rw
  mem=64M  init=/linuxrc
```

- **Insert module**

UML Debugging

- **Enable debugging symbols**
- **Use gdb to debug and start debugging kernel**

jiffies

- unsigned long volatile jiffies;
- global kernel variable (used by scheduler)
- Initialized to zero when system reboots
- Gets incremented when timer interrupts
- So it counts ‘clock-ticks’ since cpu restart
- ‘tick-frequency’ is architecture dependent

jiffies

- Write kernel module to display jiffies value on installation of kernel module?

jiffies

```
/*
 *  example_jiffies.c
 */
#include <linux/module.h>          /* Needed by all modules */
#include <linux/kernel.h>          /* Needed for KERN_INFO */
#include <linux/jiffies.h>

int init_module(void)
{
        printk(KERN_INFO "Entering Jiffies Example\n");
        printk(KERN_INFO "The jiffies value=%ld\n",jiffies);

        return 0;
}

void cleanup_module(void)
{
        printk(KERN_INFO "Exiting Jiffies Example\n");
}
```

Jiffies cont…

- **Accompany your module with a 1-line GNU Makefile:**
  - **obj-m += example_jiffies.o**
- **Run the magic make command:**
  - **make -C /lib/modules/$(uname -r)/build M=$(pwd) modulesProduces: example_jiffies.ko**
- **To manually load your module:**
  - **insmod example_jiffies.ko**
- **Where’s our message?**
  - **dmesg**
- **To unload your module:**
  - **rmmod example_jiffies**

Working with multiple files

```
/* file1.c - The simplest kernel module. */
#include <linux/module.h> /* Needed by all modules */
#include <linux/kernel.h> /* Needed for KERN_ALERT */
#include <linux/init.h>   /* Needed for macros */

int example1_init(void) {
  printk(KERN_ALERT "Initing example1\n"); /* A non 0 return means
init_module failed; module can't be loaded. */
  return 0;
}

module_init (example1_init);
```

Working with multiple files

```
/* file2.c - The simplest kernel module. */
#include <linux/module.h> /* Needed by all modules */
#include <linux/kernel.h> /* Needed for KERN_ALERT */
#include <linux/init.h>   /* Needed for macros */

void example1_stop(void) {
  printk(KERN_ALERT "Exiting example1\n");
}

module_exit (example1_stop);
```

Working with multiple files

- Sources exists in two files file1.c and file2.c

```
obj-m += multiple.o
multiple-objs += file1.o file2.o
```

Dependency between modules

```
/* dep1.c - The simplest kernel module. */
#include <linux/module.h> /* Needed by all modules */
#include <linux/kernel.h> /* Needed for KERN_ALERT */
#include <linux/init.h>   /* Needed for macros */

void dep1_func(void){
                printk(KERN_INFO "I am in dep1 func\n");
}

int dep1_init(void) {
  printk(KERN_INFO "Initing dep1\n"); /* A non 0 return means init_module failed; module can't be loaded. */
  return 0;
}

void dep1_stop(void) {
  printk(KERN_INFO "Exiting dep1\n");
}

module_init (dep1_init);
module_exit (dep1_stop);
EXPORT_SYMBOL(dep1_func);
```

Dependency between modules

```
/* dep2.c - The simplest kernel module. */
#include <linux/module.h> /* Needed by all modules */
#include <linux/kernel.h> /* Needed for KERN_ALERT */
#include <linux/init.h>   /* Needed for macros */

extern void dep1_func(void);

int dep2_init(void) {
  printk(KERN_INFO "Initing dep2\n"); /* A non 0 return means init_module failed; module can't be loaded. */
  dep1_func();
  return 0;
}

void dep2_stop(void) {
  printk(KERN_INFO "Exiting dep2\n");
}

module_init (dep2_init);
module_exit (dep2_stop);
```

Dependency between modules

- **Accompany your module with a 2-line GNU Makefile:**
  - **obj-m += dep1.o**
    obj-m += dep2.o
- **Run the magic make command:**
  - **make -C /lib/modules/$(uname -r)/build M=$(pwd) modulesProduces: dep1.ko,dep2.ko**
- **Copy module to /lib/modules/5.8.0-55-generic/ manually:**
  - **cp dep1.ko dep2.ko /lib/modules/5.8.0-55-generic/**
- **Run depmod**
  - **Depmod -a**
- **To insert module:**
  - **Modprobe dep2**
- **To remove module:**
  - **Modprobe –r dep2**

Character Driver

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/fs.h>
#include <linux/uaccess.h>

#define DEVICE_NAME     "testchar"
static int num_major;

static int test_open(struct inode *inode, struct file *file)
{
        printk(KERN_INFO "test_open called\n");
        return 0;
}

static int test_release(struct inode *inode, struct file *file)
{
        printk(KERN_INFO "test_release called\n");
        return 0;
}
```

Character Driver

```
static ssize_t test_read(struct file *filp,
                         char *buf,
                         size_t len,
                         loff_t * off)
{
        printk(KERN_INFO "test_read called\n");
        return 0;
}

static ssize_t test_write(struct file *filp,
                          const char *buf,
                          size_t len,
                          loff_t * off)
{

        printk(KERN_INFO "test_write called\n");
        return 0;
}
```

Character Driver

```
static struct file_operations fops= {
        .open = test_open,
        .release = test_release,
        .read = test_read,
        .write = test_write,
};


static int __init test_init(void)
{

        printk(KERN_INFO "Entering Test Character Driver \n");

        num_major=register_chrdev(0, DEVICE_NAME, &fops);
        printk(KERN_INFO "Major Number = %d \n",num_major);
        printk(KERN_INFO "Name =  %s \n",DEVICE_NAME);
        printk(KERN_INFO "Generate the device file with\
                mknod /dev/%s c %d 0 \n",DEVICE_NAME,num_major);

        return 0 ;

}
```

Character Driver

```
static void __exit test_cleanup(void)
{
	unregister_chrdev(num_major, DEVICE_NAME);
	printk(KERN_INFO "Exiting Test Character Driver \n");
}

module_init(test_init);
module_exit(test_cleanup);


MODULE_LICENSE("GPL");
MODULE_AUTHOR("test");
MODULE_DESCRIPTION("Character Device Driver");
MODULE_SUPPORTED_DEVICE("testchar");
```

Character Driver

- **Accompany your module with a 1-line GNU Makefile:**
  - **obj-m += example_character.o**
- **Run the magic make command:**
  - **make -C /lib/modules/$(uname -r)/build M=$(pwd) modulesProduces: example_character.ko**
- **To manually load your module:**
  - **insmod example_character.ko**
- **Where's our message?**
  - **dmesg**
- **To unload your module:**
  - **rmmod example_character**
- **Test Driver functionality using shell command:**

Proc filesysystem

```
#include <linux/module.h>
#include <linux/proc_fs.h>
#include <linux/seq_file.h>
#include <linux/jiffies.h>

static int hello_proc_show(struct seq_file *m, void *v) {
  seq_printf(m, "jiffies = %lu\n", jiffies );
  return 0;
}

static int hello_proc_open(struct inode *inode, struct
  file *file) {
  return single_open(file, hello_proc_show, NULL);
}
```

Proc filesysystem

```
static const struct proc_ops hello_proc_fops={
    .proc_open = hello_proc_open,
    .proc_release = single_release,
    .proc_read = seq_read,
    .proc_lseek = seq_lseek
};

static int __init hello_proc_init(void) {
  proc_create("jiffies_test", 0, NULL, &hello_proc_fops);
  return 0;
}

static void __exit hello_proc_exit(void) {
  remove_proc_entry("jiffies_test", NULL);
}
```

Proc filesysystem

```
MODULE_LICENSE("GPL");
module_init(hello_proc_init);
module_exit(hello_proc_exit);
```

Proc filesystem

- **Accompany your module with a 1-line GNU Makefile:**
  - `obj-m += proc_test.o`
- **Run the magic make command:**
  - `make -C /lib/modules/$(uname -r)/build M=$(pwd) modulesProduces: proc_test.ko`
- **To manually load your module:**
  - `insmod proc_test.ko`
- **Where's our message?**
  - `Cat /proc/jiffies_test`
- **To unload your module:**
  - `rmmod proc_test`

__init

The __init macro tells the compiler that the associate function or variable is used only upon initialization. The compiler places all code marked with __init into a special memory section that is freed after the initialization phase ends:

```
-----------------------------------------------------------------------
drivers/char/random.c
 679 static int __init batch_entropy_init(int size, struct entropy_store *r)
-----------------------------------------------------------------------
```

__initdata,__exit,__exitdata

Similarly, if there is data that is used only during initialization, the data needs to be marked with `__initdata`. Here, we can see how `__initdata` is used in the ESP device driver:

```
-----------------------------------------------------------------------
drivers/char/esp.c
 107 static char serial_name[] __initdata = "ESP serial driver";
 108 static char serial_version[] __initdata = "2.2";
-----------------------------------------------------------------------
```

Also, the `__exit` and `__exitdata` macros are to be used only in the exit or shutdown routines. These are commonly used when a device driver is unregistered.

__initdata,__exit,__exitdata

Similarly, if there is data that is used only during initialization, the data needs to be marked with `__initdata`. Here, we can see how `__initdata` is used in the ESP device driver:

```
-------------------------------------------------------------------
drivers/char/esp.c
 107 static char serial_name[] __initdata = "ESP serial driver";
 108 static char serial_version[] __initdata = "2.2";
-------------------------------------------------------------------
```

Also, the `__exit` and `__exitdata` macros are to be used only in the exit or shutdown routines. These are commonly used when a device driver is unregistered.

likely(), unlikely()

The specific implementation of `likely()` and `unlikely()` are specified as follows:[4]

[4] `__builtin_expect()`, as seen in the code excerpt, is nulled before GCC 2.96, because there was no way to influence branch prediction before that release of GCC.

```
------------------------------------------------------------------------
include/linux/compiler.h
  45 #define likely(x)   __builtin_expect(!!(x), 1)
  46 #define unlikely(x)  __builtin_expect(!!(x), 0)
------------------------------------------------------------------------
```

likely(), unlikely()

- **In the following example, we are marking branch as likely true:**

```
const char *home_dir ;

home_dir = getenv("HOME");
if (likely(home_dir))
    printf("home directory: %s\n", home_dir);
else
    perror("getenv");
```

- **For above example, we have marked “if” condition as “likely()” true, so compiler will put true code immediately after branch, and false code within the branch instruction. In this way compiler can achieve optimization. But don’t use “likely()” and “unlikely()” macros blindly.**

likely(), unlikely()

- **If prediction is correct, it means there is zero cycle of jump instruction, but if prediction is wrong, then it will take several cycles, because processor needs to flush it’s pipeline which is worst than no prediction.**

Process Management

- Finding process information
  - ps
  - top
  - For each process, there is a corresponding directory /proc/ to store this process information in the /proc pseudo file system

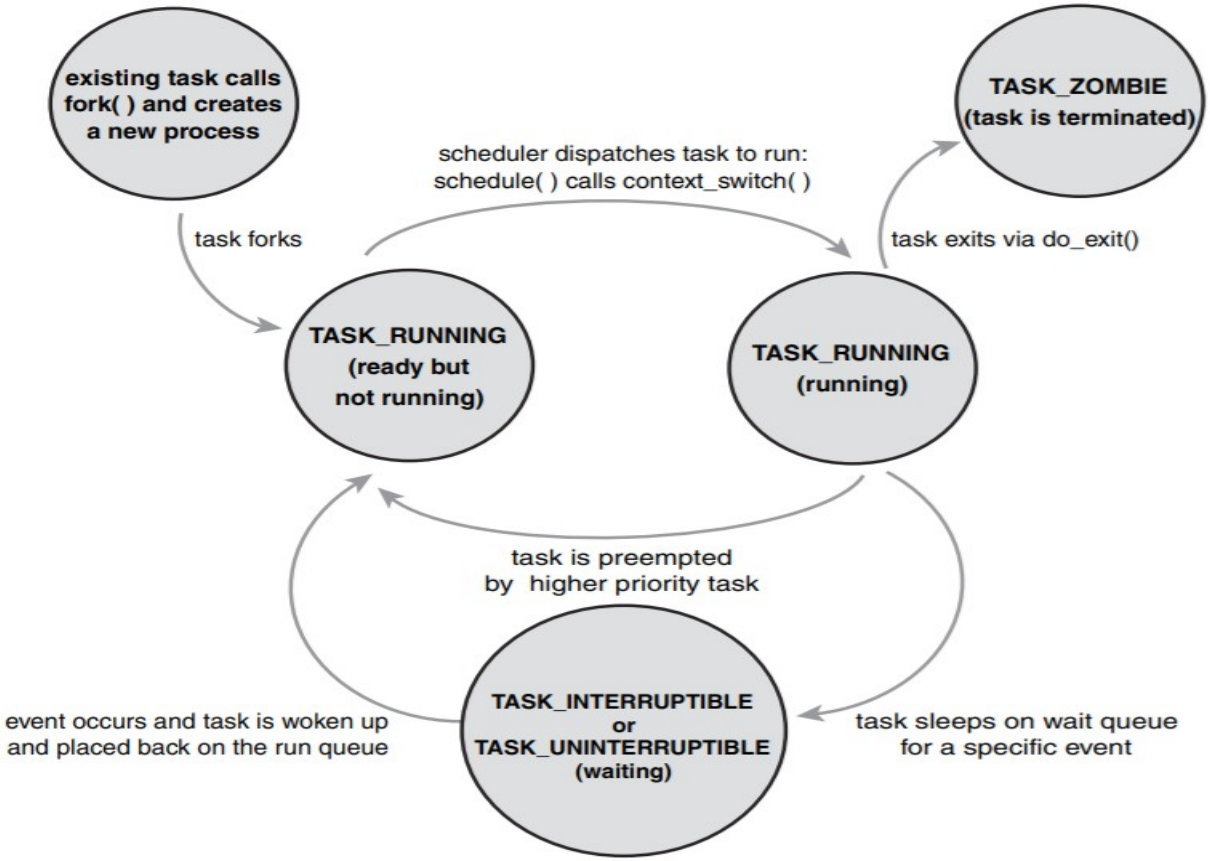
Process Management

- **Header files**
  - include/linux/sched.h – declarations for most process data structures
  - include/linux/wait.h – declarations for wait queues
  - include/asm-i386/system.h – architecture-dependent declarations

Process Management

- **Source files**
  - kernel/sched.c – process scheduling routines
  - kernel/signal.c – signal handling routines
  - kernel/fork.c – process/thread creation routines
  - kernel/exit.c – process exit routines
  - fs/exec.c – executing program
  - arch/i386/kernel/entry.S – kernel entry points
  - arch/i386/kernel/process.c – architecture-dependent process routines

