

## Linux Device Drivers

## Device drivers

View from user space:

Have special file in `/dev` associated with it, together with five systems calls:

- `open`: make device available
- `read`: read from device
- `write`: write to device
- `ioctl`: Perform operations on device (optional)
- `close`: make device unavailable

## Kernel side

Each file may have functions associated with it which are called when corresponding system calls are made  
`linux/fs.h` lists all available operations on files  
Device driver implements at least functions for `open`, `read`, `write` and `close`.

## Categorising devices

Kernel also keeps track of

- **Physical dependencies** between devices. Example:  
devices connected to a USB-hub
- **Buses**: Channels between processor and one or more devices.  
Can be either physical (eg pci, usb), or logical
- **Classes**: Sets of devices of the same type, eg keyboards, mice

# Handling Interrupts in Device Drivers

Normal cycle of interrupt handling for devices:

- Device sends interrupt
- CPU selects appropriate interrupt handler
- Interrupt handler processes interrupt
  - Two important tasks to be done:
    - Data to be transferred to/from device
    - Waking up processes which wait for data transfer to be finished
- Interrupt handler clears interrupt bit of device
  - Necessary for next interrupt to arrive

Interrupt processing time must be as short as possible

Data transfer fast, rest of processing slow

⇒ Separate interrupt processing in two halves:

- **Top Half** is called directly by interrupt handler  
Only transfers data between device and appropriate kernel buffer and schedules software interrupt to start Bottom half
- **Bottom half** still runs in interrupt context and does the rest of the processing (eg working through the protocol stack, and waking up processes)

# Memory Management

# Memory Management

Management of a **limited resource**:

(Memory hunger of applications increases with capacity!)

⇒ **Sophisticated algorithms needed**, together with support from HW and from compiler and loader.

Key point: program's view memory is set of memory cells starting at address 0x0 and finishing at some value (**logical address**)

Hardware: have set of memory cells starting at address 0x0 and finishing at some value (**physical address**)

Want to be able to store memory of several programs in main memory at the same time



need suitable mapping from logical addresses to physical addresses:

- at **compile time**: absolute references are generated (eg MS-DOS .com-files)
- at **load time**: can be done by special program
- at **execution time**: needs HW support

Address mapping can be taken one step further:

**dynamic linking**: use only one copy of system library

⇒ OS has to help: same code accessible to more than one process

# Swapping

If **memory demand is too high**, memory of some processes is **transferred to disk**

Usually **combined with scheduling**: low priority processes are swapped out

Problems:

- **Big transfer time**
- What to do with **pending I/O?**

First point reason why **swapping is not principal memory management technique**

# Fragmentation

Swapping raises two problems:

- over time, many **small holes** appear in memory (**external fragmentation**)
- programs only a little smaller than hole  $\Rightarrow$  **leftover too small to qualify as hole** (**internal fragmentation**)

**Strategies** for choosing holes:

- **First-fit**: Start from beginning and use first available hole
- **Rotating first fit**: start after last assigned part of memory
- **Best fit**: find smallest usable space
- **Buddy system**: Free holes are administered according to tree structure; smallest possible chunk used

# Paging

Alternative approach: Assign **memory of a fixed size** (**page**)

⇒ avoids **external fragmentation**

**Translation** of logical address to physical address **done via page table**

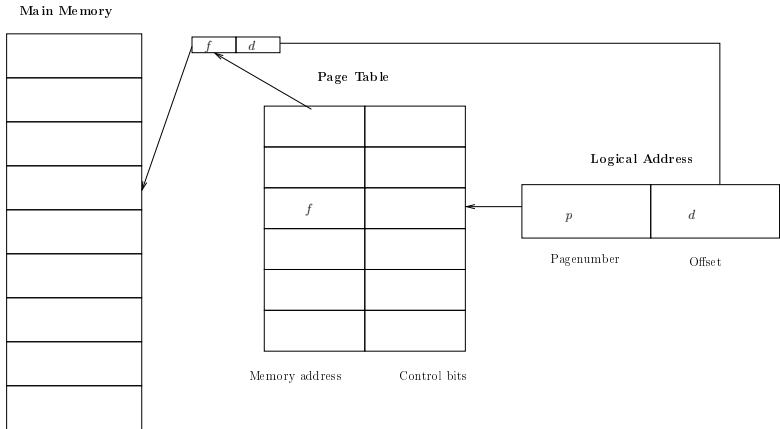
**Hardware support mandatory** for paging:

If page table **small**, use **fast registers**. Store large page tables in main memory, but **cache most recently used entries**

**Instance of a general principle:**

Whenever **large lookup** tables are required, **use cache** (**small but fast storage**) to store most recently used entries

**Memory protection easily added** to paging:  
protection information **stored in page table**



# Segmentation

Idea: Divide memory according to its usage by programs:

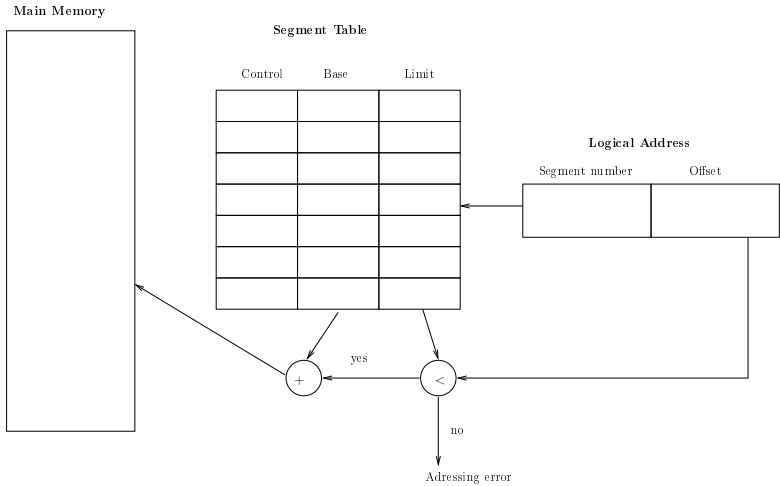
- Data: mutable, different for each instance
- Program Code: immutable, same for each instance
- Symbol Table: immutable, same for each instance, only necessary for debugging

