# Real Time Systems and Control Applications

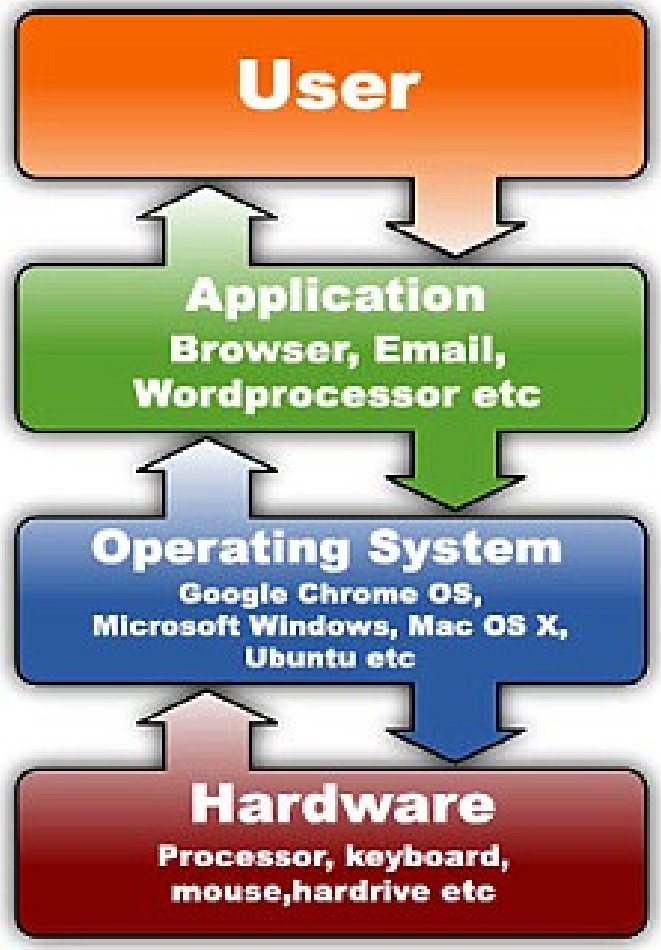
Contents

RTOS and Kernel Module

# Review of Concepts

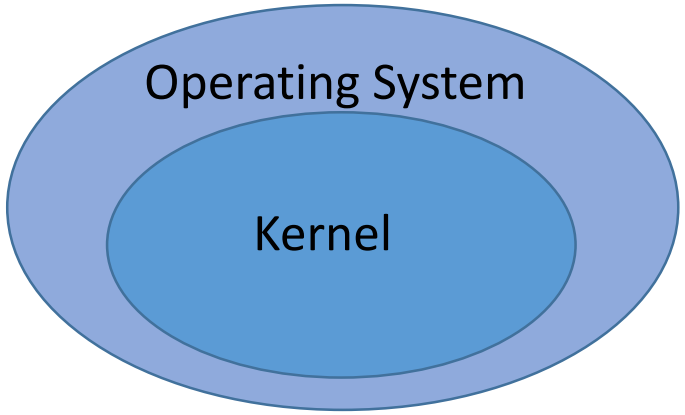
- What is an operating system?
  - System software that manages hardware and software resources and provides common services for computer programs.

- What does an OS do?
  - System Supervision
  - Resource Allocation
  - Security
  - Communication Services
  - …