Process Management

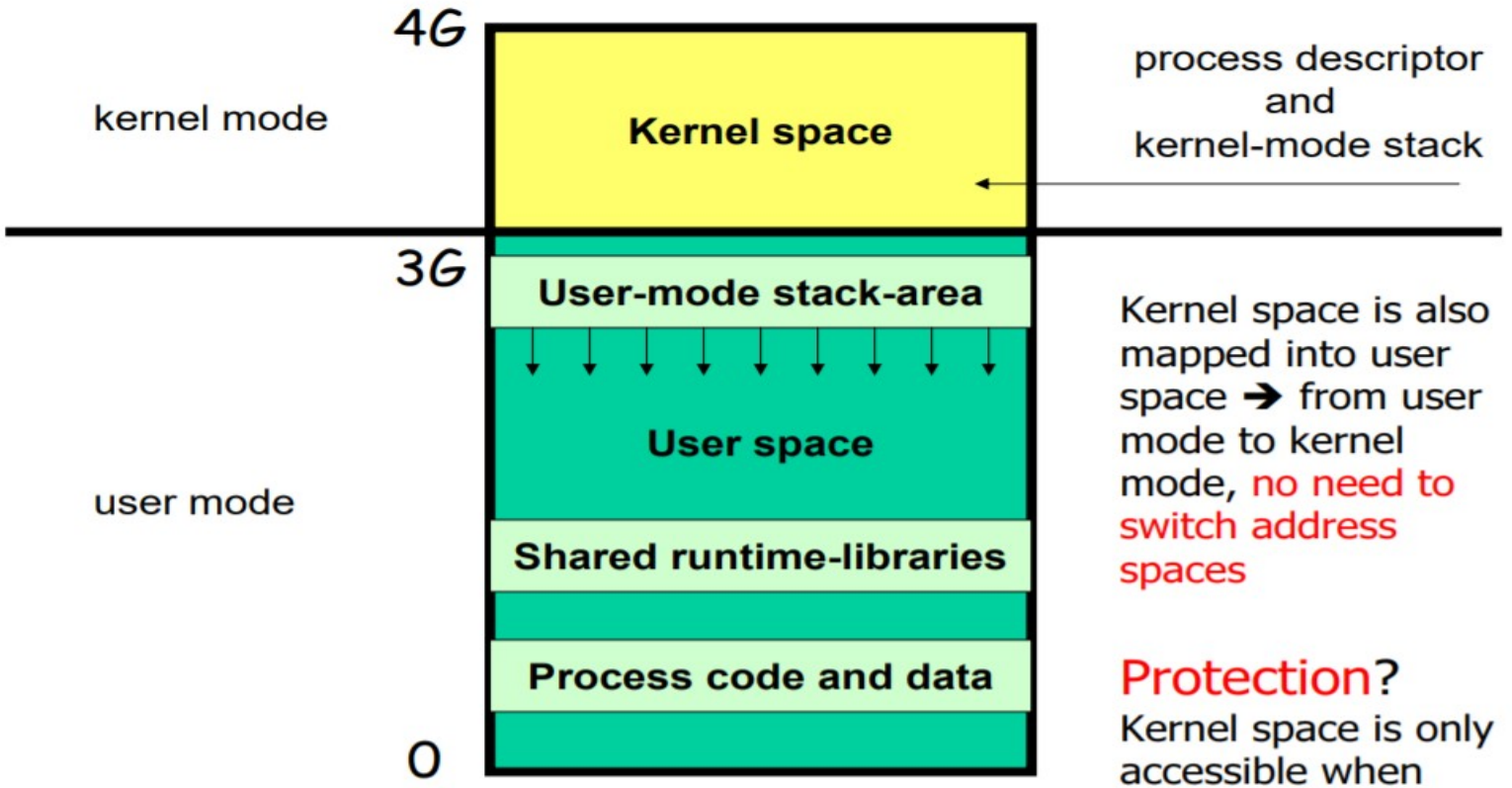


Process Management

- **Kernel needs work space as well**
  - ❖ **Store kernel code, data, heap, and stack**
  - **•E.g., process control blocks**
  - ❖ **Must be protected from user processes**
- **Can give kernel its own address space**
- **Problem: switching address space is costly**
- **Solution: map kernel address space into process address space**

Process Management

Linux process address space



Process Management

```
include/linux/sched.h
384    struct task_struct {
385      volatile long state;
386      struct thread_info *thread_info;
387      atomic_t usage;
388      unsigned long flags;
389      unsigned long ptrace;
390
391      int lock_depth;
392
393      int prio, static_prio;
394      struct list_head run_list;
395      prio_array_t *array;
396
397      unsigned long sleep_avg;
398      long interactive_credit;
399      unsigned long long timestamp;
400      int activated;
401
302      unsigned long policy;
403      cpumask_t cpus_allowed;
404      unsigned int time_slice, first_time_slice;
405
406      struct list_head tasks;
```

Process Management

- **State**
  - **TASK_RUNNING:** The process is runnable; it is either currently running or on a runqueue waiting to run .This is the only possible state for a process executing in user-space; it can also apply to a process in kernel-space that is actively running.
  - **TASK_INTERRUPTIBLE:** The process is sleeping (that is, it is blocked), waiting for some condition to exist.When this condition exists, the kernel sets the process's state to TASK_RUNNING.The process also awakes prematurely and becomes runnable if it receives a signal.

Process Management

- **TASK_UNINTERRUPTIBLE**: This state is identical to TASK_INTERRUPTIBLE except that it does not wake up and become runnable if it receives a signal. This is used in situations where the process must wait without interruption or when the event is expected to occur quite quickly. Because the task does not respond to signals in this state, TASK_UNINTERRUPTIBLE is less often used than TASK_INTERRUPTIBLE.
- **TASK_ZOMBIE**: The task has terminated, but its parent has not yet issued a wait4() system call.The task's process descriptor must remain in case the parent wants to access it. If the parent calls wait4(), the process descriptor is deallocated.

Process Management

- **TASK_STOPPED:** Process execution has stopped; the task is not running nor is it eligible to run.This occurs if the task receives the SIGSTOP, SIGTSTP, SIGTTIN, orSIGTTOU signal or if it receives any signal while it is being debugged.

Process Management

- **Manipulating the Current Process State**
  - Kernel code often needs to change a process’s state. The preferred mechanism is using set_task_state(task, state); /* set task 'task' to state 'state' */

Process Management

- **Exit_state**

  how a process exited

  ❖**EXIT_ZOMBIE:** the process is exiting but has not yet been waited for by its parent

  ❖ **EXIT_DEAD:** the process has exited and has been waited for

Process Management

- **Pid**
  - pid of first thread in process getpid() returns this ID, so all threads in a process share the same process ID

    Linux kernel uses pidhash to efficiently find processes by pids see include/linux/pid.h, kernel/pid.c

Process Management

- **Processes are related: children, sibling Parent/child**
- **Process groups: signal_struct->pgrp**
  - ❖ Possible to send signals to all members
- **Sessions: signal_struct->session**
  - ❖ Processes related to login

Process Management

- **Processes are related: children, sibling Parent/child**
- **Process groups: signal_struct->pgrp**
  - ❖ Possible to send signals to all members
- **Sessions: signal_struct->session**
  - ❖ Processes related to login

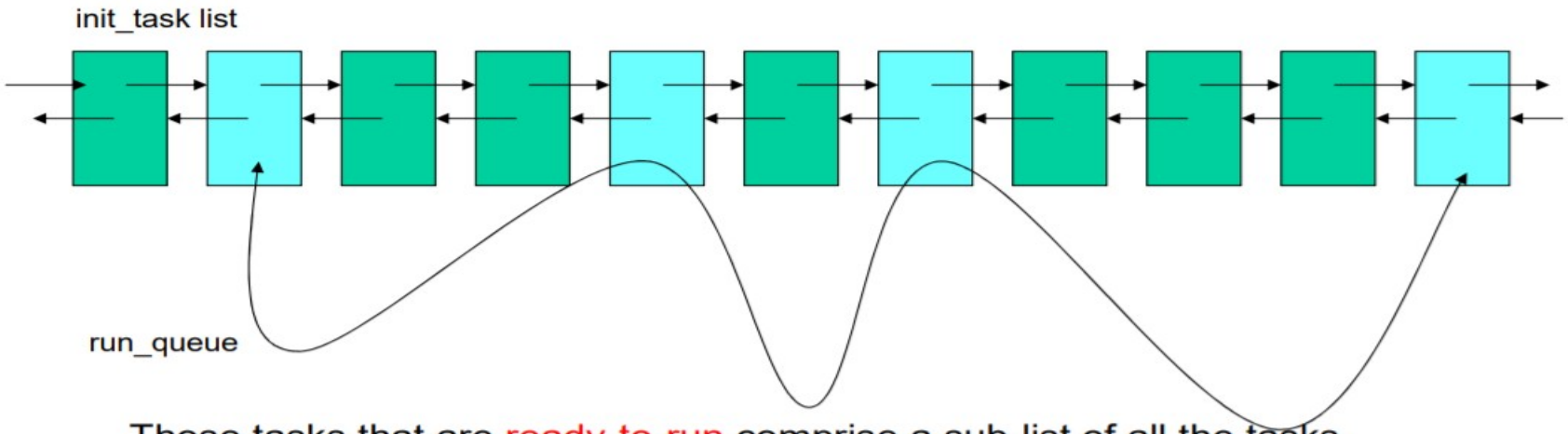
Process Management

- **Linux uses multiple queues to manage processes**
- Queue for all tasks
- Queue for “running” tasks
- Queues for tasks that temporarily are “blocked ” while waiting for a particular event to occur

- **These queues are implemented using doublylinked list (struct list_head in include/linux/list.h)**

Process Management

Some tasks are 'ready-to-run'



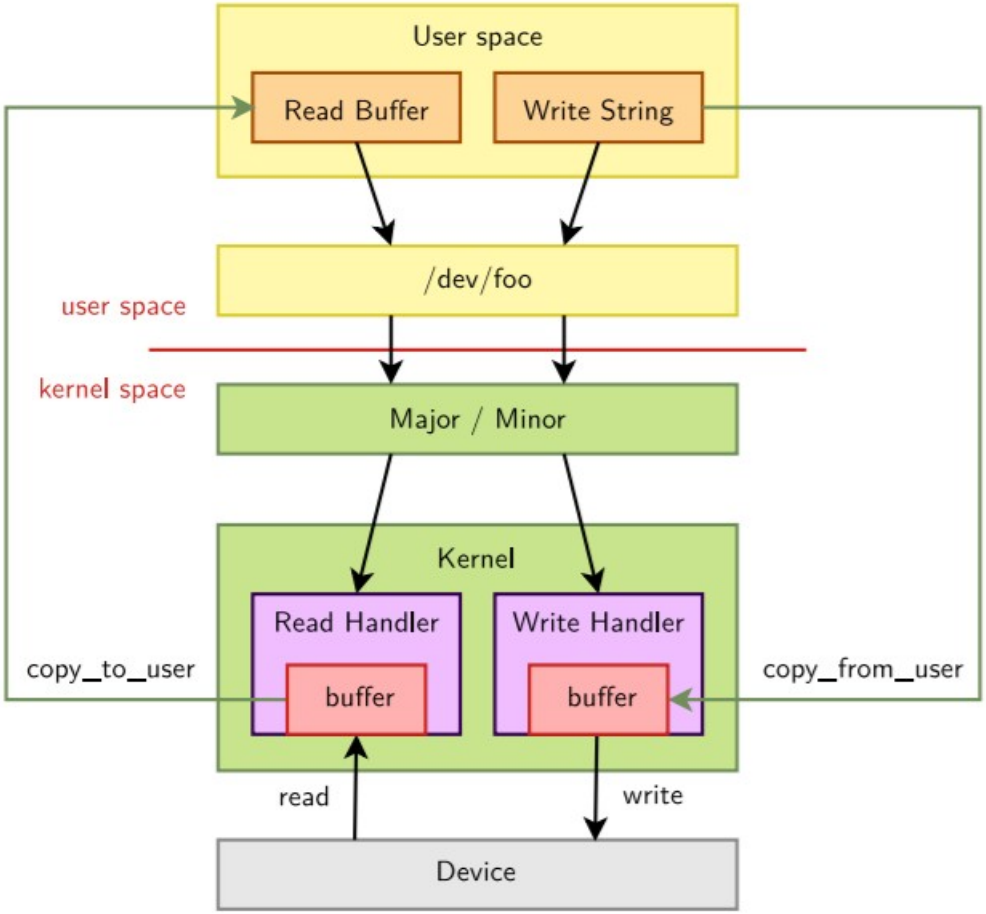
Those tasks that are ready-to-run comprise a sub-list of all the tasks, and they are arranged on a queue known as the 'run-queue' (struct *runqueue* in kernel/sched.c)

Those tasks that are blocked while awaiting a specific event to occur are put on alternative sub-lists, called 'wait queues', associated with the particular event(s) that will allow a blocked task to be unblocked (*wait_queue_t* in include/linux/wait.h and kernel/wait.c)

Character Driver

- From the point of view of an application, a *character device* is essentially a **file**.
- The driver of a character device must therefore implement **operations** that let applications think the device is a file: `open`, `close`, `read`, `write`, etc.
- In order to achieve this, a character driver must implement the operations described in the `struct file_operations` structure and register them.
- The Linux filesystem layer will ensure that the driver's operations are called when a user space application makes the corresponding system call.

Character Driver



Character Driver

Here are the most important operations for a character driver, from the definition of `struct file_operations`:

```
struct file_operations {
    struct module *owner;
    ssize_t (*read) (struct file *, char __user *,
        size_t, loff_t *);
    ssize_t (*write) (struct file *, const char __user *,
        size_t, loff_t *);
    long (*unlocked_ioctl) (struct file *, unsigned int,
        unsigned long);
    int (*mmap) (struct file *, struct vm_area_struct *);
    int (*open) (struct inode *, struct file *);
    int (*release) (struct inode *, struct file *);
    ...
};
```

Many more operations exist. All of them are optional.

Character Driver

- `int foo_open(struct inode *i, struct file *f)`
  - Called when user space opens the device file.
  - **Only implement this function when you do something special with the device at `open()` time.**
  - `struct inode` is a structure that uniquely represents a file in the filesystem (be it a regular file, a directory, a symbolic link, a character or block device)
  - `struct file` is a structure created every time a file is opened. Several file structures can point to the same `inode` structure.
    - Contains information like the current position, the opening mode, etc.
    - Has a `void *private_data` pointer that one can freely use.
    - A pointer to the `file` structure is passed to all other operations
- `int foo_release(struct inode *i, struct file *f)`
  - Called when user space closes the file.
  - **Only implement this function when you do something special with the device at `close()` time.**

Character Driver

- `ssize_t foo_read(struct file *f, char __user *buf, size_t sz, loff_t *off)`
  - Called when user space uses the `read()` system call on the device.
  - Must read data from the device, write at most `sz` bytes to the user space buffer `buf`, and update the current position in the file `off`. `f` is a pointer to the same file structure that was passed in the `open()` operation
  - Must return the number of bytes read.
    `0` is usually interpreted by userspace as the end of the file.
  - On UNIX, `read()` operations typically block when there isn't enough data to read from the device
- `ssize_t foo_write(struct file *f, const char __user *buf, size_t sz, loff_t *off)`
  - Called when user space uses the `write()` system call on the device
  - The opposite of `read`, must read at most `sz` bytes from `buf`, write it to the device, update `off` and return the number of bytes written.

Character Driver

- Kernel code isn't allowed to directly access user space memory, using `memcpy()` or direct pointer dereferencing
  - Doing so does not work on some architectures
  - If the address passed by the application was invalid, the application would segfault.
  - **Never** trust user space. A malicious application could pass a kernel address which you could overwrite with device data (`read` case), or which you could dump to the device (`write` case).
- To keep the kernel code portable, secure, and have proper error handling, your driver must use special kernel functions to exchange data with user space.

