ProSO

Operating Systems Project (PROSO)
Barcelona School of Informatics (FIB)
Universitat Politècnica de Catalunya (UPC)
2010-2011

FIB

Goals

- Get a good O.S. internal understanding
  - P1: Be able to program low level O.S. code, developing from scratch the basic system components
  - P2: Be able to add functionalities to an existing O.S.

Project

- Project 1
  - Implement a minimal O.S. kernel (ZeOS) Linux based.
- Project 2
  - Implement a Linux Device Driver as a Linux Kernel Module (LKM).

Course Description

- 10h/week of work
  - Lecture classes (1-2 h. not every week)
  - Lab classes
    - Scheduling-fixed (4h/week)
      - » with teacher assistance (2h/week)
    - On your own (5h/week)
- Intermediate meeting and code delivery to your advisor to follow the project evolution
- Teams of two students
  - Use the forum to put your group
  - Name; DNI; Lab group to assist

Grading

- Project 1 (Zeos) → 70%
  - Intermediate delivery P1.1 → 10%
  - Intermediate delivery P1.2 → 25%
  - Final Project delivery & Global evaluation P1.3→ (25% + 10%)
  - Questionnaire to check out the minimum concepts
- Project 2 (Linux modules)→ 30%
  - Final Project delivery → 30%
  - Questionnaire to check out the minimum concepts

Deliveries

- No one delivery is mandatory
- Intermediate deliveries
  - Evaluation & Feedback
    - You must meet with your advisor
- Important dates (Fridays)
  - P1.1: October 1^{st}
  - P1.2: October 29^{th}
  - P1.3: November 11^{th}

  - P2.1: December 17^{th}

Validation Test (Questionnaire)

- Questions about delivery (individual test)
- Important dates
  - QE1.1: October 11th
  - QE1.2: November 15th
  - QE1.3: December 13th

  - QE2.1: December 22th

Initial scheduling (can be modified)

	Monday	Tuesday	Wednesday	Thursday	Friday	
9/13/2010	T-S1.1(10/40/MAS)					9/17/2010
9/20/2010	T_S1.1(10/40/MAS)					9/24/2010
9/27/2010					S1.1	10/1/2010
10/4/2010	T-S1.2(10/40/MAS)					10/8/2010
10/11/2010	T-S1.2(10/40/MAS) **QE1.1**					10/15/2010
10/18/2010						10/22/2010
10/25/2010	T-S1.3(10/40/MAS)				S1.2	10/29/2010
11/1/2010						11/5/2010
11/8/2010	T-S1.3(10/40/MAS)					11/12/2010
11/15/2010	**QE1.2**					11/19/2010
11/22/2010					S1.3	11/26/2010
11/29/2010	T-S2.1(10/40/MAS)					12/3/2010
12/6/2010						12/10/2010
12/13/2010	**QE1.3**				S2.1	12/17/2010
12/20/2010			(Monday) Q2.1	(Tuesday)		12/24/2010

Holidays
Change school day
Submissions

Material

- http://studies.ac.upc.es/FIB/PROSO
- **RACO**
  - **Deliveries**
  - **Forum**
  - **Noticeboard**
- **Main source of documentation**
  - **PROSO web page**
  - **P1 Documentation (in web page)**

Requirements

- OS concepts (SO)
- Computer Architecture Concepts (EC2)
- Data Structures and Algorithms (PRED)

Overview of P1

Operating Systems Project (PROSO)
Barcelona School of Informatics (FIB)
Technical University of Catalunya (UPC)

Bibliography

- Understanding The Linux Kernel
  - Appendix A: System Startup
  - Chap.1, chap.2, chap.4
- Basic OS books
  - Galvin&Silberschatz&Peterson
  - Tanembaum
  - …

Goals and efforts in the Project

- OS job
- OS code and data characterization
  - Event-driven code:
    - How to manage functions
    - Independent/dependent layers
    - System initialization
  - Resource management:
    - Object representation
    - Object management: LISTS

To Take into account... (I)

- kernel vs. user-space applications

  (*From* Robert Love *"Linux Kernel Development"*)
  - The kernel does not have access to the C library
  - The kernel is coded in GNU C
  - The kernel lacks memory protection like user-space
  - The kernel cannot easily use floating point
  - The kernel has a small fixed-size stack
  - Synchronization and concurrency are concerns within the kernel
  - Portability is important

To Take into account… (II)

- Efficiency is important
  - Code that executes a lot of times

- OS has to guaranty the machine integrity
  - Robustness is important
  - User code is not reliable

- Clarity, scalability and modularity are important

To Take into account… (III)

- Some OS code has to be machine dependent
  - Assembly language

  - This code has to observe the compiler ABI
    - Application Binary Interface (ABI): Set of conventions that allows a linker to combine separately compiled modules into one unit without recompilation, such as
      - Calling conventions
      - Machine interface
      - Operating System interface

P1 files: What do you have?

- P1 documentation → see web page
- Basic Zeos files
  - Basic file's structure
  - Makefile
  - Assembler Macros
  - Basic Boot till system.c
  - Basic memory initialization
    - Segmentation
    - Pagination
  - Functions for accessing hardware
    - IDT (Interrupt Descriptor Table)
    - Memory
    - …

What are you going to implement?