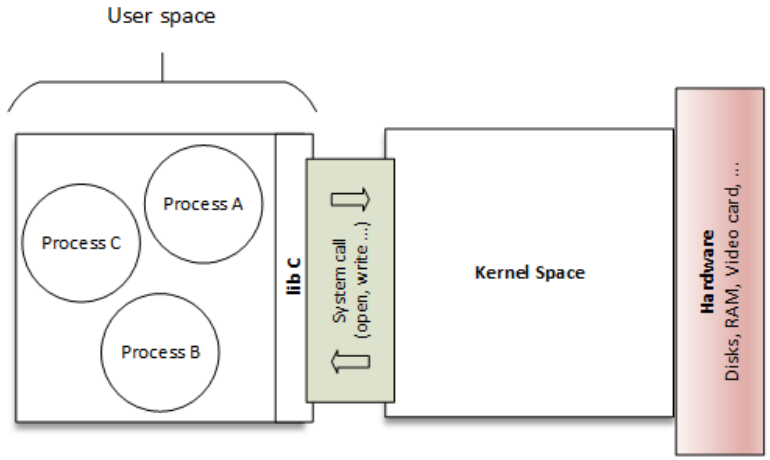


# Kernel and Kernel Space

- Services provided by Kernels
  - Process management
  - Memory Management
  - File System
  - Scheduling
  - Interrupt Handling
  - Inter-process Communication and Networking




- User Space is the space in memory where user processes run
- Kernel Space is the space in memory where kernel processes run



# User and Kernel (or Supervisor) Modes

- The set of instructions are usually divided into two classes:
  - Those that can be executed by a user
  - Those that can be executed by the kernel

| Operation Modes | Kernel Mode | User Mode |
|---|---|---|
| 1. | All the instruction are allowed, including OS routings | Only limited instruction are allowed |
| 2. | Unlimited hardware access | Direct access to the hardware and the memory is prohibited to avoid malicious users |

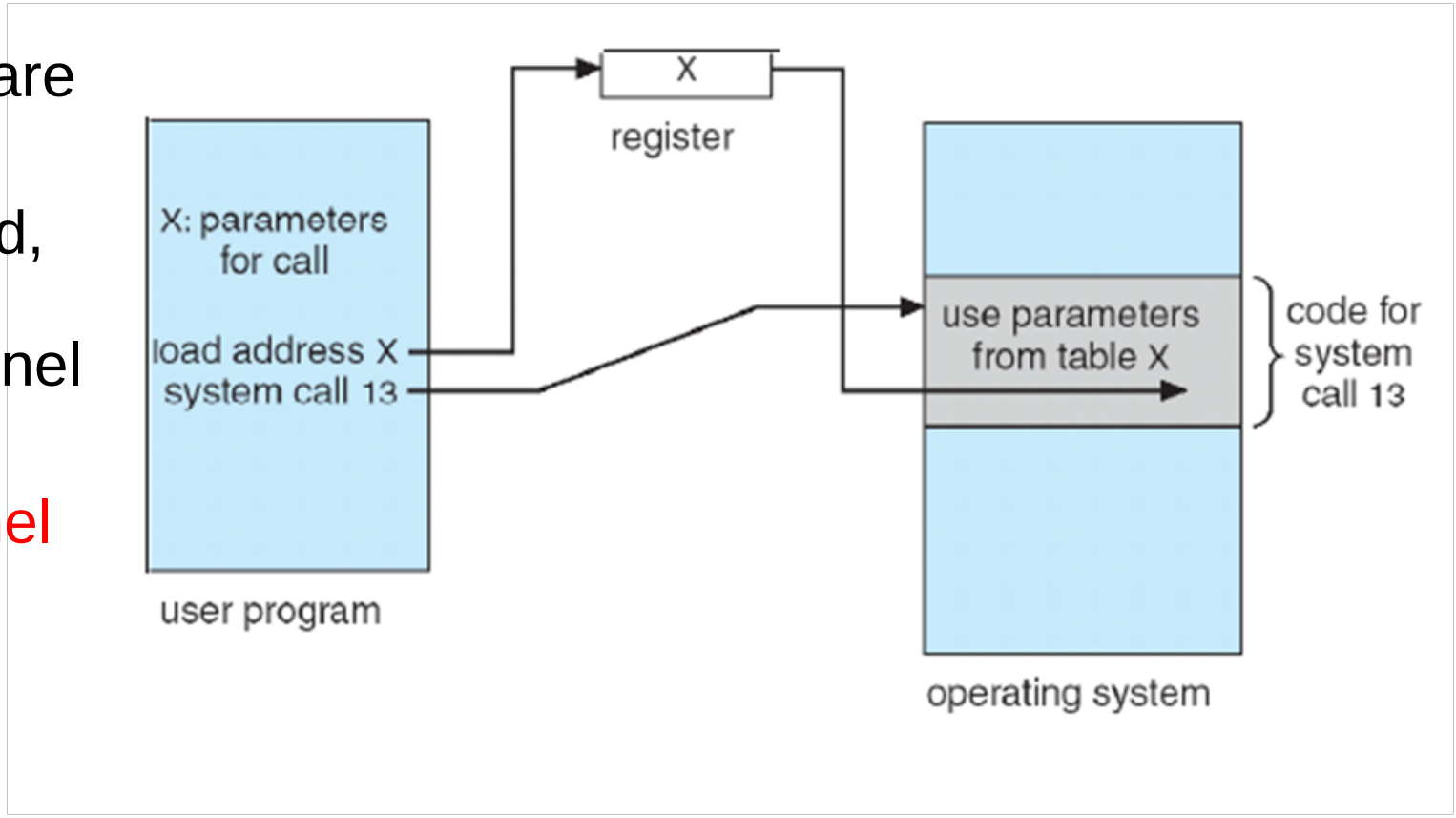
# How OS Delivers The Services To User Process?