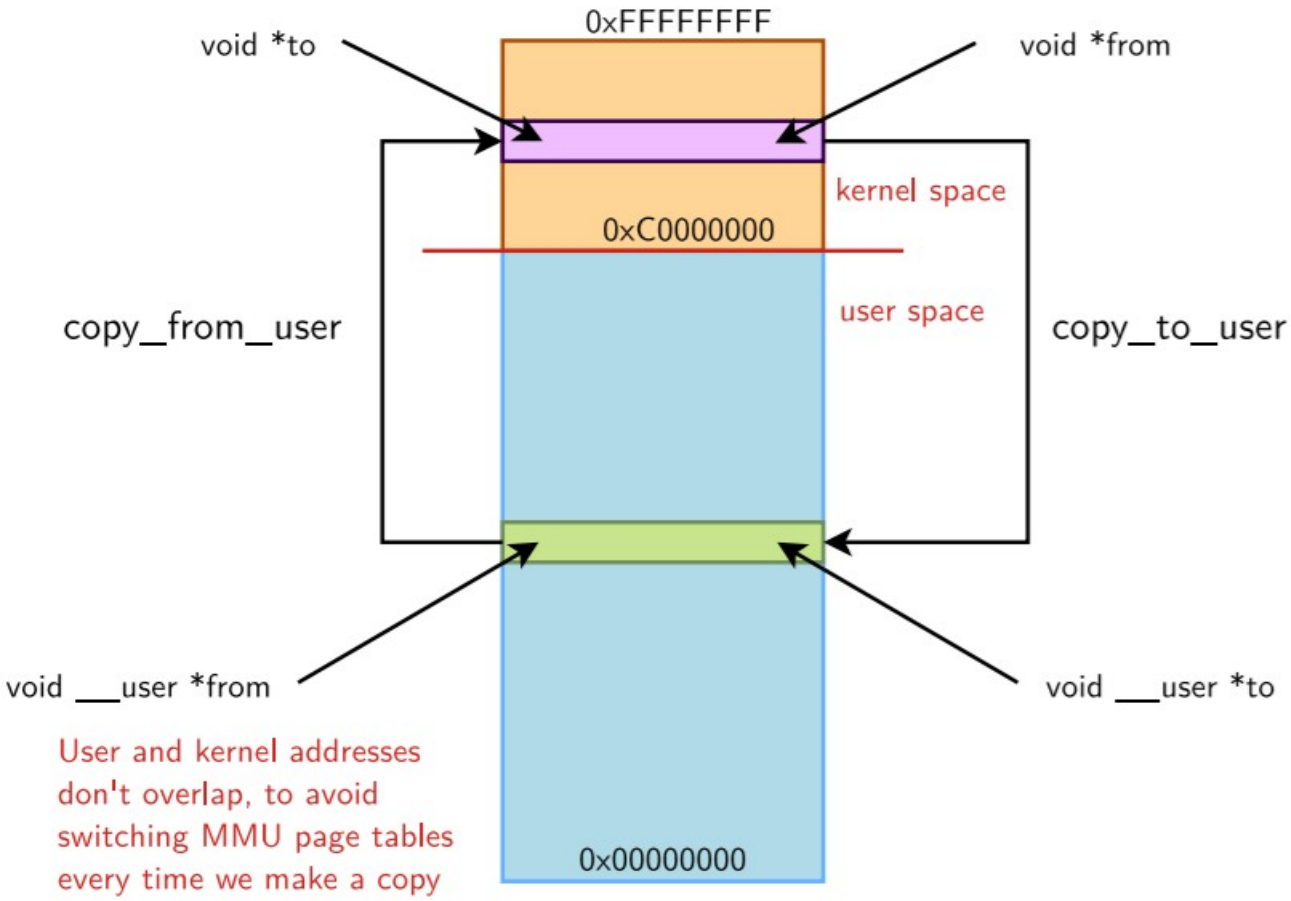
Character Driver

- A single value
  - `get_user(v, p);`
    - The kernel variable `v` gets the value pointed by the user space pointer `p`
  - `put_user(v, p);`
    - The value pointed by the user space pointer `p` is set to the contents of the kernel variable `v`.
- A buffer
  - `unsigned long copy_to_user(void __user *to,`

    `const void *from, unsigned long n);`
  - `unsigned long copy_from_user(void *to,`

    `const void __user *from, unsigned long n);`
- The return value must be checked. Zero on success, non-zero on failure. If non-zero, the convention is to return `-EFAULT`.

Character Driver



Character Driver

- Having to copy data to or from an intermediate kernel buffer can become expensive when the amount of data to transfer is large (video).
- *Zero copy* options are possible:
  - `mmap()` system call to allow user space to directly access memory mapped I/O space. See our `mmap()` chapter.
  - `get_user_pages()` and related functions to get a mapping to user pages without having to copy them.

Character Driver

- `long unlocked_ioctl(struct file *f, unsigned int cmd, unsigned long arg)`
  - Associated to the `ioctl()` system call.
  - Called unlocked because it didn't hold the Big Kernel Lock (gone now).
  - Allows to extend the driver capabilities beyond the limited read/write API.
  - For example: changing the speed of a serial port, setting video output format, querying a device serial number... Used extensively in the V4L2 (video) and ALSA (sound) driver frameworks.
  - `cmd` is a number identifying the operation to perform.
    See `driver-api/ioctl` for the recommended way of choosing `cmd` numbers.
  - `arg` is the optional argument passed as third argument of the `ioctl()` system call.
    Can be an integer, an address, etc.
  - The semantic of `cmd` and `arg` is driver-specific.

Character Driver

```
static long phantom_ioctl(struct file *file, unsigned int cmd,
    unsigned long arg)
{
    struct phm_reg r;
    void __user *argp = (void __user *)arg;

    switch (cmd) {
    case PHN_SET_REG:
        if (copy_from_user(&r, argp, sizeof(r)))
            return -EFAULT;
        /* Do something */
        break;
    ...
    case PHN_GET_REG:
        if (copy_to_user(argp, &r, sizeof(r)))
            return -EFAULT;
        /* Do something */
        break;
    ...
    default:
        return -ENOTTY;
    }

    return 0;
}
```

Character Driver

```c
int main(void)
{
    int fd, ret;
    struct phm_reg reg;

    fd = open("/dev/phantom");
    assert(fd > 0);

    reg.field1 = 42;
    reg.field2 = 67;

    ret = ioctl(fd, PHN_SET_REG, &reg);
    assert(ret == 0);

    return 0;
}
```

Character Driver

char1.c

char2.c

Character Driver

try_module_get

```
int try_module_get(struct module *module);
```

- **Features:** This function is used to increase the count module
- **parameter:**struct module structure pointer
- **return value:**If the return is 0, indicating that the call fails, hope to use the module is not being loaded or unloading

module_put

```
void module_put(struct module *module);
```

- **Features:** This function is used to reduce the count module
- **parameter:**struct module structure pointer

Platform Driver

Platform drivers

Platform Driver

- On embedded systems, devices are often not connected through a bus allowing enumeration, hotplugging, and providing unique identifiers for devices.
- For example, the devices on I2C buses or SPI buses, or the devices directly part of the system-on-chip.
- However, we still want all of these devices to be part of the device model.
- Such devices, instead of being dynamically detected, must be statically described in either:
  - The kernel source code
  - The *Device Tree*, a hardware description file used on some architectures.
  - BIOS ACPI tables (x86/PC architecture)

Platform Driver

- Amongst the non-discoverable devices, a huge family are the devices that are directly part of a system-on-chip: UART controllers, Ethernet controllers, SPI or I2C controllers, graphic or audio devices, etc.
- In the Linux kernel, a special bus, called the **platform bus** has been created to handle such devices.
- It supports **platform drivers** that handle **platform devices**.
- It works like any other bus (USB, PCI), except that devices are enumerated statically instead of being discovered dynamically.

Platform Driver

The driver implements a `struct platform_driver` structure (example taken from `drivers/tty/serial/imx.c`, simplified)

```
static struct platform_driver serial_imx_driver = {
        .probe          = serial_imx_probe,
        .remove         = serial_imx_remove,
        .id_table       = imx_uart_devtype,
        .driver         = {
                .name   = "imx-uart",
                .of_match_table = imx_uart_dt_ids,
                .pm     = &imx_serial_port_pm_ops,
        },
};
```

Platform Driver

... and registers its driver to the platform driver infrastructure

```
static int __init imx_serial_init(void) {
    ret = platform_driver_register(&serial_imx_driver);
}

static void __exit imx_serial_cleanup(void) {
    platform_driver_unregister(&serial_imx_driver);
}

module_init(imx_serial_init);
module_exit(imx_serial_cleanup);
```

Most drivers actually use the `module_platform_driver()` macro when they do nothing special in `init()` and `exit()` functions:

```
module_platform_driver(serial_imx_driver);
```

Platform Driver

- As platform devices cannot be detected dynamically, they are defined statically
  - By direct instantiation of `struct platform_device` structures, as done on a few old ARM platforms. Definition done in the board-specific or SoC specific code.
  - By using a *device tree*, as done on Power PC (and on most ARM platforms) from which `struct platform_device` structures are created
- Example on ARM, where the instantiation was done in `arch/arm/mach-imx/mx1ads.c`

```
static struct platform_device imx_uart1_device = {
    .name = "imx-uart",
    .id = 0,
    .num_resources = ARRAY_SIZE(imx_uart1_resources),
    .resource = imx_uart1_resources,
    .dev = {
        .platform_data = &uart_pdata,
    }
};
```

Platform Driver

- The device was part of a list

```
static struct platform_device *devices[] __initdata = {
    &cs89x0_device,
    &imx_uart1_device,
    &imx_uart2_device,
};
```

- And the list of devices was added to the system during board initialization

```
static void __init mx1ads_init(void)
{
    [...]
    platform_add_devices(devices, ARRAY_SIZE(devices));
}

MACHINE_START(MX1ADS, "Freescale MX1ADS")
    [...]
    .init_machine = mx1ads_init,
MACHINE_END
```

Platform Driver

- Each device managed by a particular driver typically uses different hardware resources: addresses for the I/O registers, DMA channels, IRQ lines, etc.
- Such information can be represented using `struct resource`, and an array of `struct resource` is associated to a `struct platform_device`
- Allows a driver to be instantiated for multiple devices functioning similarly, but with different addresses, IRQs, etc.

Platform Driver

```
static struct resource imx_uart1_resources[] = {
    [0] = {
        .start = 0x00206000,
        .end = 0x002060FF,
        .flags = IORESOURCE_MEM,
    },
    [1] = {
        .start = (UART1_MINT_RX),
        .end = (UART1_MINT_RX),
        .flags = IORESOURCE_IRQ,
    },
};
```

Platform Driver

- When a `struct platform_device` was added to the system using `platform_add_devices()`, the `probe()` method of the platform driver was called
- This method is responsible for initializing the hardware, registering the device to the proper framework (in our case, the serial driver framework)
- The platform driver has access to the I/O resources:

```
res = platform_get_resource(pdev, IORESOURCE_MEM, 0);
base = ioremap(res->start, PAGE_SIZE);
sport->rxirq = platform_get_irq(pdev, 0);
```

Platform Driver

- In addition to the well-defined resources, many drivers require driver-specific information for each platform device
- Such information could be passed using the `platform_data` field of `struct device` (from which `struct platform_device` inherits)
- As it is a `void *` pointer, it could be used to pass any type of information.
  - Typically, each driver defines a structure to pass information through `struct platform_data`

Platform Driver

- The i.MX serial port driver defines the following structure to be passed through `struct platform_data`

```
struct imxuart_platform_data {
    int (*init)(struct platform_device *pdev);
    void (*exit)(struct platform_device *pdev);
    unsigned int flags;
    void (*irda_enable)(int enable);
    unsigned int irda_inv_rx:1;
    unsigned int irda_inv_tx:1;
    unsigned short transceiver_delay;
};
```

- The MX1ADS board code instantiated such a structure

```
static struct imxuart_platform_data uart_pdata = {
    .flags = IMXUART_HAVE_RTSCTS,
};
```

Platform Driver

- The uart_pdata structure was associated to the struct platform_device structure in the MX1ADS board file (the real code was slightly more complicated)

```
struct platform_device mx1ads_uart1 = {
    .name = "imx-uart",
    .dev {
         .platform_data = &uart_pdata,
    },
    .resource = imx_uart1_resources,
    [...]
};
```

- The driver can access the platform data:

```
static int serial_imx_probe(struct platform_device *pdev)
{
    struct imxuart_platform_data *pdata;
    pdata = pdev->dev.platform_data;
    if (pdata && (pdata->flags & IMXUART_HAVE_RTSCTS))
        sport->have_rtscts = 1;
    [...]
```

Platform Driver

- On many embedded architectures, manual instantiation of platform devices was considered to be too verbose and not easily maintainable.
- Such architectures have moved, to use the *Device Tree*.
- It is a **tree of nodes** that models the hierarchy of devices in the system, from the devices inside the processor to the devices on the board.
- Each node can have a number of **properties** describing various properties of the devices: addresses, interrupts, clocks, etc.
- At boot time, the kernel is given a compiled version, the **Device Tree Blob**, which is parsed to instantiate all the devices described in the DT.
- On ARM, they are located in `arch/arm/boot/dts/`.

Platform Driver

```
uart0: serial@44e09000 {
        compatible = "ti,omap3-uart";
        ti,hwmods = "uart1";
        clock-frequency = <48000000>;
        reg = <0x44e09000 0x2000>;
        interrupts = <72>;
        status = "disabled";
};
```

- `serial@44e09000` is the **node name**
- `uart0` is a **label**, that can be referred to in other parts of the DT as `&uart0`
- other lines are **properties**. Their values are usually strings, list of integers, or references to other nodes.

Platform Driver

- Each particular hardware platform has its own *device tree*.
- However, several hardware platforms use the same processor, and often various processors in the same family share a number of similarities.
- To allow this, a *device tree* file can include another one. The trees described by the including file overlays the tree described by the included file. This can be done:
  - Either by using the `/include/` statement provided by the Device Tree language.
  - Either by using the `#include` statement, which requires calling the C preprocessor before parsing the Device Tree.

  Linux currently uses either one technique or the other (different from one ARM subarchitecture to another, for example).