- Mechanisms for entering the system (P1.1)
  - Interruptions, exceptions, system calls
- Process Management (P1.2)
- Input/Output Management (P1.3)

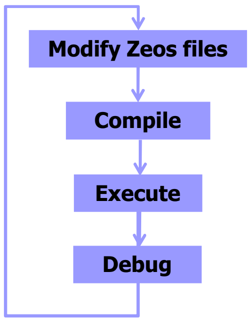
Previous concepts you will need

- Interrupts/Exceptions management:
  - Interrupt
  - Trap
  - System gate. Handlers.
- Changing to Protected mode
- IRQ's Initialization
- Components
  - Global Descriptor Table (GDT) (Chap.4)
  - Interrupt Descriptor Table (IDT) (Chap.4)
  - Task State Segment (TSS) (Chap.4)

Working environment

- Remote boot using REMBO
  - Ubuntu 6.06
- Work in the local PC
  1. Get files from albanta to the PC (sftp)
  2. WORK
  3. **Put files** from PC to albanta (sftp)

WORK



- Execute options
  - Boot Zeos from a floppy
    - Slowdown the development process
  - **Boot Zeos on a machine emulator: Bochs**

Tools

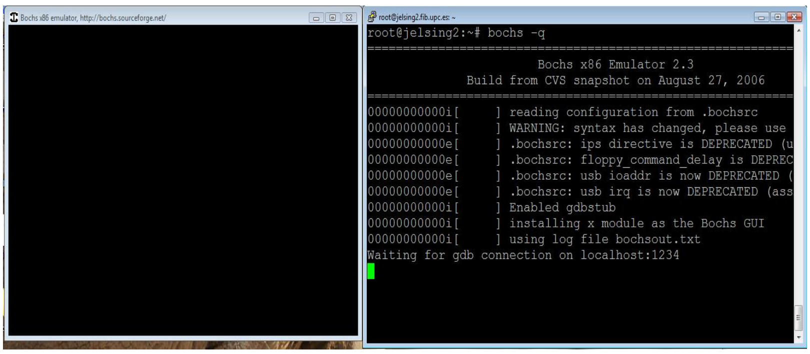
- Makefile (DONE in SO courses)
- Bochs emulator → see the documentation
- nm
  - To see where data is mapped
  - Read man pages
- objdump
  - To see code memory addresses
  - Read man pages

What do we ask you for in P0?

- Be familiar with the working environment
  - Linux (Development framework)
  - Bochs (emulator)
  - ZeOS (Basic files)
- Understand the boot mechanism: steps
- Build a ZeOS version from the basic files provided
- Be familiar with the debugger:
  - GDB
  - Bochs' internal debugger

Bochs emulator (v2.3)

- http://bochs.sourceforge.net



Boot Process General Overview

- Bootstrap: initializing the machine and the operating system
- Highly dependent on the computer architecture
- Intel Boot process
  - BIOS (*Basic Input/Output System*) code
    - Executed in real mode
    - Tests on the computer hardware
    - Initializes hardware devices
    - Finds the suitable boot device and loads the OS boot code (*boot loader*)
  - The "boot loader" completes the system load

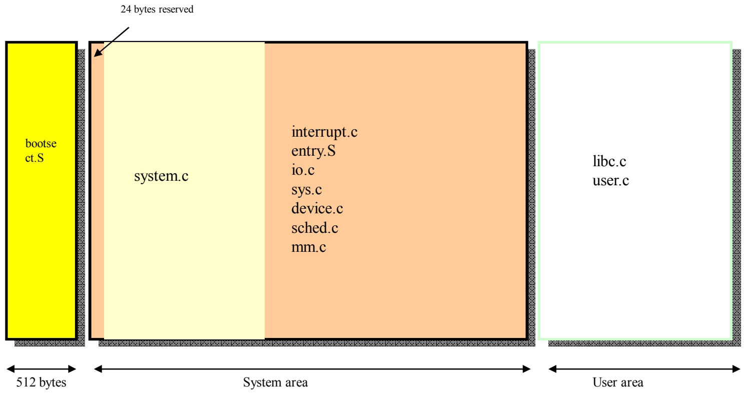
FIB 25 UPC

ZeOS Boot file

- Created by `build.c`
  - Boot Sector (bootsect.S)
    - 512 bytes to fit in a floppy sector
  - System code
    - ZeOS code
    - Loads user code (start address and size must be indicated)
  - User code
    - ZeOS doesn't have a program loader

FIB 26 UPC

ZeOS Boot file scheme



Main() (in system.c)

- Initializes a minimum execution context
  - Prepares Null process environment
  - Initializes Segment Registers (memory management)
  - Initializes Null process kernel stack
  - Sets gdtr and idtr from GDT and IDT address
- Initializes IDT (set_idt)
  - There aren't any defined interrupts
- Fills the first memory page with the system parameters
- Jumps to user space (main() function in user.c)

TO DO

- Some user code modifications
  - To get used to the working environment
- System code modifications
  - Modify printc to consider \n characters
  - Include some additional messages when booting
  - Output in the Ubuntu console
- Understand the basic ZeOSFiles.