- User processes request a service from Kernel by making a System Call

- The library procedure involved in a system call.
  - This procedure puts parameters of the system call in suitable registers and then issues a TRAP instruction.
  - The control is passed on to Kernel, it checks the validity of parameters, performs the requested service
  - When finished, a code is put in a register telling if the operation was carried out successfully or it failed.
  - A Return from TRAP instruction is then executed and the control is passed back to the user process.

# System Call And Argument Passing

- User Space and Kernel Space are in different spaces.
- When a System Call is executed, the arguments to the call are passed from User Space to Kernel Space
- A user process becomes a kernel process when it executes a system call.



# Real-Time Operating System (RTOS)

- RTOS
  - Designed for real-time tasks, where correctness depends on logic of the result and response time
  - Hard or Soft RTS depends on how to define cost of missing deadline

- Examples of commercial RTOSs:

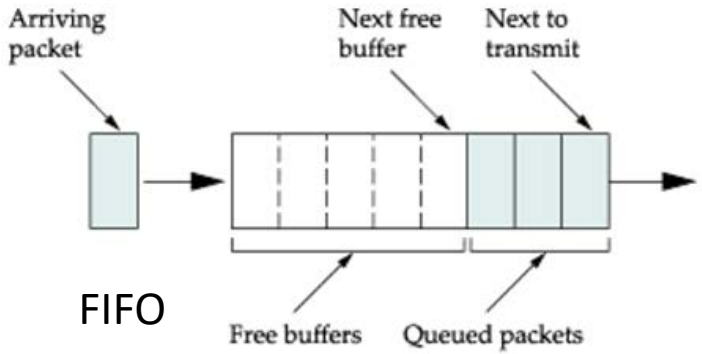
  VxWorks, RTLinux, LynxOS, Windows **CE**, RTAI, QNX, etc.

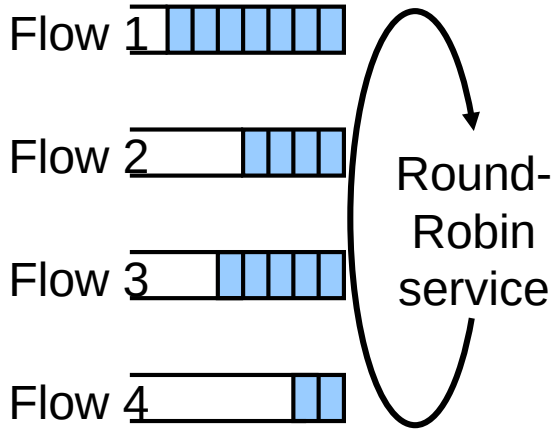
- What makes an OS Real Time?
  - Guarantee that deadlines are met
  - Predictability
  - Not necessarily fast, and may be mediocre throughputs