Platform Driver

```
/ {
    compatible = "ti,am33xx";
    [...]
    ocp {
        [...]
        uart0: serial@0 {
            compatible = "ti,am3352-uart", "ti,omap3-uart";
            reg = <0x0 0x1000>;
            interrupts = <72>;
            status = "disabled";
            [...]
        };
    };
};
```

am33xx-l4.dtsi (included by am33xx.dtsi)

Definition of the AM33xx SoC

+

```
[...]
&uart0 {
        pinctrl-names = "default";
        pinctrl-0 = <&uart0_pins>;
        status = "okay";
};
```

am335x-bone-common.dtsi

Common definitions for BeagleBone boards

+

```
#include "am33xx.dtsi"
#include "am335x-bone-common.dtsi"
#include "am335x-boneblack-common.dtsi"

/ {
        model = "TI AM335x BeagleBone Black";
        compatible = "ti,am335x-bone-black",
                     "ti,am335x-bone",
                     "ti,am33xx";
};

[...]
```

am335x-boneblack.dts

Definition for BeagleBone Black

=

Compiled DTB

```
/ {
    compatible = "ti,am335x-bone-black", "ti,am335x-bone", "ti,am33xx";
    model = "TI AM335x BeagleBone Black";
    [...]
    ocp {
        [...]
        uart0: serial@0 {
            compatible = "ti,am3352-uart", "ti,omap3-uart";
            reg = <0x0 0x1000>;
            interrupts = <72>;
            pinctrl-names = "default";
            pinctrl-0 = <&uart0_pins>;
            status = "okay";
        };
    };
};
```

am335x-boneblack.dtb

Note: the real DTB is in binary format. Here we show the text equivalent of the DTB contents;

Platform Driver

- With the *device tree*, a *device* is bound to the corresponding *driver* using the **compatible** string.
- The `of_match_table` field of `struct device_driver` lists the compatible strings supported by the driver. `drivers/tty/serial/omap-serial.c` example:

```
#if defined(CONFIG_OF)
static const struct of_device_id omap_serial_of_match[] = {
        { .compatible = "ti,omap2-uart" },
        { .compatible = "ti,omap3-uart" },
        { .compatible = "ti,omap4-uart" },
        {},
};
MODULE_DEVICE_TABLE(of, omap_serial_of_match);
#endif
static struct platform_driver serial_omap_driver = {
        .probe          = serial_omap_probe,
        .remove         = serial_omap_remove,
        .driver         = {
                .name   = DRIVER_NAME,
                .pm     = &serial_omap_dev_pm_ops,
                .of_match_table = of_match_ptr(omap_serial_of_match),
        },
};
```

- Note: the `of_match_ptr()` macro instantiates to `NULL` when `CONFIG_OF` is not set.

Platform Driver

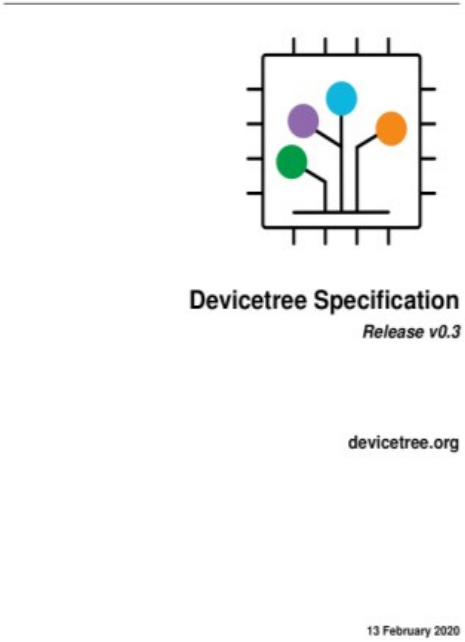
- The drivers will use the same mechanism that we saw previously to retrieve basic information: interrupts numbers, physical addresses, etc.
- The available resources list will be built up by the kernel at boot time from the device tree, so that you don't need to make any unnecessary lookups to the DT when loading your driver.
- Any additional information will be specific to a driver or the class it belongs to, defining the *bindings*.

Platform Driver

- **Describe hardware** (how the hardware is), not configuration (how I choose to use the hardware)
- **OS-agnostic**
  - For a given piece of HW, Device Tree should be the same for U-Boot, FreeBSD or Linux
  - There should be no need to change the Device Tree when updating the OS
- Describe **integration of hardware components**, not the internals of hardware components
  - The details of how a specific device/IP block is working is handled by code in device drivers
  - The Device Tree describes how the device/IP block is connected/integrated with the rest of the system: IRQ lines, DMA channels, clocks, reset lines, etc.
- Like all beautiful design principles, these principles are sometimes violated.

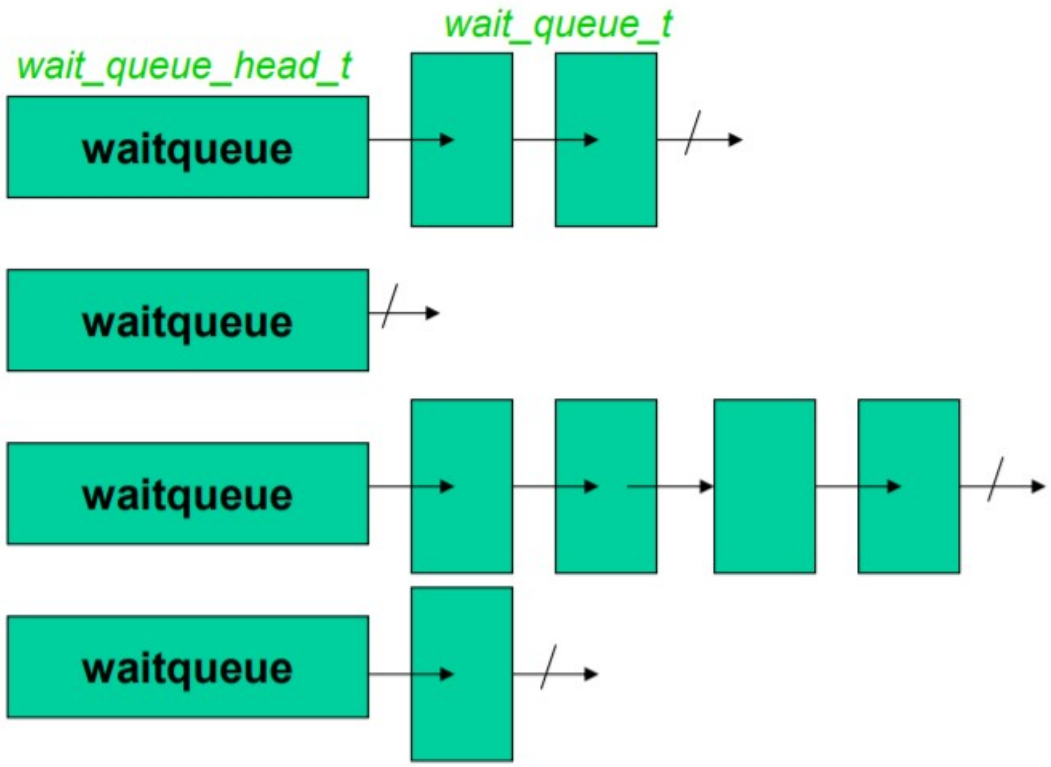
Platform Driver

- How to write the correct nodes/properties to describe a given hardware platform ?
- **DeviceTree Specifications** → base Device Tree syntax + number of standard properties.
  - https://www.devicetree.org/specifications/
  - Not sufficient to describe the wide variety of hardware.
- **Device Tree Bindings** → documents that each specify how a piece of HW should be described
  - Documentation/devicetree/bindings/ in Linux kernel sources
  - Reviewed by DT bindings maintainer team
  - Legacy: human readable documents
  - New norm: YAML-written specifications



Platform Driver Handson

Process Management



wait_queue_head_t can have 0 or more *wait_queue_t* chained onto them

However, usually just one element

Each *wait_queue_t* contains a *list_head* of tasks

All processes waiting for specific "event“

Used for timing, synch, device i/o, etc.

Process Management

fork() call chain

- libc fork()
- system_call (arch/i386/kernel/entry.S)
- sys_clone() (arch/i386/kernel/process.c)
- do_fork() (kernel/fork.c)
- copy_process() (kernel/fork.c)
- p = dup_task_struct(current) // shallow copy
- copy_* // copy point-to structures
- copy_thread () // copy stack, regs, and eip
- wake_up_new_task() // set child runnable

Process Management

exit() call chain

- libc exit(code)
- system_call (arch/i386/kernel/entry.S)
- sys_exit() (kernel/exit.c)
- do_exit() (kernel/exit.c)
- exit_*() // free data structures
- exit_notify() // tell other processes we exit
  // reparent children to init
  // EXIT_ZOMBIE
  // EXIT_DEAD

Process Management

Context switch call chain

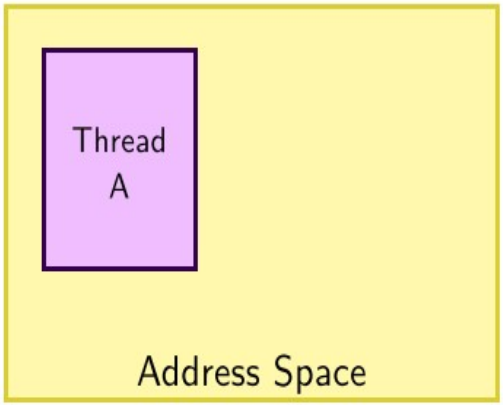
- schedule() (kernel/sched.c) (talk about scheduling later)
- context_switch()
- swtich_mm (include/asm-i386/mmu_context.h)
  // switch address space
- switch_to (include/asm-i386/system.h)
- __swtich_to (arch/i386/kernel/process.c)
  // switch stack to switch CPU context

Process Management

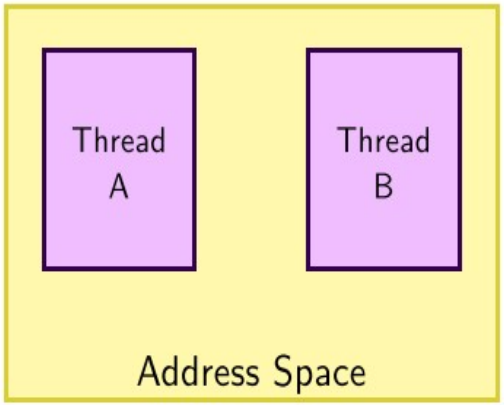
- Confusion about the terms *process*, *thread* and *task*
- In UNIX, a process is created using `fork()` and is composed of
  - An address space, which contains the program code, data, stack, shared libraries, etc.
  - A single thread, which is the only entity known by the scheduler.
- Additional threads can be created inside an existing process, using `pthread_create()`
  - They run in the same address space as the initial thread of the process
  - They start executing a function passed as argument to `pthread_create()`

Process Management

- In kernel space, each thread running in the system is represented by a structure of type `struct task_struct`
- From a scheduling point of view, it makes no difference between the initial thread of a process and all additional threads created dynamically using `pthread_create()`



Process after fork()

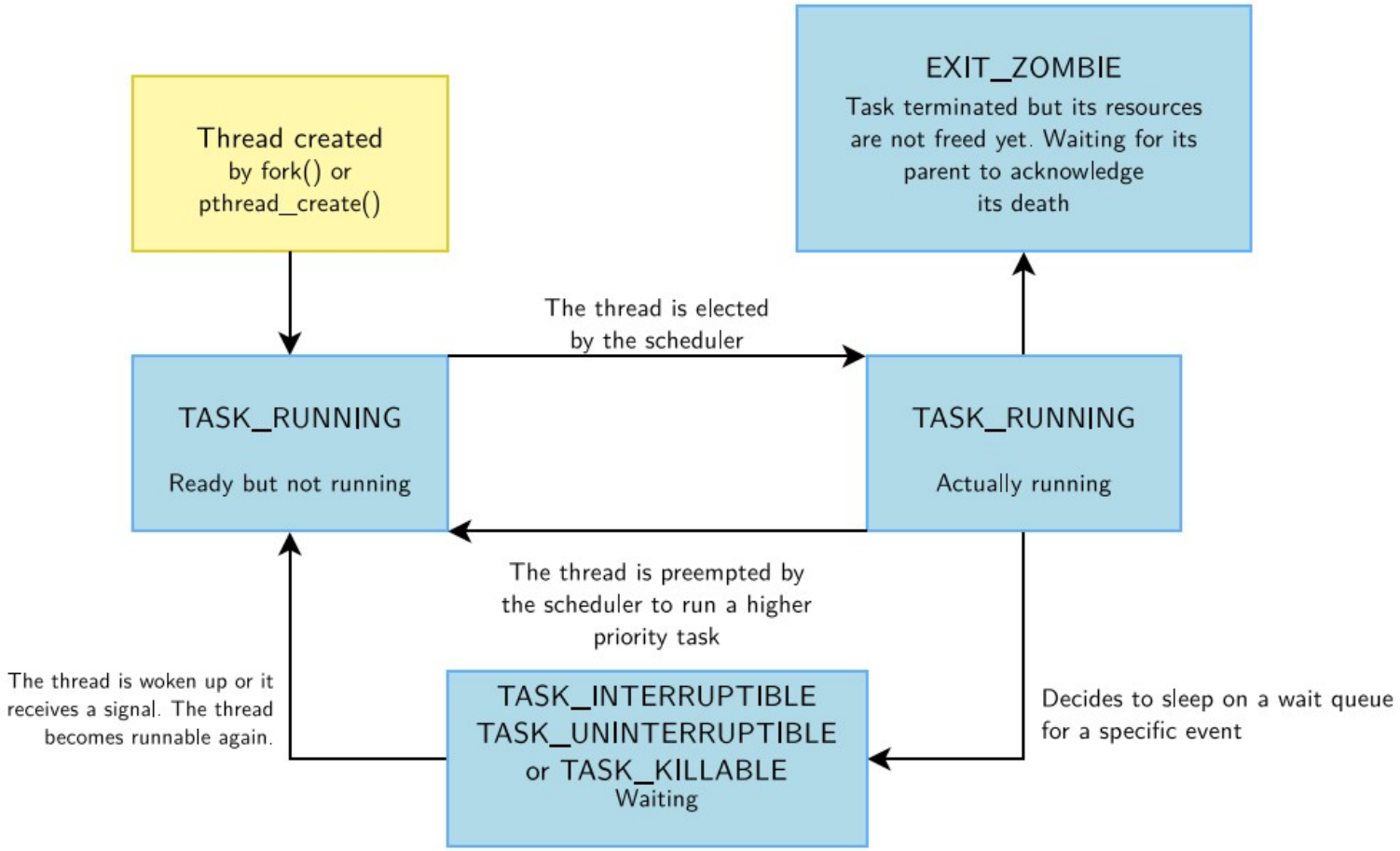


Same process after pthread_create()

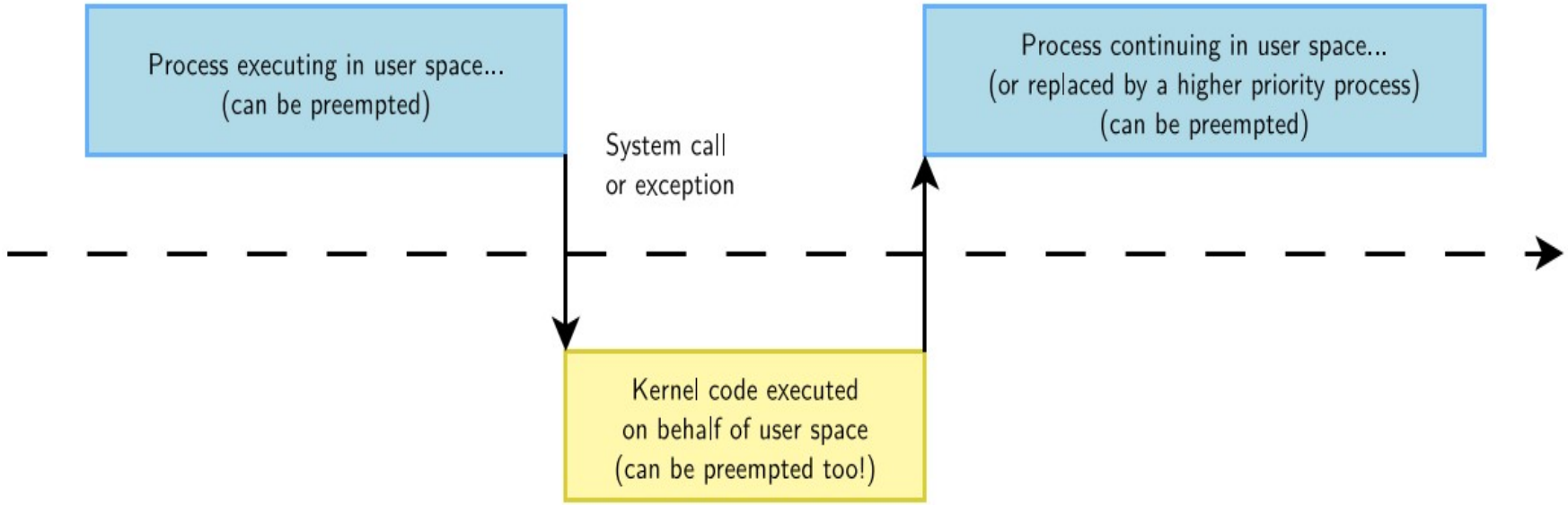
Process Management

- When speaking about *process* and *thread*, these concepts need to be clarified:
  - *Mode* is the level of privilege allowing to perform some operations:
    - *Kernel Mode*: in this level CPU can perform any operation allowed by its architecture; any instruction, any I/O operation, any area of memory accessed.
    - *User Mode*: in this level, certain instructions are not permitted (especially those that could alter the global state of the machine), some memory areas cannot be accessed.
  - Linux splits its *address space* in *kernel space* and *user space*
    - *Kernel space* is reserved for code running in *Kernel Mode*.
    - *User space* is the place were applications execute (accessible from *Kernel Mode*).
  - *Context* represents the current state of an execution flow.
    - The *process context* can be seen as the content of the registers associated to this process: execution register, stack register...
    - The *interrupt context* replaces the *process context* when the interrupt handler is executed.