FIB 29

Deliverable P1.1
Mechanisms for entering the system

Operating Systems Project (PROSO)
Barcelona School of Informatics (FIB)
Technical University of Catalunya (UPC)

FIB

Contents

- Mechanisms for entering the system
- Overview
  1. Initialization
  2. Management
- Procedure for entering/exiting the system
- Exceptions
- Interruptions: clock and keyboard
- System calls: write()

Mechanisms for entering the system

- Always through the IDT (Interrupt Descriptor Table)
  - From 0 to 31
    - Exceptions
      - Synchronous, produced by the CPU control unit, only after terminating the execution of an instruction
    - Non-masked interrupts
  - From 32 to 47
    - Masked Interrupts
      - Asynchronous, generated by other hardware devices at arbitrary times
  - From 48 to 255
    - Software interrupts (Traps)
      - Synchronous, explicitly requested by the application

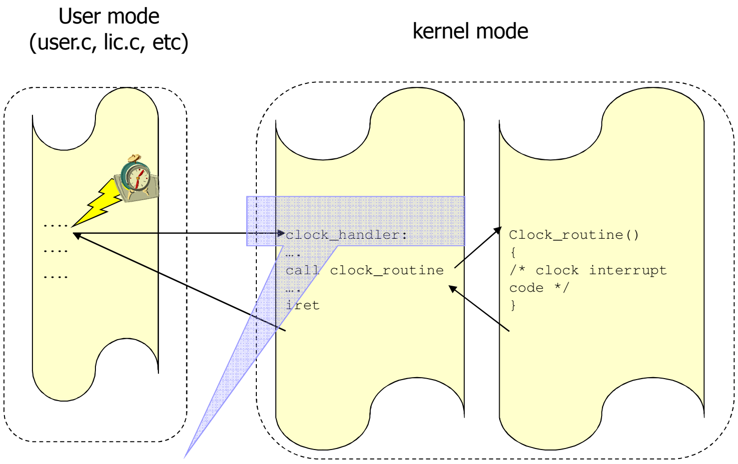
Overview: initialization

- Initialize the corresponding entry in the IDT
  - Each entry in the table has...
    - Interrupt number
    - Address to jump (*entry point*)
    - Privilege level
- Unmask the corresponding interrupt **<u>if needed</u>**
  - See enable_int function

Overview: management code

- Entry point→ handler (entry.S)
  - Assembly code
  - Basic hardware context management
- Service routine (depends on the interrupt)
  - C code
  - Specific algorithm

Example: clock interrupt behavior

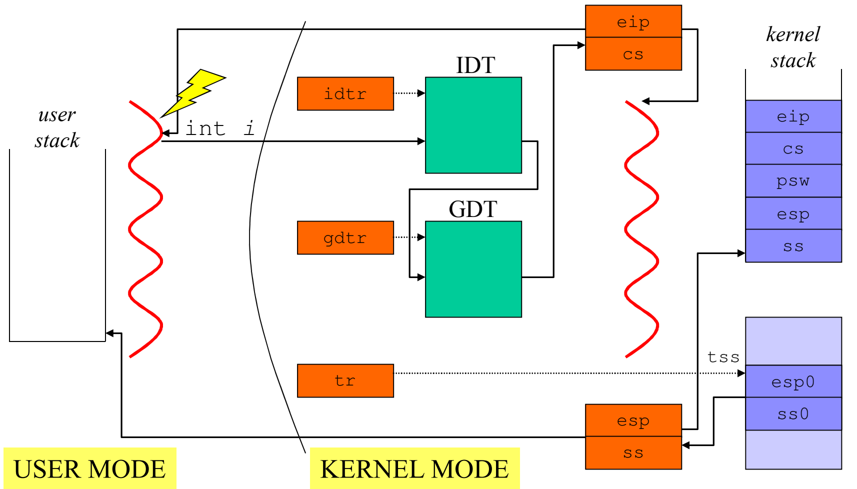


Entry point specified in the IDT

Procedure for entering the system

- Switch to protected execution mode (*HW*)
  - User mode→Kernel mode
- Save hardware context: CPU registers
  - ss, esp, psw, cs i eip (*HW*)
  - general purpose registers (SAVE_ALL macro)
- Execute service routine (*handler*)

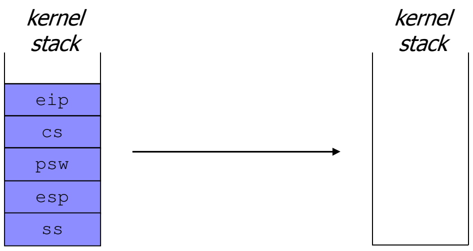
Procedure for entering the system



Procedure for exiting the system

- Restore hardware context
  - general purpose registers (RESTORE_ALL macro)
  - ss, esp, psw, cs i eip (iret instruction)
- Switch execution mode
  - Kernel mode -> User mode (iret instruction)

Procedure for exiting the system



These must be the contents of the kernel stack just before executing `iret` instruction