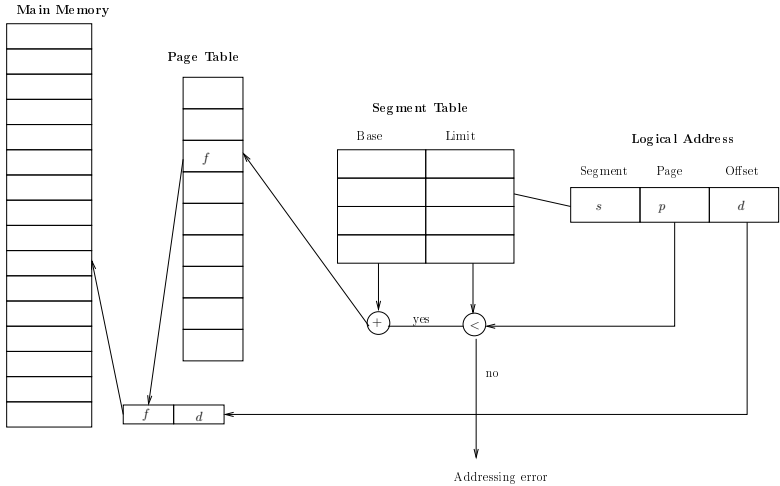
Requires again HW support

can use same principle as for paging, but have to do overflow check

Paging motivated by ease of allocation, segmentation by use of memory

⇒ combination of both works well (eg 80386)







# Virtual memory

**Idea:** complete separation of logical and physical memory

⇒ Program can have extremely large amount of virtual memory

works because most programs use only small fraction of memory intensively.

Efficient implementation tricky

*Reason:* Enormous difference between

- memory access speed (ca. 60ns)
- disk access speed (ca. 6ms)

**Factor 100,000 !!**

# Demand Paging

Virtual memory implemented as demand paging: memory divided into **units of same length** (**pages**), together with valid/invalid bit

Two strategic decisions to be made:

- Which process to **“swap out”** (move whole memory to disk and block process): swapper
- **which pages to move to disk** when additional page is required: pager

**Minimisation of rate of page faults** (page has to be fetched from memory) **crucial**

If we want 10% slowdown due to page fault, require fault rate  $p < 10^{-6}!!$

# Page replacement algorithms

## 1.) FIFO:

easy to implement, but does not take locality into account

Further problem: Increase in number of frames can cause increase in number of page faults (Belady's anomaly)

## 2.) Optimal algorithm:

select page which will be re-used at the latest time (or not at all)

⇒ not implementable, but good for comparisons

3.) **Least-recently used:**  
use past as guide for future and replace page which has been unused for the longest time

Problem: **Requires a lot of HW support**

Possibilities:

- Stack in microcode

- Approximation using reference bit: HW sets bit to 1 when page is referenced.

Now use FIFO algorithm, but skip pages with reference bit 1, resetting it to 0

⇒ **Second-chance algorithm**

# Thrashing

- If **process lacks frames it uses constantly**, page-fault rate very high.
- ⇒ **CPU-throughput decreases dramatically.**
- ⇒ **Disastrous effect on performance.**

Two solutions:

1.) **Working-set model** (based on locality):

Define working set as set of pages used in the most recent  $\Delta$  page references

keep only **working set in main memory**

⇒ Achieves high CPU-utilisation and prevents thrashing

Difficulty: **Determine the working set!**

Approximation: **use reference bits**; copy them each 10,000 references and define working set as pages with reference bit set.

## 2.) Page-Fault Frequency:

takes direct approach:

- give process additional frames if page frequency rate high
- remove frame from process if page fault rate low

# Memory Management in the Linux Kernel

Have only four segments in total:

- Kernel Code
- Kernel Data
- User Code
- User Data

Paging used as described earlier

Have elaborate permission system for pages

# Kernel memory and user memory

Have separate logical addresses for kernel and user memory  
For 32-bit architectures (4 GB virtual memory):

- kernel space addresses are the upper 1 GB of address space ( $\geq 0xC0000000$ )
- user space addresses are 0x0 to 0xBFFFFFFF (lower 3 GB)

kernel memory always mapped but protected against access by user processes



# Kernel memory and user memory

For 64-bit architectures:

- kernel space address are the upper half of address space ( $\geq 0x8000\ 0000\ 0000\ 0000$ )
- user space addresses are  $0x0$  to  $0x7fff\ ffff\ ffff$ , starting from above.

kernel memory always mapped but protected against access by user processes

## Page caches

Experience shows: have repeated cycles of allocation and freeing same kind of objects (eg inodes, dentries)  
can have pool of pages used as cache for these objects (so-called slab cache)  
cache maintained by application (eg file system)  
`kmalloc` uses slab caches for commonly used sizes

# Summary

Two topics:

## Device drivers

implement `open`, `read`, `write` and `close` with common structure

## Memory management

- Management of a limited resource  $\Rightarrow$  serious effort required
- Need to isolate memory for each process
- If memory demand is too high, need to swap out part of the memory of a process
- Looked at paging and segmentation to achieve this
- Requires hardware support