Process Management



Process Management



The execution of system calls takes place in the context of the thread requesting them.

Process Management

`fork` is the heavy-weight call because it creates a full copy of the parent process that then executes as a child process. To reduce the effort associated with this call, Linux uses the *copy-on-write* technique, discussed below.

`vfork` is similar to fork but does not create a copy of the data of the parent process. Instead, it shares the data between the parent and child process. This saves a great deal of CPU time (and if one of the processes were to manipulate the shared data, the other would notice automatically).

`vfork` is designed for the situation in which a child process just generated immediately executes an `execve` system call to load a new program. The kernel also guarantees that the parent process is blocked until the child process exits or starts a new program.

Quoting the manual page `vfork(2)`, it is "rather unfortunate that Linux revived this specter from the past." Since `fork` uses copy-on-write, the speed argument for `vfork` does not really count anymore, and its use should therefore be avoided.

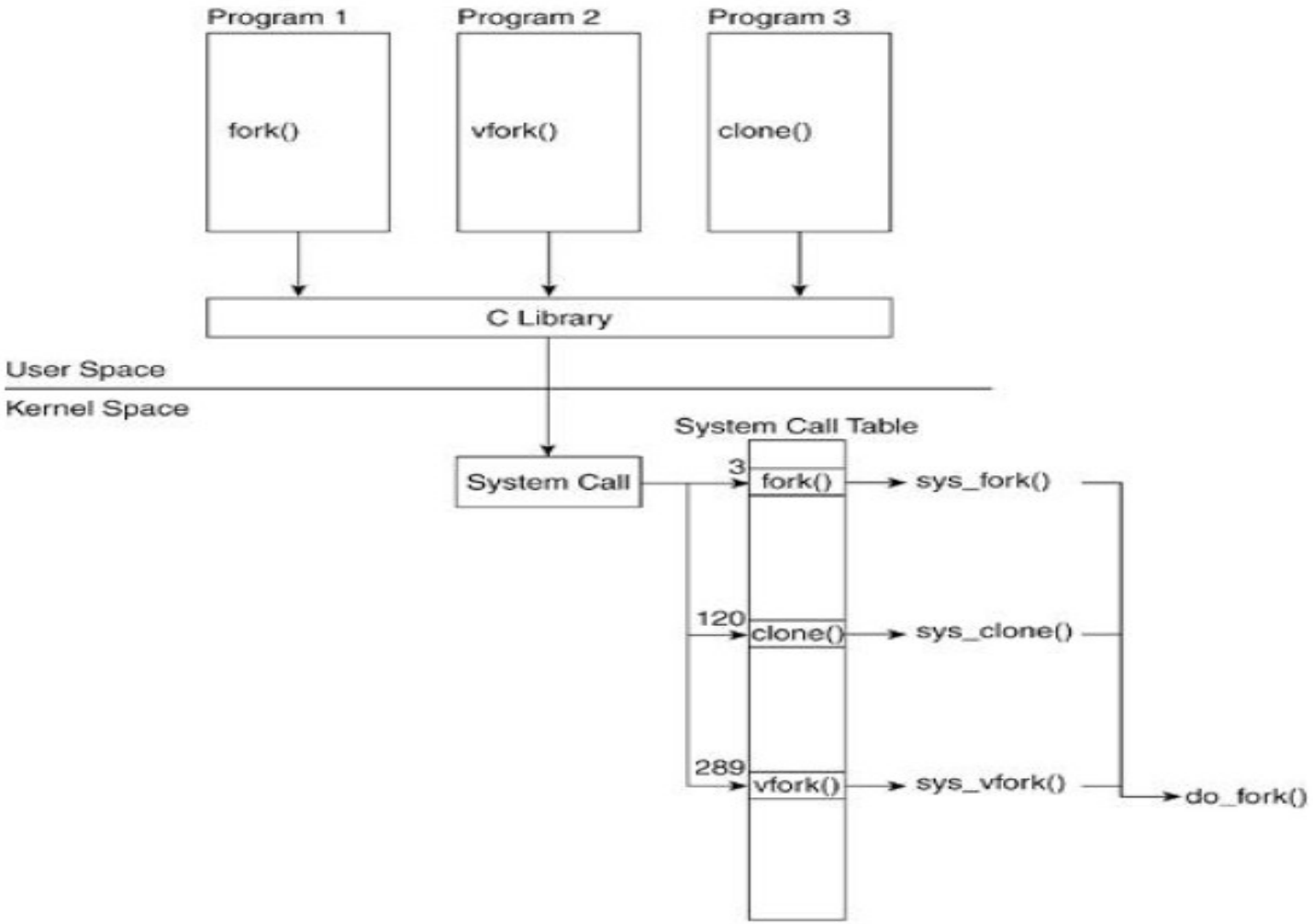
`clone` generates threads and enables a decision to be made as to exactly which elements are to be shared between the parent and the child process and which are to be copied.

Process Management

The entry points for the `fork`, `vfork`, and `clone` system calls are the `sys_fork`, `sys_vfork`, and `sys_clone` functions. Their definitions are architecture-dependent because the way in which parameters are passed between userspace and kernel space differs on the various architectures (see Chapter 13 for further information). The task of the above functions is to extract the information supplied by userspace from the registers of the processors and then to invoke the architecture-*in*dependent `do_fork` function responsible for process duplication. The prototype of the function is as follows.

kernel/fork.c

```
long do_fork(unsigned long clone_flags,
             unsigned long stack_start,
             struct pt_regs *regs,
             unsigned long stack_size,
             int __user *parent_tidptr,
             int __user *child_tidptr)
```


Program 1
Program 2
Program 3
fork()
vfork()
clone()
C Library
User Space
Kernel Space
System Call Table
System Call
3
fork()
sys_fork()
120
clone()
sys_clone()
289
vfork()
sys_vfork()
do_fork()

Process Management

arch/x86/kernel/process_32.c

```
asmlinkage int sys_fork(struct pt_regs regs)
{
        return do_fork(SIGCHLD, regs.esp, &regs, 0, NULL, NULL);
}
```

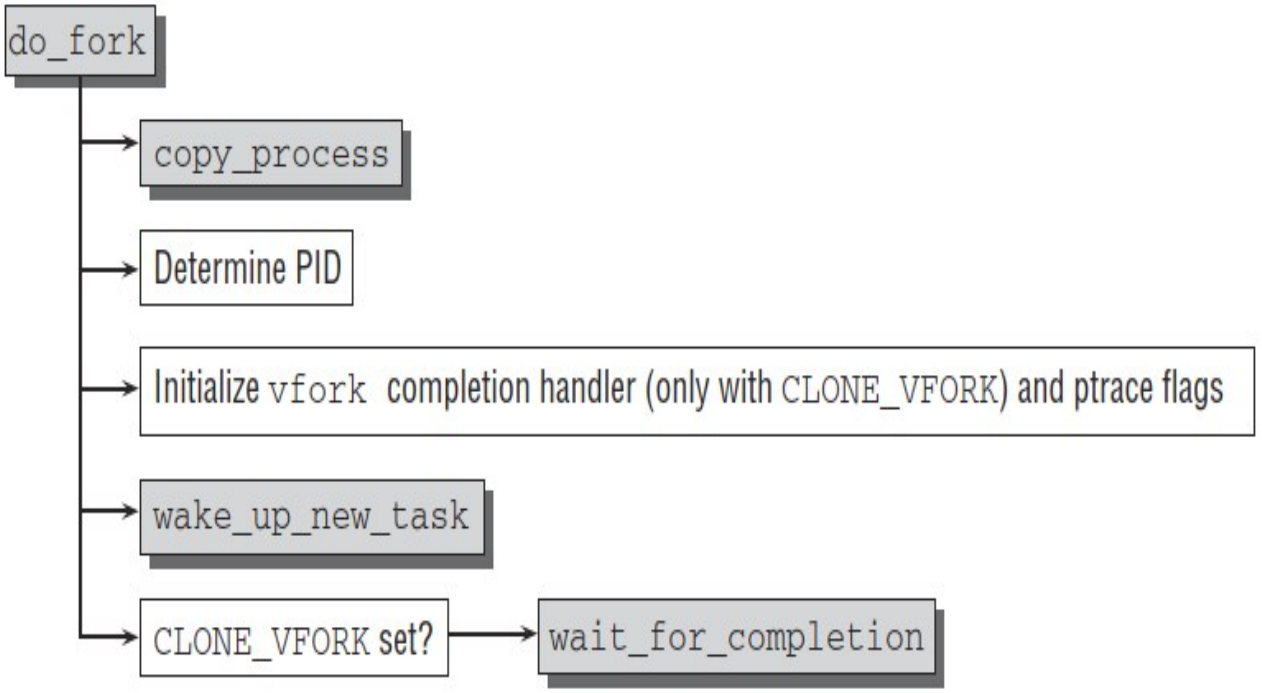
The only flag used is `SIGCHLD`. This means that the `SIGCHLD` signal informs the parent process once the child process has terminated. Initially, the same stack (whose start address is held in the `esp` register on IA-32 systems) is used for the parent and child processes. However, the COW mechanism creates a copy of the stack for each process if it is manipulated and therefore written to.

If `do_fork` was successful, the PID of the newly created task is returned as the result of the system call. Otherwise the (negative) error code is returned.

The implementation of `sys_vfork` differs only slightly from that of `sys_fork` in that additional flags are used (`CLONE_VFORK` and `CLONE_VM` whose meaning is discussed below).

`sys_clone` is also implemented in a similar way to the above calls with the difference that `do_fork` is invoked as follows:

Process Management



Process Management

sys_clone is also implemented in a similar way to the above calls with the difference that do_fork is invoked as follows:

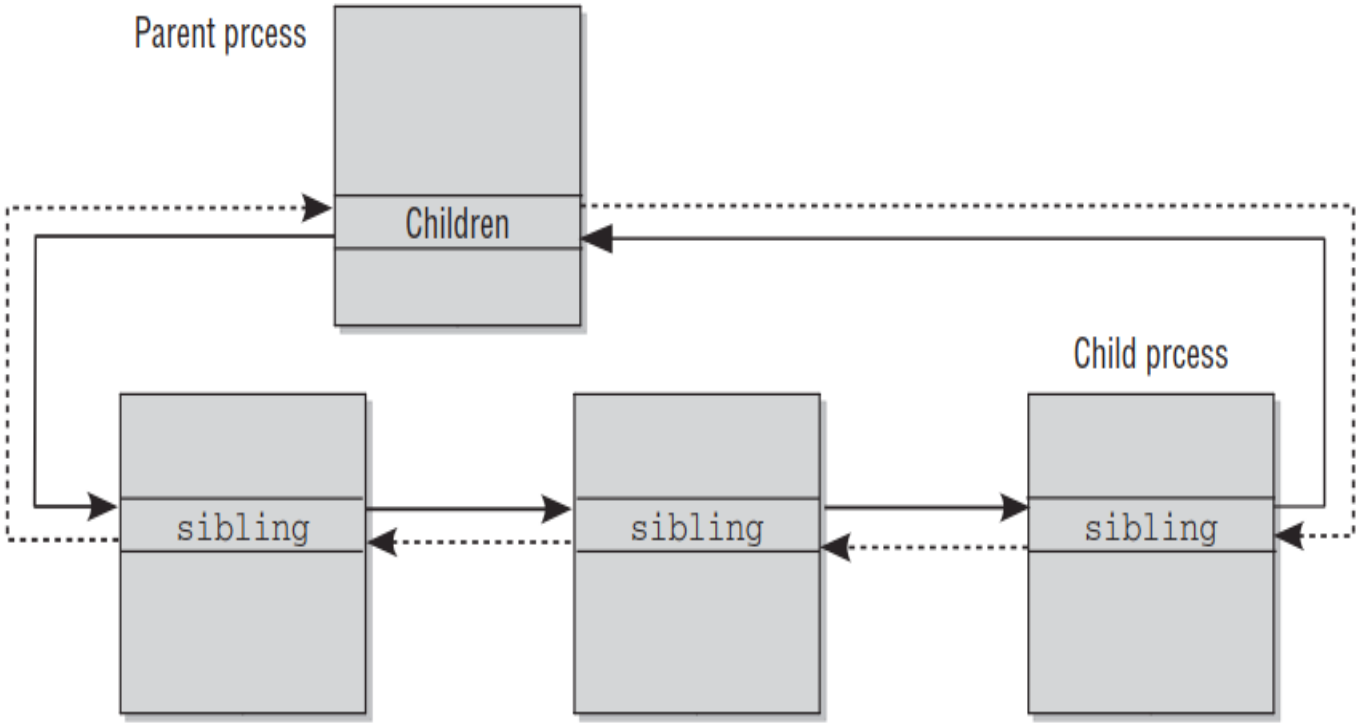
arch/x86/kernel/process_32.c

```
asmlinkage int sys_clone(struct pt_regs regs)
{
        unsigned long clone_flags;
        unsigned long newsp;
        int __user *parent_tidptr, *child_tidptr;

        clone_flags = regs.ebx;
        newsp = regs.ecx;
        parent_tidptr = (int __user *)regs.edx;
        child_tidptr = (int __user *)regs.edi;
        if (!newsp)
                newsp = regs.esp;
        return do_fork(clone_flags, newsp, &regs, 0, parent_tidptr, child_tidptr);
}
```

The clone flags are no longer permanently set but can be passed to the system call as parameters in various registers. Thus, the first part of the function deals with extracting these parameters. Also, the stack of the parent process is not copied; instead, a new address (newsp) can be specified for it. (This is required to generate threads that share the address space with the parent process but use their own stack in this address space.) Two pointers (parent_tidptr and child_tidptr) in userspace are also specified for purposes of communication with thread libraries. Their meaning is discussed in Section 2.4.1.