`iret` instruction empties the kernel stack and switches execution mode

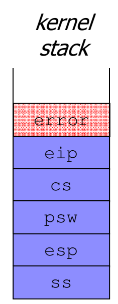
Exceptions

Exceptions list

0. Divide error
1. Debug
2. Not used
3. Breakpoint
4. Overflow
5. Bounds check
6. Invalid opcode
7. Device not available
8. Double fault*
9. Coprocessor segment overrun
10. Invalid TSS*
11. Segment not present*
12. Stack segment*
13. General protection*
14. Page fault*
15. Reserved
16. Floating point error
17. Alignment check*
18 to 31. Reserved

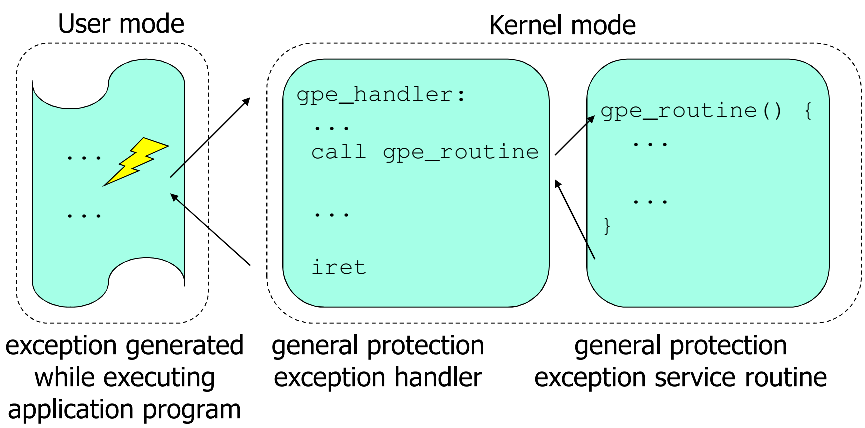
* hardware error code (4 bytes)

Handling exceptions: stack layout



For some exceptions, a hardware error code (or parameter), 4 bytes, is pushed in the kernel stack just after entering in kernel mode

Handling exceptions: overview



Handling exceptions: initialization

- Init IDT
  - one IDT entry per exception
  - void setInterruptHandler (int n, void (*f)(), int DPL)
    - n: nth IDT entry
    - f: exception handler routine address
    - DPL: max privilege level
  - e.g. DEFINE KERNEL_LEVEL AND USER_LEVEL CONSTANTS

    `setInterruptHandler(0,divide_error_handler,KERNEL_LEVEL)`

Handling exceptions: handler

- Save hardware context (SAVE_ALL)
- call exception service routine
- Restore hardware context (RESTORE_ALL)

YOU HAVE TO IMPLEMENT IT

- Remove (if any) the exception hardware error code from the kernel stack
- Return to user (iret)

Handling exceptions: service routines

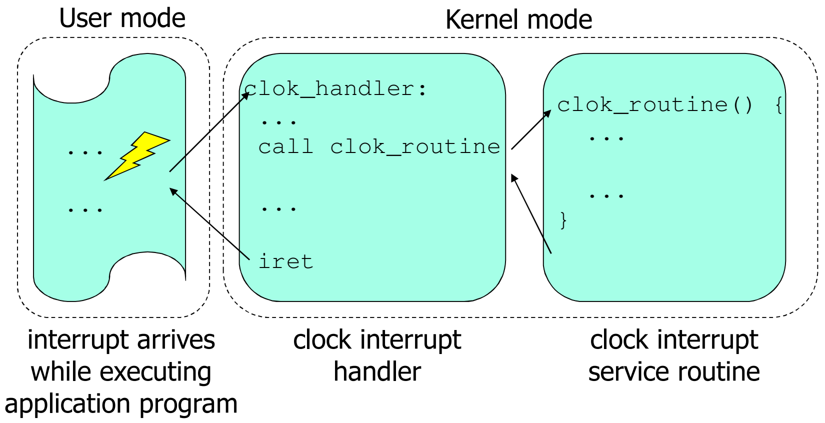
- Not managed in delivery 1

- Service routines will just print a message and waits forever
  - If an exception happens, and the kernel doesn't fix it, the program cannot continue

Interrupts

-clock
-keyboard

Handling interrupts: overview



Handling interrupts: initialization

- Interrupts initialization
  - Init IDT entry
    - 32: clock,
    - 33: keyboard
  - You have to decide when (in the code sequence) the system is prepared to receive interrupts in a save way.

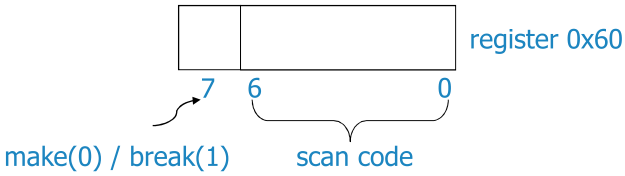
Handling interrupts: handler

- Similar to exceptions but:
  - No hardware error parameter in the kernel stack
  - It is necessary to notify the controller with the end of the interrupt management
    - It means that a new interrupt can be managed
    - End Of Interrupt (EOI)

```
movb $0x20, %al
outb %al, $0x20
```

Keyboard interrupt service routine

- Keyboard interrupt service routine
  - Write character at the down-right side of the screen
  - Access to keyboard data register
    - `int inb (int port)`



  - Scan code must be translated to character using the char_map table (interrupt.c)