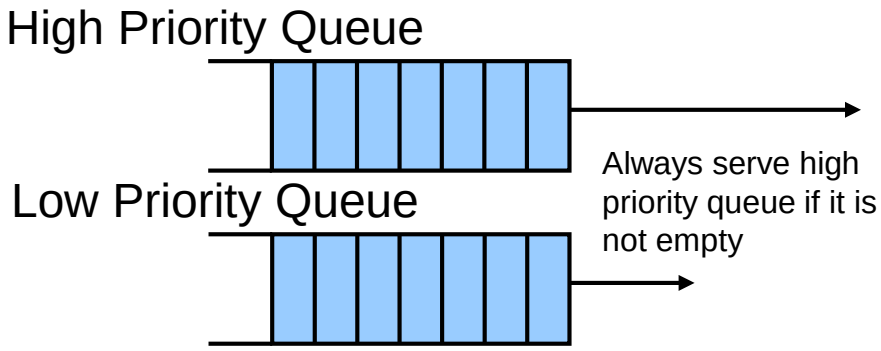
# Process Scheduling

- OS must decide which process to run first and in what order the remaining processes should run.

- Scheduling Algorithms
  - FIFO
  - Round Robin
  - Priority Queueing
  - …



# Objectives of A Scheduler

- A good scheduling algorithm for non-real-time system has the following objectives:

  Fairness: make sure that each process gets its fair share of CPU.

  Efficiency: keep the CPU busy to serve as much workload as possible.

  Response Time: minimize waiting time of users to obtain results

  Throughput: maximize the number of tasks processed per hour.

- Objective of RTOS

  Meet deadlines!

# OS v.s. RTOS

Difference between OS  and RTOS.

| | Operating System | Real-Time Operating Systems |
|---|---|---|
| **Design philosophy** | Time-sharing | Event driven |
| **Design requirement** | High throughput | Schedulability |
| **Performance metric** | Fast average response | Ensured worst-case response |
| **Overload** | Fairness | Meet critical deadlines |

Schedulability is the ability of tasks to meet all hard deadlines

# Is LINUX a RTOS?

- Linux provides limited kernel-mode preemption to execute any task during system calls
  - Kernel programs can always preempt user-space programs
  - Until 2.6 kernels, the kernel itself was not preemtible. Kernel preemption has been introduced in 2.6 kernels, and one can enable or disable the preemption.

- If without kernel mode pre-emption, a kernel task may have exclusive access to some data for a long time delaying a RT task

- Linux reorders requests from multiple processes and batch operations to use the hardware more efficiently

# Two Approaches To Implement RTOS Based on Linux

- Add a new layer of RT Kernel with full control of interrupts and processor key features.
  - RTAI and RTLinux, both use real-time kernel as the main kernel, and treat Linux OS the lowest running task
- Make necessary changes in Linux Kernel to make it suitable for real time applications. (Preempt RT)
  - Implement RT tasks as kernel modules

Note that Preempt RT in labs is provided by National Instruments as part of NI myRIO, so the kernel threads are preemptable.

# Kernel Module

- A Kernel module is piece of code that can be loaded and unloaded into the kernel upon demand. They extend the functionality of the kernel without the need to reboot the system.

- We do not need to include all the possible anticipated functionality into the base kernel.
  - Do not waste memory, and
  - Users do not have to rebuild and reboot the base kernel every time they require new functionality

# Related Commands

- insmod, rmmod, lsmod, modinfo

Insert a module: `insmod ./myModule.ko`

Remove a module: `rmmod myModule`

`lsmod` shows what are currently loaded into the kernel

`modinfo` shows information about a Linux Kernel module

# Development Fundamentals

A real time task running as a kernel module consists of three sections:

1. Function init_module(): Invoked by insmod to prepare for later invocation of module's functions. It can be used to allocate required system resources, declare and start tasks etc.
2. Task specific code (based on POSIX API)
3. Function cleanup_module(): Invoked by rmmod to inform kernel that the module's functions will not be called any more. A good place to release all of the system resources allocated during the lifetime of the module, stop and delete tasks etc.

