

## Kernel programming

Kernel has access to *all* resources  
Kernel programs not subject to any constraints for memory access  
or hardware access  
⇒ faulty kernel programs can cause system crash

## Interaction between kernel and user programs

Kernel provides its functions only via special functions, called *system calls*  
standard C-library provides them  
Have strict separation of kernel data and data for user programs  
⇒ need explicit copying between user program and kernel

## Structure of kernel

Simplified structure of kernel;

```
initialise data structures at boot time;  
while (true) {  
    while (timer not gone off) {  
        assign CPU to suitable process;  
        execute process;  
    }  
    select next suitable process;  
}
```

The process can execute code either in user mode or kernel mode

In addition, have *interrupts*:

kernel asks HW to perform certain action  
HW sends interrupt to kernel which performs desired action

Key points:

- No user context available while interrupts are processed
- interrupts must be processed quickly  
⇒ any code called from interrupts must not sleep

## Linux kernel modes

A kernel program may be in two main modes:

- Working for user programs by executing a system call (*user context*)
- Handling an interrupt (eg by a device) *interrupt context*

In user mode, have access to user data

Any code running in user mode may be pre-empted at any time by an interrupt

Interrupts have priority levels

Interrupt of lower priority are pre-empted by interrupts of higher priority

## Kernel modules

can add code to running kernel

useful for providing device drivers which are required only if hardware present

`modprobe` inserts module into running kernel

`rmmmod` removes module from running kernel (if unused)

`lsmod` lists currently running modules

## Concurrency issues in the kernel

Consequence for handling concurrency in the kernel:

Manipulation of data structures which are shared between

- code running in user mode and code running in interrupt mode
- code running in interrupt mode

must happen only within critical regions

In multi-processor system even manipulation of data structures shared between code running in user mode must happen only within critical sections

## Achieving mutual exclusion

Two ways:

- **Semaphores:** when entering critical section fails, current process is put to sleep until critical region is available  
⇒ only usable if *all* critical regions are in user context
- **Spinlocks:** processor tries repeatedly to enter critical section  
Usable anywhere  
Disadvantage: Have busy waiting

## Use of semaphores

Have two kinds of semaphores:

- **Normal semaphores**
- **Read-Write semaphores:** useful if some critical regions only read shared data structures, and this happens often

## A tour of the Linux kernel

Major parts of the kernel:

- Device drivers: in the subdirectory `drivers`, sorted according to category
- file systems: in the subdirectory `fs`
- scheduling and process management: in the subdirectory `kernel`
- memory management: in the subdirectory `mm`
- networking code: in the subdirectory `net`
- architecture specific low-level code (including assembly code): in the subdirectory `arch`
- include-files: in the subdirectory `include`