Clock interrupt service routine

- Clock Service routine
  - Write “minutes: seconds” at the top-right side of the screen
  - You will need to implement helping functions
    - Itoa
    - Printk_xy
    - See the documentation

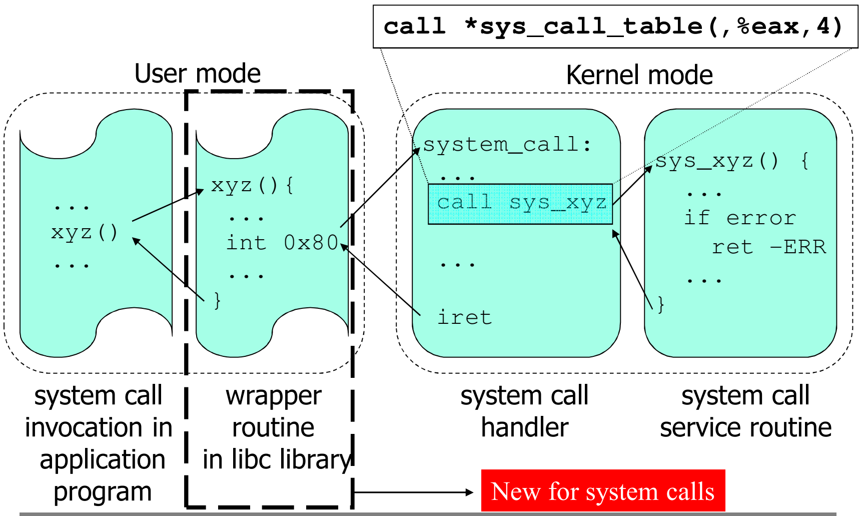
System calls

-write

Handling system calls

- System calls are generated by users
  - Intel: int assembly instruction (int idt_entry)
  - We must provide the users with an easy and portable way to invoke them
    - New layer: wrappers (libc.c)
  - User code is not reliable
    - **OS must validate all the data provided by the users**
      - Parameters
      - System calls identifiers
    - You have to decide which type of validation and where to include it (read the documentation)

Handling system calls: overview



55

Handling system calls: steps

- Initialization
- Wrapper routine
- System call handler
  - 1 for all the system calls
  - Redirects using a system call table
- System call service routine
  - 1 per system call

56

Handling system calls: initialization

- Init IDT
  - IDT entry: 128 (0x80)
  - void setTrapHandler (int n, void (*f)(), int DPL)
  - Similar to interrupts and exceptions

Handling system calls: wrapper (I)

- Written in assembly language and called by user C code
  - Observe the C compiler conventions
    - Which registers must be saved
    - How are parameters stacked
    - How results are returned
- Invoke the system call handler
  - Pass parameters
  - Identify the system call
  - Generate the trap
- Return the result to the user code

Handling system calls: wrapper (II)

- Pass parameters: Stack is not shared
  - Copy parameters from user stack to the registers
    - (first parameter) ebx, ecx, edx, esi, edi
  - Parameters in C are stacked from right to left
- Identify the system call
  - Copy system call number to eax
- Generate Trap: int $0x80
- Return to user code
  - If error: returns -1 and updates the libc errno variable

Handling system calls: handler

- Save hardware context
  - Registers with system call parameters are located in the top of the kernel stack
- Execute system call service routine
  - Specified by eax (error checking)
  - Using system_call_table
    - call *sys_call_table(,%eax,4)
- Update kernel context with the system call result
- Restore context
- Return to user

PARAMETERS(which,where)?

Handling system calls: service routines

- System call service routine
  - Check parameters
  - Access the process address space (if needed)
    - `copy_from_user`
    - `copy_to_user`
  - Specific system call code
    - Can include the invocation to a device dependent routine
      - DELIVERY 2
    - Sys_write_console

PARAMETERS (which,where)?

FIB

UPC

TO DO

- Handling exceptions
- Handling interruptions: clock and keyboard
- Handling system calls: write
- Some additional functions
  - Printk_xy
  - Itoa
  - See documentation!

FIB

UPC

Deliverable P1.2
Process management

-Kernel Data structures
-Memory organization
-Process Scheduling

FIB

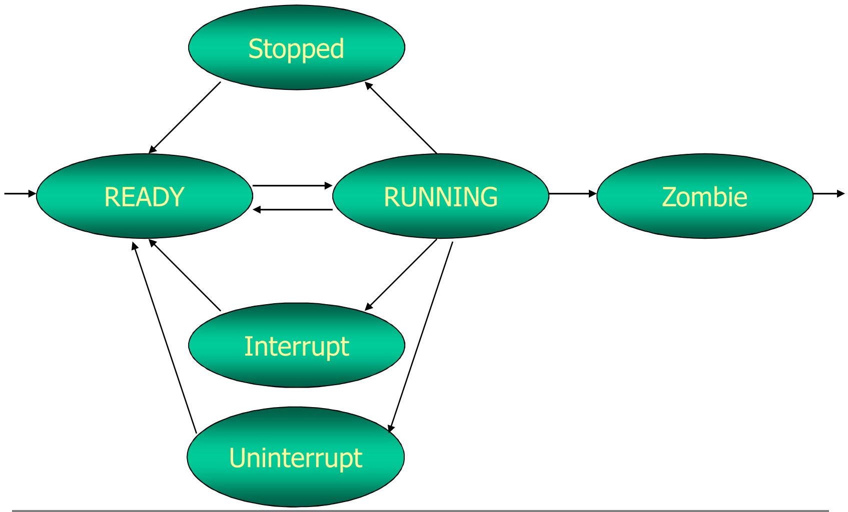
Summary

- Concepts
- Kernel data structures: process management
- Init process
- Memory organization
- System calls
- Scheduling
  - Context switch
  - Policies
- Process monitoring
- Process synchronization