Process Management



Misc Driver

```
#include <linux/module.h>
#include <linux/init.h>
#include <linux/fs.h>
#include <linux/slab.h>
#include <linux/miscdevice.h>
#include <linux/kernel.h>
#include <linux/uaccess.h>
#include <linux/mm.h>

MODULE_AUTHOR("Abha Patidar");
MODULE_DESCRIPTION("RD WR");
MODULE_LICENSE("GPL");
MODULE_VERSION("1.1.1");

#define MAXBYTES 10
```

Misc Driver

```
static char *kernel_buffer;

static int openFunc(struct inode *openNode, struct file *openFile)
{
    char *openBuff = kzalloc(PATH_MAX, GFP_KERNEL);
    printk(KERN_INFO "Opening device file %s \n", file_path(openFile,
openBuff, PATH_MAX));
    kfree(openBuff);

    return nonseekable_open(openNode, openFile);
}
```

Misc Driver

```
static ssize_t readFunc(struct file *readFile, char __user *readBuffer,
size_t readCount, loff_t *readOffset)
{
    printk(KERN_INFO "Inside reader \n");

    printk(KERN_INFO "Request to read %ld bytes from driver to transmit
to user space \n", readCount);

    if (copy_to_user(readBuffer, kernel_buffer, readCount))
    {
        printk(KERN_INFO "Issue with copying data to user buffer \n");
    }
    *readOffset += readCount;
    return readCount;
}
```

Misc Driver

```
static ssize_t writeFunc(struct file *writeFIle, const char __user
*writeBuffer, size_t writeCount, loff_t *writeOffset)
{
    printk(KERN_INFO "Inside writer = %ld \n", writeCount);

    copy_from_user(kernel_buffer, writeBuffer, writeCount);
    kernel_buffer[writeCount] = '\0';
    return writeCount;
}
```

Misc Driver

```
static int releaseFunc(struct inode *releaseNode, struct file
*releaseFile)
{
    char *file_buff = kzalloc(PATH_MAX, GFP_KERNEL);

    printk(KERN_INFO "Closing the driver file %s \n",
file_path(releaseFile, file_buff, PATH_MAX));

    return 0;
}

static const struct file_operations misc_driver_fops =
{
    .open    = openFunc,
    .read    = readFunc,
    .write   = writeFunc,
    .release = releaseFunc,
    .llseek  = no_llseek,
};
```

Misc Driver

```
static struct miscdevice misc_driver =
{
    .name  = "test",
    .mode  = 0666,
    .minor = MISC_DYNAMIC_MINOR,
    .fops  = &misc_driver_fops,
};
```

Misc Driver

```
static int __init startUp(void)
{
    int retcode;
   int m;
    printk(KERN_INFO "Registering module \n");
    retcode = misc_register(&misc_driver);
    if (retcode)
    {
        printk(KERN_ERR "Error in creating misc driver \n");
        return retcode;
    }
    kernel_buffer = kvmalloc(100, GFP_KERNEL);
    m = strlcpy(kernel_buffer, "initmsg", 8);
    printk(KERN_INFO "Misc device registered with minor = %d ; m = %d ,
dev node = %s \n", misc_driver.minor,m, misc_driver.name);
    return 0;
}
```

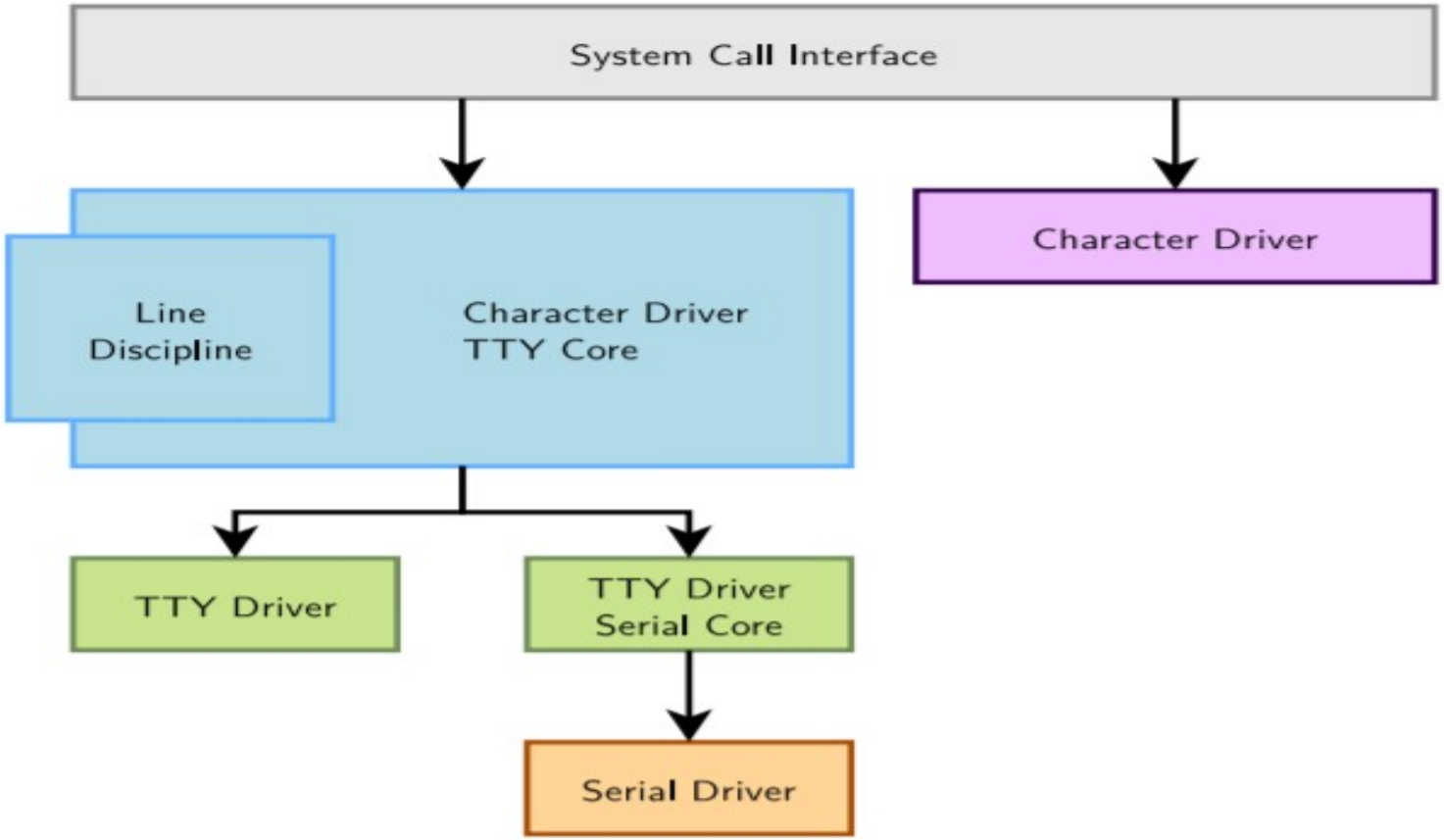
Misc Driver

```
static void __exit terminate(void)
{
    printk(KERN_INFO "Removing module \n");
    misc_deregister(&misc_driver);
}

module_init(startUp);
module_exit(terminate);
```

Module_param_cb

UART Driver


System Call Interface
Line Discipline
Character Driver TTY Core
Character Driver
TTY Driver
TTY Driver Serial Core
Serial Driver

UART Driver

- To be properly integrated in a Linux system, serial ports must be visible as TTY devices from user space applications
- Therefore, the serial driver must be part of the kernel TTY subsystem
- Until 2.6, serial drivers were implemented directly behind the TTY core
  - A lot of complexity was involved
- Since 2.6, a specialized TTY driver, `serial_core`, eases the development of serial drivers
  - See `include/linux/serial_core.h` for the main definitions of the `serial_core` infrastructure
- The line discipline that cooks the data exchanged with the `tty` driver. For normal serial ports, `N_TTY` is used.

UART Driver

- A data structure representing a driver: `uart_driver`
  - Single instance for each driver
  - `uart_register_driver()` and `uart_unregister_driver()`
- A data structure representing a port: `uart_port`
  - One instance for each port (several per driver are possible)
  - `uart_add_one_port()` and `uart_remove_one_port()`
- A data structure containing the pointers to the operations: `uart_ops`
  - Linked from `uart_port` through the `ops` field

UART Driver

- Usually
  - Defined statically in the driver
  - Registered in `module_init()`
  - Unregistered in `module_cleanup()`
- Contains
  - `owner`, usually set to `THIS_MODULE`
  - `driver_name`
  - `dev_name`, the device name prefix, usually `ttyS`
  - `major` and `minor`
    - Use `TTY_MAJOR` and `64` to get the normal numbers. But they might conflict with the 8250-reserved numbers
  - `nr`, the maximum number of ports
  - `cons`, pointer to the console device (covered later)

UART Driver

```
static struct uart_driver atmel_uart = {
    .owner = THIS_MODULE,
    .driver_name = "atmel_serial",
    .dev_name = ATMEL_DEVICENAME,
    .major = SERIAL_ATMEL_MAJOR,
    .minor = MINOR_START,
    .nr = ATMEL_MAX_UART,
    .cons = ATMEL_CONSOLE_DEVICE,
};

static struct platform_driver atmel_serial_driver = {
    .probe = atmel_serial_probe,
    .remove = atmel_serial_remove,
    .suspend = atmel_serial_suspend,
    .resume = atmel_serial_resume,
    .driver = {
        .name = "atmel_usart",
        .owner = THIS_MODULE,
    },
};
```

Example code from drivers/serial/atmel_serial.c

UART Driver

```
static int __init atmel_serial_init(void)
{
    /* Warning: Error management removed */
    uart_register_driver(&atmel_uart);
    platform_driver_register(&atmel_serial_driver);
    return 0;
}

static void __exit atmel_serial_exit(void)
{
    platform_driver_unregister(&atmel_serial_driver);
    uart_unregister_driver(&atmel_uart);
}

module_init(atmel_serial_init);
module_exit(atmel_serial_exit);
```

UART Driver

- Can be allocated statically or dynamically
- Usually registered at `probe()` time and unregistered at `remove()` time
- Most important fields
  - `iotype`, type of I/O access, usually `UPIO_MEM` for memory-mapped devices
  - `mapbase`, physical address of the registers
  - `irq`, the IRQ channel number
  - `membase`, the virtual address of the registers
  - `uartclk`, the clock rate
  - `ops`, pointer to the operations
  - `dev`, pointer to the device (`platform_device` or other)

UART Driver

```
static int atmel_serial_probe(struct platform_device *pdev)
{
    struct atmel_uart_port *port;

    port = &atmel_ports[pdev->id];
    port->backup_imr = 0;

    atmel_init_port(port, pdev);

    uart_add_one_port(&atmel_uart, &port->uart);

    platform_set_drvdata(pdev, port);

    return 0;
}

static int atmel_serial_remove(struct platform_device *pdev)
{
    struct uart_port *port = platform_get_drvdata(pdev);

    platform_set_drvdata(pdev, NULL);
    uart_remove_one_port(&atmel_uart, port);

    return 0;
}
```

UART Driver

```
static void atmel_init_port(
    struct atmel_uart_port *atmel_port,
    struct platform_device *pdev)
{
    struct uart_port *port = &atmelt_port->uart;
    struct atmel_uart_data *data = pdev->dev.platform_data;

    port->iotype = UPIO_MEM;
    port->flags = UPF_BOOT_AUTOCONF;
    port->ops = &atmel_pops;
    port->fifosize = 1;
    port->line = pdev->id;
    port->dev = &pdev->dev;

    port->mapbase = pdev->resource[0].start;
    port->irq = pdev->resource[1].start;

    tasklet_init(&atmel_port->tasklet, atmel_tasklet_func,
        (unsigned long)port);
```

UART Driver

```
    if (data->regs)
        /* Already mapped by setup code */
        port->membase = data->regs;
    else {
        port->flags |= UPF_IOREMAP;
        port->membase = NULL;
    }

    /* for console, the clock could already be configured */
    if (!atmel_port->clk) {
        atmel_port->clk = clk_get(&pdev->dev, "usart");
        clk_enable(atmel_port->clk);
        port->uartclk = clk_get_rate(atmel_port->clk);
        clk_disable(atmel_port->clk);
        /* only enable clock when USART is in use */
    }
}
```

UART Driver

- Important operations
  - `tx_empty()`, tells whether the transmission FIFO is empty or not
  - `set_mctrl()` and `get_mctrl()`, allow to set and get the modem control parameters (RTS, DTR, LOOP, etc.)
  - `start_tx()` and `stop_tx()`, to start and stop the transmission
  - `stop_rx()`, to stop the reception
  - `startup()` and `shutdown()`, called when the port is opened/closed
  - `request_port()` and `release_port()`, request/release I/O or memory regions
  - `set_termios()`, change port parameters
- See the detailed description in `Documentation/serial/driver`

UART Driver

- The `start_tx()` method should start transmitting characters over the serial port
- The characters to transmit are stored in a circular buffer, implemented by a `struct uart_circ` structure. It contains
  - `buf[]`, the buffer of characters
  - `tail`, the index of the next character to transmit. After transmit, tail must be updated using

    ```
    tail = tail &(UART_XMIT_SIZE - 1)
    ```
- Utility functions on `uart_circ`
  - `uart_circ_empty()`, tells whether the circular buffer is empty
  - `uart_circ_chars_pending()`, returns the number of characters left to transmit
- From an `uart_port` pointer, this structure can be reached using `port->state->xmit`

UART Driver

```
foo_uart_putc(struct uart_port *port, unsigned char c) {
    while(__raw_readl(port->membase + UART_REG1) & UART_TX_FULL)
        cpu_relax();
    __raw_writel(c, port->membase + UART_REG2);
}

foo_uart_start_tx(struct uart_port *port) {
    struct circ_buf *xmit = &port->state->xmit;

    while (!uart_circ_empty(xmit)) {
        foo_uart_putc(port, xmit->buf[xmit->tail]);
        xmit->tail = (xmit->tail + 1) & (UART_XMIT_SIZE - 1);
        port->icount.tx++;
    }
}
```

UART Driver

```
foo_uart_interrupt(int irq, void *dev_id) {
    [...]
    if (interrupt_cause & END_OF_TRANSMISSION)
        foo_uart_handle_transmit(port);
    [...]
}

foo_uart_start_tx(struct uart_port *port) {
    enable_interrupt_on_txrdy();
}
```