# A Simple Example

- #include <linux/module.h>
- #include <linux/kernel.h>

- int init_module(void) {
  printk(KERN_INFO "Hello World\n");
  return 0;
  }

- void cleanup_module(void){
  printk(KERN_INFO "Goodbye Cruel World!\n");
  }

# Compiling Kernel Modules

A Sample MakeFile

```
obj-m +=hello.o

all:
  make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules

clean:
  make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

# module_init() and init_module()

In the current versions of Linux it is possible to use any suitable name for the init and cleanup functions.

In order to do that one has to begin the name of functions with _ _init and _ _exit macros, then use module_init() and module_exit() macros after defining these functions.

The module_init() macro defines which function is to be called at module insertion time

Init_module() is the default entrance of module initialization function, and one can define his or her own initialization function in module_init

```
#include <linux/module.h>    /*Every module requires it*/
#include <linux/kernel.h>    /*KERN_INFO needs it*/
#include <linux/init.h>      /* Required by macros*/

static char *my_string __initdata = "dummy";
static int my_int __initdata = 4;
/* Init function with user defined name*/
static int __init hello_4_init(void)  {
  printk (KERN_INFO "Hello %s world, number %d\n",my_string, my_int);
  return 0;
}
/* Exit function with user defined name*/
static void __exit hello_4_exit(void)  {
   printk(KERN_INFO "Goodbye cruel world 4\n");
}
/*Macros to be used after defining init and exit functions*/
module_init(hello_4_init);
module_exit(hello_4_exit);
```

# Other Examples of Macros

- MODULE_LICENSE(): use GPL
- MODULE_DESCRIPTION(): To describe what the module does
- MODULE_AUTHOR(): Name of the person who wrote code for the module
- MODULE_SUPPORTED_DEVICES(): Declares what type of devices the module supports

# Example

```
#include <linux/module.h>        /*Every module requires it*/
#include <linux/kernel.h>        /*KERN_INFO needs it*/
#include <linux/init.h>          /* Required by macros*/
#define DRIVER_AUTHOR " Wenbo He"
#define DRIVER_DESC "SE 4AA4/6GA3 example4"
static char *my_string __initdata = "dummy";
static int my_int __initdata = 4;
/* Init function with user defined name*/
static int __init hello_4_init(void)
{
  printk (KERN_INFO "Hello %s world, number %d\n",my_string, my_int);
  return 0;
}
```

```
/* Exit function with user defined name*/
static void __exit hello_4_exit(void)
{
    printk(KERN_INFO "Goodbye cruel world 4\n");
}

// Task specific functions go here.

/*Macros to be used after defining init and exit functions*/
module_init(hello_4_init);
module_exit(hello_4_exit);

MODULE_LICENSE("GPL"); /* Avoids kernel taint message*/
MODULE_AUTHOR(DRIVER_AUTHOR); /* Who wrote this module? */
MODULE_DESCRIPTION(DRIVER_DESC); /* What does this module do */
```

# Passing Commandline Arguments

- Command line arguments can be passed to modules but NOT with *argv* and *argc*
- First declare variables that will be used to store values passed on commandline
- Then set them up using macro **module_param**(name, type, permissions)
- At run time insmod will fill up the variables with values pass

# Example:

static int my_int = 5;  (initialize defaults)

module_param(my_int, int, S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH)

MODULE_PARM_DESC(my_int, "An integer")

Macro function, MODULE_PARM_DESC(), is used to document arguments that the module can take.

After compiling, the module with parameter can be inserted as (Assume the model is called hello)

insmod hello.ko my_int=10