Concepts

- Process
  - "instance of a program in execution"
    - Understanding the Linux kernel. Pag. 72
  - Resources are allocated to processes
  - The process has a life cycle
  - Processes are grouped in lists
    - Running, Waiting for I/O, Etc
  - Hierarchical relationship
    - 1 father → N children

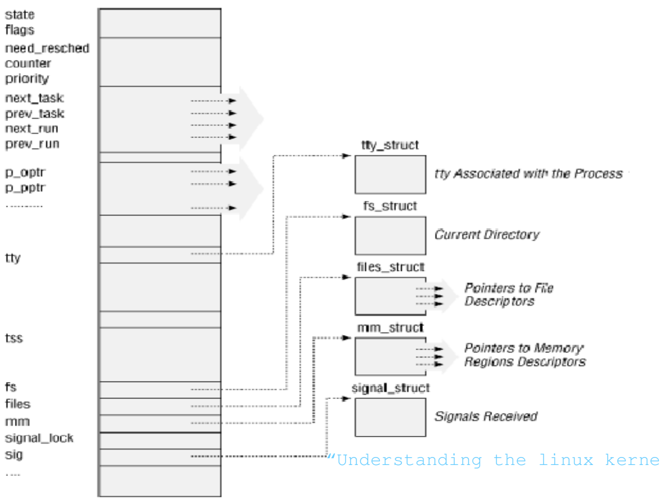
Concepts: Process life cycle



Concepts: process execution context

- The process execution context is:
  - The process address space : code,data,stack
  - The hardware context (registers value)
  - TSS (Task Segment Selector)
  - Kernel stack
- The hardware context is shared (1 set of registers). We must SAVE and RESTORE them
  - When changing the execution mode user←→kernel
  - When performing a context switch procA ←→procB

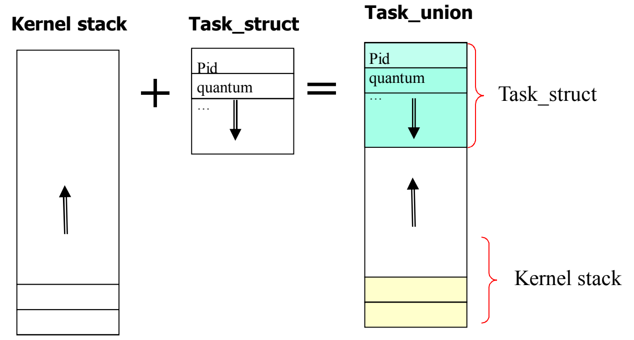
Kernel data structures: task_struct



"Understanding the linux kernel" Chapter 3

Kernel data structures: task_union

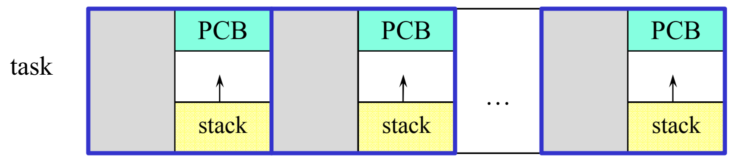
```
union task_union {
        struct task_struct task;
        unsigned long stack[1024];};
```



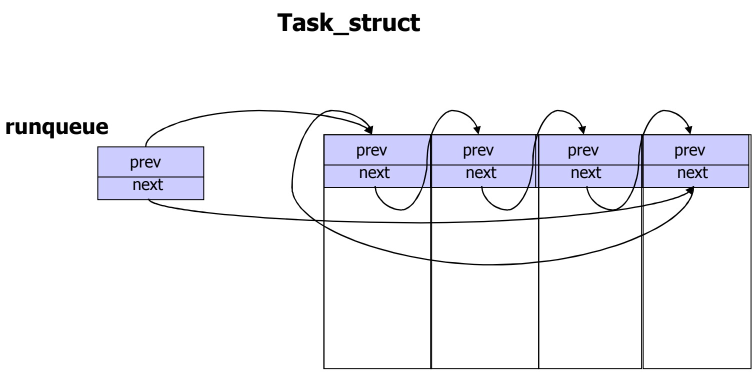
Kernel data structures: Task's array

- Array 'task' contains all tasks.
- To detect stack overflows each task is protected by a special page.

```
struct protected_task_struct {
  unsigned long task_protect [KERNEL_STACK_SIZE];
  union task_union t;
};
struct protected_task_struct task [NR_TASKS];
```



Kernel data structures: lists



FIB 71 UPC

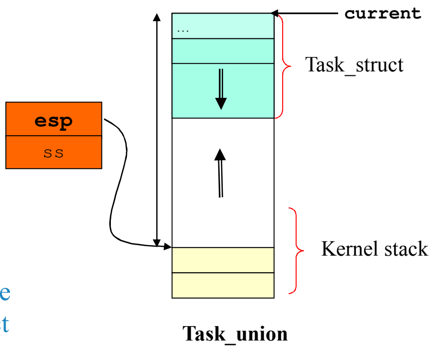
Kernel data structures

- How can we access the pointer to the running task_struct when entering the kernel?