UART Driver

```
foo_uart_handle_transmit(port) {
    struct circ_buf *xmit = &port->state->xmit;
    if (uart_circ_empty(xmit) || uart_tx_stopped(port)) {
        disable_interrupt_on_txrdy();
        return;
    }

    while (!uart_circ_empty(xmit)) {
        if (!(__raw_readl(port->membase + UART_REG1) &
            UART_TX_FULL))
            break;
        __raw_writel(xmit->buf[xmit->tail],
            port->membase + UART_REG2);
        xmit->tail = (xmit->tail + 1) & (UART_XMIT_SIZE - 1);
        port->icount.tx++;
    }

    if (uart_circ_chars_pending(xmit) < WAKEUP_CHARS)
        uart_write_wakeup(port);
```

UART Driver

- On reception, usually in an interrupt handler, the driver must
  - Increment `port->icount.rx`
  - Call `uart_handle_break()` if a `BRK` has been received, and if it returns TRUE, skip to the next character
  - If an error occurred, increment `port->icount.parity`, `port->icount.frame`, `port->icount.overrun` depending on the error type
  - Call `uart_handle_sysrq_char()` with the received character, and if it returns TRUE, skip to the next character
  - Call `uart_insert_char()` with the received character and a status
    - Status is `TTY_NORMAL` is everything is OK, or `TTY_BREAK`, `TTY_PARITY`, `TTY_FRAME` in case of error
  - Call `tty_flip_buffer_push()` to push data to the TTY layer

UART Driver

- Part of the reception work is dedicated to handling `Sysrq`
  - Sysrq are special commands that can be sent to the kernel to make it reboot, unmount filesystems, dump the task state, nice real-time tasks, etc.
  - These commands are implemented at the lowest possible level so that even if the system is locked, you can recover it.
  - Through serial port: send a `BRK` character, send the character of the `Sysrq` command
  - See `Documentation/sysrq.txt`
- In the driver
  - `uart_handle_break()` saves the current time + 5 seconds in a variable
  - `uart_handle_sysrq_char()` will test if the current time is below the saved time, and if so, will trigger the execution of the Sysrq command

UART Driver

```
foo_receive_chars(struct uart_port *port) {
    int limit = 256;

    while (limit-- > 0) {
        status = __raw_readl(port->membase + REG_STATUS);
        ch = __raw_readl(port->membase + REG_DATA);
        flag = TTY_NORMAL;

        if (status & BREAK) {
            port->icount.break++;
            if (uart_handle_break(port))
                continue;
        }
        else if (status & PARITY)
            port->icount.parity++;
        else if (status & FRAME)
            port->icount.frame++;
        else if (status & OVERRUN)
            port->icount.overrun++;

        [...]
```

UART Driver

```
        [...]
        status &= port->read_status_mask;

        if (status & BREAK)
            flag = TTY_BREAK;
        else if (status & PARITY)
            flag = TTY_PARITY;
        else if (status & FRAME)
            flag = TTY_FRAME;

        if (uart_handle_sysrq_char(port, ch))
            continue;

        uart_insert_char(port, status, OVERRUN, ch, flag);
    }

    spin_unlock(& port->lock);
    tty_flip_buffer_push(port->state->port.tty);
    spin_lock(& port->lock);
}
```

UART Driver

- Set using the `set_mctrl()` operation
  - The `mctrl` argument can be a mask of `TIOCM_RTS` (request to send), `TIOCM_DTR` (Data Terminal Ready), `TIOCM_OUT1`, `TIOCM_OUT2`, `TIOCM_LOOP` (enable loop mode)
  - If a bit is set in `mctrl`, the signal must be driven active, if the bit is cleared, the signal must be driven inactive
- Status using the `get_mctrl()` operation
  - Must return read hardware status and return a combination of `TIOCM_CD` (Carrier Detect), `TIOCM_CTS` (Clear to Send), `TIOCM_DSR` (Data Set Ready) and `TIOCM_RI` (Ring Indicator)

UART Driver

```
foo_set_mctrl(struct uart_port *uart, u_int mctrl) {
    unsigned int control = 0, mode = 0;

    if (mctrl & TIOCM_RTS)
        control |= ATMEL_US_RTSEN;
    else
        control |= ATMEL_US_RTSDIS;

    if (mctrl & TIOCM_DTS)
        control |= ATMEL_US_DTREN;
    else
        control |= ATMEL_US_DTRDIS;

    __raw_writel(port->membase + REG_CTRL, control);

    if (mctrl & TIOCM_LOOP)
        mode |= ATMEL_US_CHMODE_LOC_LOOP;
    else
        mode |= ATMEL_US_CHMODE_NORMAL;

    __raw_writel(port->membase + REG_MODE, mode);
}
```

UART Driver

```
foo_get_mctrl(struct uart_port *uart, u_int mctrl) {
    unsigned int status, ret = 0;

    status = __raw_readl(port->membase + REG_STATUS);

    /*
     * The control signals are active low.
     */
     if (!(status & ATMEL_US_DCD))
         ret |= TIOCM_CD;
     if (!(status & ATMEL_US_CTS))
         ret |= TIOCM_CTS;
     if (!(status & ATMEL_US_DSR))
         ret |= TIOCM_DSR;
     if (!(status & ATMEL_US_RI))
         ret |= TIOCM_RI;

     return ret;
}
```

UART Driver

- *The termios functions describe a general terminal interface that is provided to control asynchronous communication ports*
- A mechanism to control from user space serial port parameters such as
  - Speed
  - Parity
  - Byte size
  - Stop bit
  - Hardware handshake
  - Etc.
- See `termios(3)` for details

UART Driver

- The `set_termios()` operation must
  - apply configuration changes according to the arguments
  - update `port->read_config_mask` and `port->ignore_config_mask` to indicate the events we are interested in receiving
-
```
static void set_termios(struct uart_port *port,
struct ktermios *termios, struct ktermios *old)
```
  - `port`, the port, `termios`, the new values and `old`, the old values
- Relevant `ktermios` structure fields are
  - `c_cflag` with word size, stop bits, parity, reception enable, CTS status change reporting, enable modem status change reporting
  - `c_iflag` with frame and parity errors reporting, break event reporting

UART Driver

```
static void atmel_set_termios(struct uart_port *port,
    struct ktermios *termios, struct ktermios *old)
{
    unsigned long flags;
    unsigned int mode, imr, quot, baud;

    mode = __raw_readl(port->membase + REG_MODE);
    baud = uart_get_baud_rate(port, termios, old, 0, port->uartclk / 16);
    /* Read current configuration */
    quot = uart_get_divisor(port, baud);

    /* Compute the mode modification for the byte size parameter */
    switch (termios->c_cflag & CSIZE) {
    case CS5:
        mode |= ATMEL_US_CHRL_5;
        break;
    case CS6:
        mode |= ATMEL_US_CHRL_6;
        break;
    [...]
    default:
        mode |= ATMEL_US_CHRL_8;
```

UART Driver

```
/* Compute the mode modification for the stop bit */
if (termios->c_cflag & CSTOPB)
    mode |= ATMEL_US_NBSTOP_2;

/* Compute the mode modification for parity */
if (termios->c_cflag & PARENB) {
    /* Mark or Space parity */
    if (termios->c_cflag & CMSPAR) {
        if (termios->c_cflag & PARODD)
            mode |= ATMEL_US_PAR_MARK;
        else
            mode |= ATMEL_US_PAR_SPACE;
    } else if (termios->c_cflag & PARODD)
        mode |= ATMEL_US_PAR_ODD;
    else
        mode |= ATMEL_US_PAR_EVEN;
} else
    mode |= ATMEL_US_PAR_NONE;

/* Compute the mode modification for CTS reporting */
if (termios->c_cflag & CRTSCTS)
    mode |= ATMEL_US_USMODE_HWHS;
```

UART Driver

```
/* Compute the read_status_mask and ignore_status_mask
 * according to the events we're interested in. These
 * values are used in the interrupt handler. */
port->read_status_mask = ATMEL_US_OVRE;
if (termios->c_iflag & INPCK)
    port->read_status_mask |= (ATMEL_US_FRAME | ATMEL_US_PARE);
if (termios->c_iflag & (BRKINT | PARMRK))
    port->read_status_mask |= ATMEL_US_RXBRK;

port->ignore_status_mask = 0;
if (termios->c_iflag & IGNPAR)
    port->ignore_status_mask |= (ATMEL_US_FRAME | ATMEL_US_PARE);
if (termios->c_iflag & IGNBRK) {
    port->ignore_status_mask |= ATMEL_US_RXBRK;
    if (termios->c_iflag & IGNPAR)
        port->ignore_status_mask |= ATMEL_US_OVRE;
}
/* The serial_core maintains a timeout that corresponds to the
 * duration it takes to send the full transmit FIFO. This timeout has
 * to be updated. */
uart_update_timeout(port, termios->c_cflag, baud);
```

UART Driver

```
    /* Finally, apply the mode and baud rate modifications. Interrupts,
     * transmission and reception are disabled when the modifications
     * are made. */

    /* Save and disable interrupts */
    imr = UART_GET_IMR(port);
    UART_PUT_IDR(port, -1);
    /* disable receiver and transmitter */
    UART_PUT_CR(port, ATMEL_US_TXDIS | ATMEL_US_RXDIS);
    /* set the parity, stop bits and data size */
    UART_PUT_MR(port, mode);
    /* set the baud rate */
    UART_PUT_BRGR(port, quot);
    UART_PUT_CR(port, ATMEL_US_RSTSTA | ATMEL_US_RSTRX);
    UART_PUT_CR(port, ATMEL_US_TXEN | ATMEL_US_RXEN);
    /* restore interrupts */
    UART_PUT_IER(port, imr);
    /* CTS flow-control and modem-status interrupts */
    if (UART_ENABLE_MS(port, termios->c_cflag))
        port->ops->enable_ms(port);
}
```

UART Driver

- To allows early boot messages to be printed, the kernel provides a separate but related facility: `console`
  - This console can be enabled using the `console=` kernel argument
- The driver developer must
  - Implement a `console_write()` operation, called to print characters on the console
  - Implement a `console_setup()` operation, called to parse the `console=` argument
  - Declare a `struct console` structure
  - Register the console using a `console_initcall()` function

UART Driver

```
static struct console serial_txx9_console = {
    .name = TXX9_TTY_NAME,
    .write = serial_txx9_console_write,
    /* Helper function from the serial_core layer */
    .device = uart_console_device,
    .setup = serial_txx9_console_setup,
    /* Ask for the kernel messages buffered during
     * boot to be printed to the console when activated */
    .flags = CON_PRINTBUFFER,
    .index = -1,
    .data = &serial_txx9_reg,
};

static int __init serial_txx9_console_init(void)
{
    register_console(&serial_txx9_console);
    return 0;
}

/* This will make sure the function is called early during the boot process.
 * start_kernel() calls console_init() that calls our function */
console_initcall(serial_txx9_console_init);
```

UART Driver

```
static int __init serial_txx9_console_setup(struct console *co,
    char *options)
{
    struct uart_port *port;
    struct uart_txx9_port *up;
    int baud = 9600;
    int bits = 8;
    int parity = 'n';
    int flow = 'n';

    if (co->index >= UART_NR)
        co->index = 0;
    up = &serial_txx9_ports[co->index];
    port = &up->port;
    if (!port->ops)
        return -ENODEV;

    /* Function shared with the normal serial driver */
    serial_txx9_initialize(&up->port);

    if (options)
        /* Helper function from serial_core that parses the console= string */
        uart_parse_options(options, &baud, &parity, &bits, &flow);

    /* Helper function from serial_core that calls the ->set_termios() */
    /* operation with the proper arguments to configure the port */
    return uart_set_options(port, co, baud, parity, bits, flow);
}
```

UART Driver

```
static void serial_txx9_console_putchar(struct uart_port *port, int ch)
{
^^Istruct uart_txx9_port *up = (struct uart_txx9_port *)port;
^^I/* Busy-wait for transmitter ready and output a single character. */
^^Iwait_for_xmitr(up);
^^Isio_out(up, TXX9_SITFIFO, ch);
}

static void serial_txx9_console_write(struct console *co,
    const char *s, unsigned int count)
{
    struct uart_txx9_port *up = &serial_txx9_ports[co->index];
    unsigned int ier, flcr;

    /* Disable interrupts */
    ier = sio_in(up, TXX9_SIDICR);
    sio_out(up, TXX9_SIDICR, 0);

    /* Disable flow control */
    flcr = sio_in(up, TXX9_SIFLCR);
    if (!(up->port.flags & UPF_CONS_FLOW) && (flcr & TXX9_SIFLCR_TES))
        sio_out(up, TXX9_SIFLCR, flcr & ~TXX9_SIFLCR_TES);

    /* Helper function from serial_core that repeatedly calls the given putchar() */
    /* callback */
    uart_console_write(&up->port, s, count, serial_txx9_console_putchar);

    /* Re-enable interrupts */
    wait_for_xmitr(up);
    sio_out(up, TXX9_SIFLCR, flcr);
```

QEMU Vexpress-a9

- Extract toolchain

  # tar -xvf gcc-linaro-10.2.1-2021.02-x86_64_arm-linux-gnueabihf.tar.xz

- Extract kernel, configure and build

  # tar -xvf gcc-linaro-10.2.1-2021.02-x86_64_arm-linux-gnueabihf.tar.xz

  # cd linux-5.1.16/

  # cp arch/arm/configs/vexpress_defconfig .config

  # make menuconfig ARCH=arm CROSS_COMPILE=/home/test/qemu_test3/gcc-linaro-10.2.1-2021.02-x86_64_arm-linux-gnueabihf/bin/arm-linux-gnueabihf-

QEMU Vexpress-a9

make ARCH=arm CROSS_COMPILE=/home/test/qemu_test3/gcc-linaro-10.2.1-2021.02-x86_64_arm-linux-gnueabihf/bin/arm-linux-gnueabihf-

- Setup Rootfs using Busybox

tar -xvjf busybox-1.33.1.tar.bz2

cd busybox-1.33.1/

make defconfig

make menuconfig ARCH=arm CROSS_COMPILE=/home/test/qemu_test3/gcc-linaro-10.2.1-2021.02-x86_64_arm-linux-gnueabihf/bin/arm-linux-gnueabihf- <Enable static linking>

QEMU Vexpress-a9

```
# make ARCH=arm CROSS_COMPILE=/home/test/qemu_test3/gcc-linaro-10.2.1-2021.02-x86_64_arm-linux-gnueabihf/bin/arm-linux-gnueabihf-
# make install ARCH=arm CROSS_COMPILE=/home/test/qemu_test3/gcc-linaro-10.2.1-2021.02-x86_64_arm-linux-gnueabihf/bin/arm-linux-gnueabihf-
# cd ..
# mkdir rootfs
# cd rootfs
```

QEMU Vexpress-a9

```
# sudo cp -rf ../busybox-1.33.1/_install/* .
# cd ..
# sudo tar xvzf rootfs_base.tar.gz
#  cd rootfs
# sudo cp -rf ../rootfs_base/* .
# find . | cpio -o --format=newc > ../rootfs.img
# cd ..
# gzip -c rootfs.img > rootfs.img.gz
# qemu-system-arm -M vexpress-a9 -m 512M -dtb linux-5.1.16/arch/arm/boot/dts/vexpress-v2p-ca9.dtb -kernel linux-5.1.16/arch/arm/boot/zImage -initrd rootfs.img.gz -append "root=/dev/ram  rdinit=/linuxrc"
```