The kernel stack and the task_struct **share** the same memory page (4k)

We know the kernel stack pointer of the running process (**esp**)

Based on the esp (applying a mask) we can obtain the address of the task_struct of the running process(current)



FIB 72 UPC

Kernel data structures

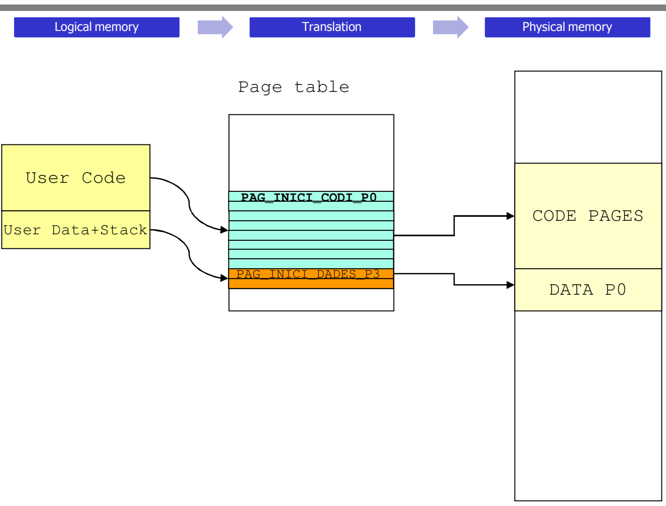
- Main work to do:
  - Complete type definitions
    - task_struct
  - Implement Kernel data needed
    - Task_struct array
    - Runqueue list

Init process

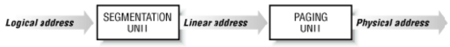
- Previously implemented
  - System/user Address space (see mm.c)
  - User/system stack allocated
  - TSS initialization (See setTSS function)
- You must:
  - Initialize task_struct for init process
    - PID=0
    - User stack
    - system stack
  - Insert it in runqueue list

Memory organization

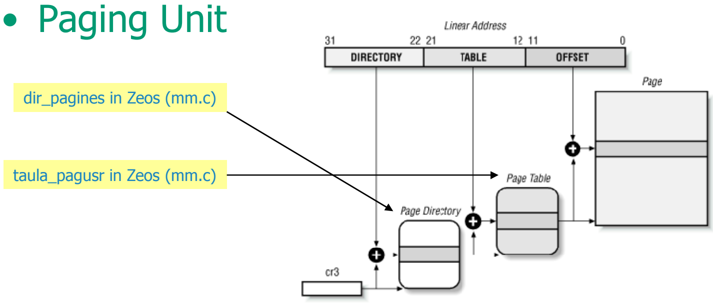
From logical to Physical addresses



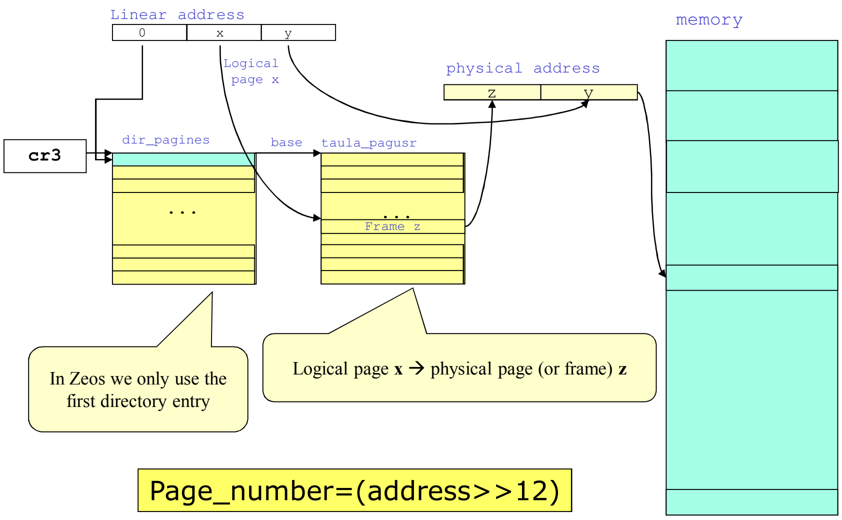
Segmentation+Pagination



- Segmentation
  - Initialized in Zeos and fixed → no modifications are required (see documentation P1)
- Paging Unit



Organization in Zeos

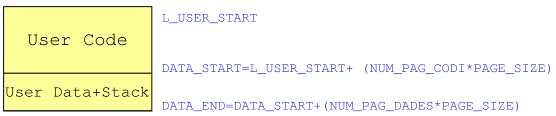


Data to manage memory(mm.c)