QEMU Vexpress-a9

qemu-system-arm -M vexpress-a9 -m 512M -dtb linux-5.1.16/arch/arm/boot/dts/vexpress-v2p-ca9.dtb -nographic -kernel linux-5.1.16/arch/arm/boot/zImage -initrd rootfs.img.gz -append "root=/dev/ram console=ttyAMA0 rdinit=/linuxrc“ (Withou GUI)

QEMU Vexpress-a9 boot with hello world

```
#include <stdio.h>

int main()
{
	printf("Hello World\n");

	 while(1);

	return 0;
}
```

QEMU Vexpress-a9 boot with hello world

- Compile program with static option

  # /home/test/qemu_test3/gcc-linaro-10.2.1-2021.02-x86_64_arm-linux-gnueabihf/bin/arm-linux-gnueabihf-gcc -static -g -o helloworld helloworld.c

- Create initrd using helloworld

  # find helloworld | cpio -o --format=newc > rootfs_hello.img

  # gzip -c rootfs_hello.img > rootfs_helloworld.img.gz

  # qemu-system-arm -M vexpress-a9 -m 512M -dtb linux-5.1.16/arch/arm/boot/dts/vexpress-v2p-ca9.dtb -kernel linux-5.1.16/arch/arm/boot/zImage -initrd rootfs_helloworld.img.gz -append "root=/dev/ram rdinit=/helloworld"

QEMU Vexpress-a9 boot with hello world debugging

- Start qemu with debug options

  # qemu-system-arm -M vexpress-a9 -m 512M -S -s -dtb linux-5.1.16/arch/arm/boot/dts/vexpress-v2p-ca9.dtb -kernel linux-5.1.16/arch/arm/boot/zImage -initrd rootfs_helloworld.img.gz -append "root=/dev/ram rdinit=/helloworld"

- Setup gdb client to connect to this gdb server

  # /home/test/qemu/gcc-linaro-10.2.1-2021.02-x86_64_arm-linux-gnueabihf/bin/arm-linux-gnueabihf-gdb

  (gdb) target remote localhost:1234

  (gdb) file helloworld

  (gdb) break main

  (gdb) continue

QEMU Vexpress-a9 boot with kernel debugging

- Start qemu with debug options

  # qemu-system-arm -M vexpress-a9 -m 512M -S -s -dtb linux-5.1.16/arch/arm/boot/dts/vexpress-v2p-ca9.dtb -kernel linux-5.1.16/arch/arm/boot/zImage -initrd rootfs_helloworld.img.gz -append "root=/dev/ram rdinit=/helloworld"

- Setup gdb client to connect to this gdb server

  # /home/test/qemu/gcc-linaro-10.2.1-2021.02-x86_64_arm-linux-gnueabihf/bin/arm-linux-gnueabihf-gdb

  (gdb) target remote localhost:1234

  (gdb) file vmlinux

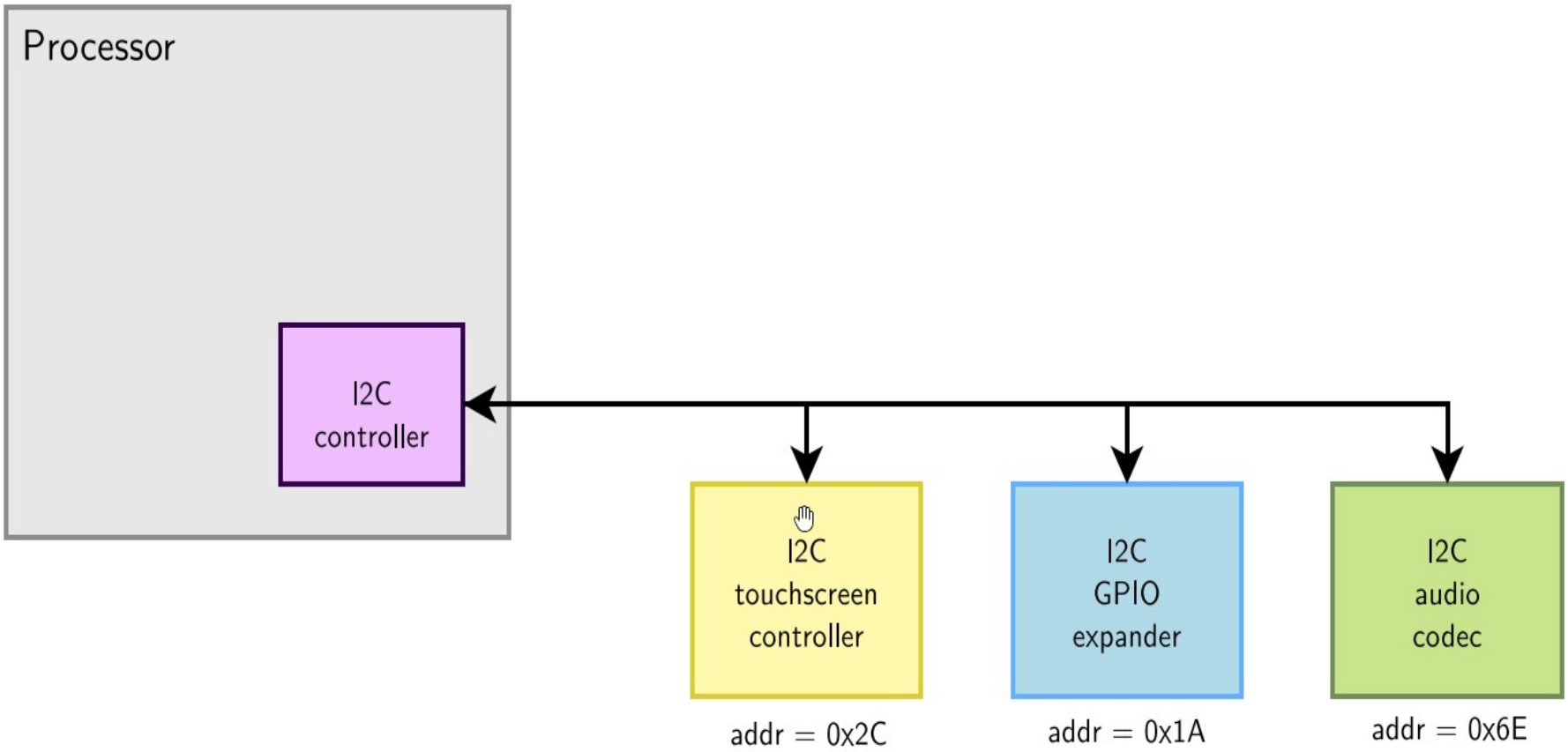
  (gdb) break start_kernel

  (gdb) continue

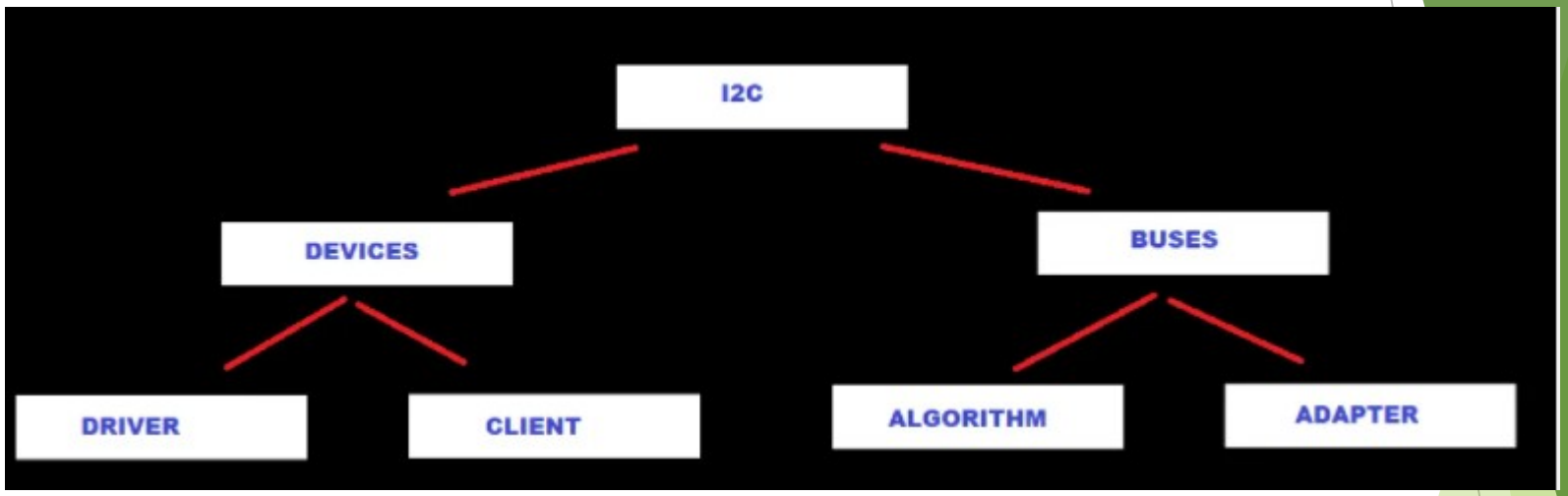
I2C Driver

- A very commonly used low-speed bus to connect on-board and external devices to the processor.
- Uses only two wires: SDA for the data, SCL for the clock.
- It is a master/slave bus: only the master can initiate transactions, and slaves can only reply to transactions initiated by masters.
- In a Linux system, the I2C controller embedded in the processor is typically the master, controlling the bus.
- Each slave device is identified by an I2C address (you can't have 2 devices with the same address on the same bus). Each transaction initiated by the master contains this address, which allows the relevant slave to recognize that it should reply to this particular transaction.

I2C Driver


Processor
I2C controller
I2C touchscreen controller
I2C GPIO expander
I2C audio codec
addr = 0x2C
addr = 0x1A
addr = 0x6E

I2C Driver



I2C Driver

- Like all bus subsystems, the I2C bus driver is responsible for:
  - Providing an API to implement I2C controller drivers
  - Providing an API to implement I2C device drivers, in kernel space
  - Providing an API to implement I2C device drivers, in user space
- The core of the I2C bus driver is located in `drivers/i2c/`.
- The I2C controller drivers are located in `drivers/i2c/busses/`.
- The I2C device drivers are located throughout `drivers/`, depending on the framework used to expose the devices (e.g. `drivers/input/` for input devices).

I2C Driver

- Like all bus subsystems, the I2C subsystem defines a `struct i2c_driver` that inherits from `struct device_driver`, and which must be instantiated and registered by each I2C device driver.
  - As usual, this structure points to the `->probe()` and `->remove()` functions.
  - It also contains an `id_table`, used for non-DT based probing of I2C devices.
  - A `->probe_new()` function can replace `->probe()` when no `id_table` is provided.
- The `i2c_add_driver()` and `i2c_del_driver()` functions are used to register/unregister the driver.
- If the driver doesn't do anything else in its `init()`/`exit()` functions, it is advised to use the `module_i2c_driver()` macro instead.

I2C Driver

```
static const struct i2c_device_id adxl345_i2c_id[] = {
        { "adxl345", ADXL345 },
        { "adxl375", ADXL375 },
        { }
};

MODULE_DEVICE_TABLE(i2c, adxl345_i2c_id);

static const struct of_device_id adxl345_of_match[] = {
        { .compatible = "adi,adxl345" },
        { .compatible = "adi,adxl375" },
        { },
};

MODULE_DEVICE_TABLE(of, adxl345_of_match);

static struct i2c_driver adxl345_i2c_driver = {
        .driver = {
                .name   = "adxl345_i2c",
                .of_match_table = adxl345_of_match,
        },
        .probe          = adxl345_i2c_probe,
        .remove         = adxl345_i2c_remove,
        .id_table       = adxl345_i2c_id,
};

module_i2c_driver(adxl345_i2c_driver);
```

From drivers/iio/accel/adxl345_i2c.c

I2C Driver

- On non-DT platforms, the `struct i2c_board_info` structure allows to describe how an I2C device is connected to a board.
- Such structures are normally defined with the `I2C_BOARD_INFO()` helper macro.
  - Takes as argument the device name and the slave address of the device on the bus.
- An array of such structures is registered on a per-bus basis using `i2c_register_board_info()`, when the platform is initialized.

I2C Driver

```
static struct i2c_board_info __initdata em7210_i2c_devices[] = {
        {
                I2C_BOARD_INFO("rs5c372a", 0x32),
        },
};

...

static void __init em7210_init_machine(void)
{
        register_iop32x_gpio();
        platform_device_register(&em7210_serial_device);
        platform_device_register(&iop3xx_i2c0_device);
        platform_device_register(&iop3xx_i2c1_device);
        platform_device_register(&em7210_flash_device);
        platform_device_register(&iop3xx_dma_0_channel);
        platform_device_register(&iop3xx_dma_1_channel);

        i2c_register_board_info(0, em7210_i2c_devices,
                ARRAY_SIZE(em7210_i2c_devices));
}
```

From arch/arm/mach-iop32x/em7210.c

I2C Driver

- In the Device Tree, the I2C controller device is typically defined in the `.dtsi` file that describes the processor.
  - Normally defined with `status = "disabled"`.
- At the board/platform level:
  - the I2C controller device is enabled (`status = "okay"`)
  - the I2C bus frequency is defined, using the `clock-frequency` property.
  - the I2C devices on the bus are described as children of the I2C controller node, where the `reg` property gives the I2C slave address on the bus.
- See the binding for the corresponding driver for a specification of the expected DT properties. Example: `Documentation/devicetree/bindings/i2c/i2c-omap.txt`

I2C Driver

Definition of the I2C controller

```
i2c0: i2c@01c2ac00 {
        compatible = "allwinner,sun7i-a20-i2c",
                     "allwinner,sun4i-a10-i2c";
        reg = <0x01c2ac00 0x400>;
        interrupts = <GIC_SPI 7 IRQ_TYPE_LEVEL_HIGH>;
        clocks = <&apb1_gates 0>;
        status = "disabled";
        #address-cells = <1>;
        #size-cells = <0>;
};
```

From arch/arm/boot/dts/sun7i-a20.dtsi

I2C Driver

Definition of the I2C device

```
&i2c0 {
        pinctrl-names = "default";
        pinctrl-0 = <&i2c0_pins_a>;
        status = "okay";

        axp209: pmic@34 {
                compatible = "x-powers,axp209";
                reg = <0x34>;
                interrupt-parent = <&nmi_intc>;
                interrupts = <0 IRQ_TYPE_LEVEL_LOW>;

                interrupt-controller;
                #interrupt-cells = <1>;
        };
};
```

I2C Driver

- The `->probe_new()` function is responsible for initializing the device and registering it in the appropriate kernel framework. It receives as argument:
  - A `struct i2c_client` pointer, which represents the I2C device itself. This structure inherits from `struct device`.
- Alternatively, the `->probe()` function receives as arguments:
  - A similar `struct i2c_client` pointer.
  - A `struct i2c_device_id` pointer, which points to the I2C device ID entry that matched the device that is being probed.
- The `->remove()` function is responsible for unregistering the device from the kernel framework and shut it down. It receives as argument:
  - The same `struct i2c_client` pointer that was passed as argument to `->probe_new()` or `->probe()`