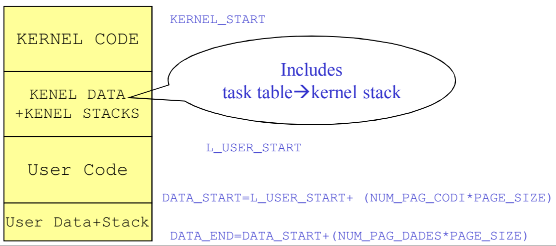
```
/* SEGMENTATION */
/* Memory segements description table */
Descriptor  *gdt = (Descriptor *) GDT_START;
/* Register pointing to the memory segments table */
Register    gdtR;

/* PAGING */
/* Variables containing the page directory and the page table */

page_table_entry dir_pages[TOTAL_PAGES]
  __attribute__((__section__(".data.task")));

page_table_entry pagusr_table[TOTAL_PAGES]
  __attribute__((__section__(".data.task")));

/* TSS */
TSS         tss;
```


Zeos Logical address space: consecutive

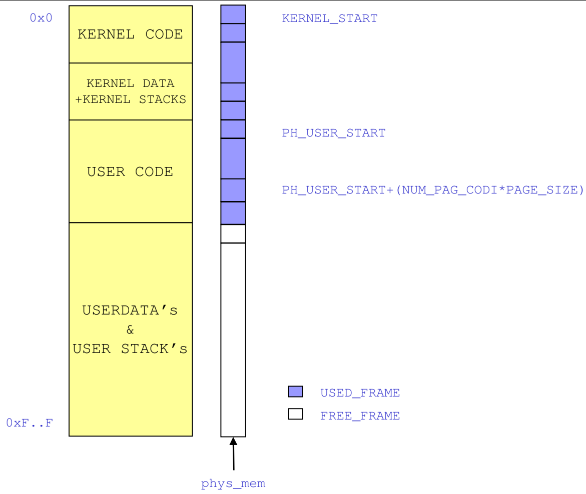
- User mode



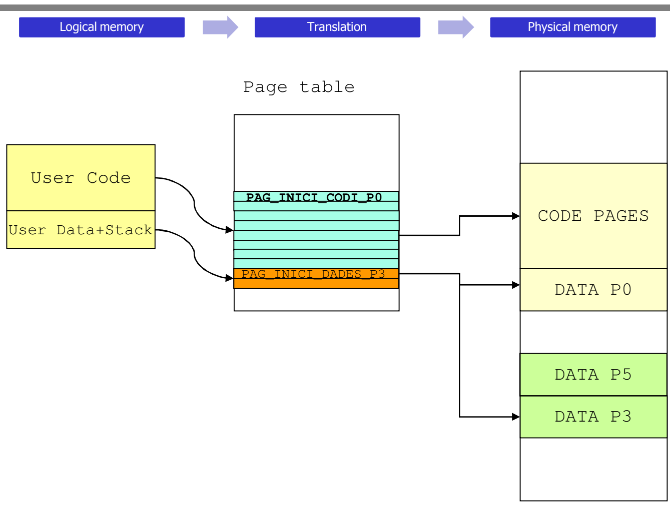
- Kernel mode



ZeOS Physical Memory Layout



Changing address space:P0→P3



System calls

- getpid→ returns process ID (PID)
- fork→ creates a child process
- nice→ modifies the process quantum
- exit→ finalizes a process
- get_stats→ returns monitoring info.

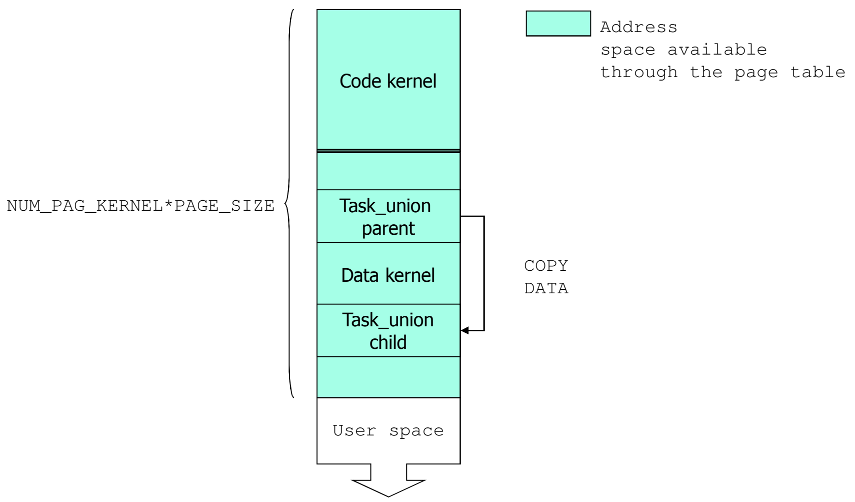
Fork: main steps

- Get free entry in task table
- Inherit system data
- Inherit user data
- Assign new PID (unique!)
- Update task_struct data
    - Includes I/O data
- Insert process in runqueue list
- Return pid of child process

Fork: Inherit system data

- 2-minutes to think about:
  1. Do we have access to the data?
  2. Are the data shared by both processes or not?
- System data: We have access to all the kernel data → no actions are required
  - Code → shared → no actions
  - Data → shared → no actions
  - Task_union (stack+task_struct) → **private**→ we must copy from parent to child process

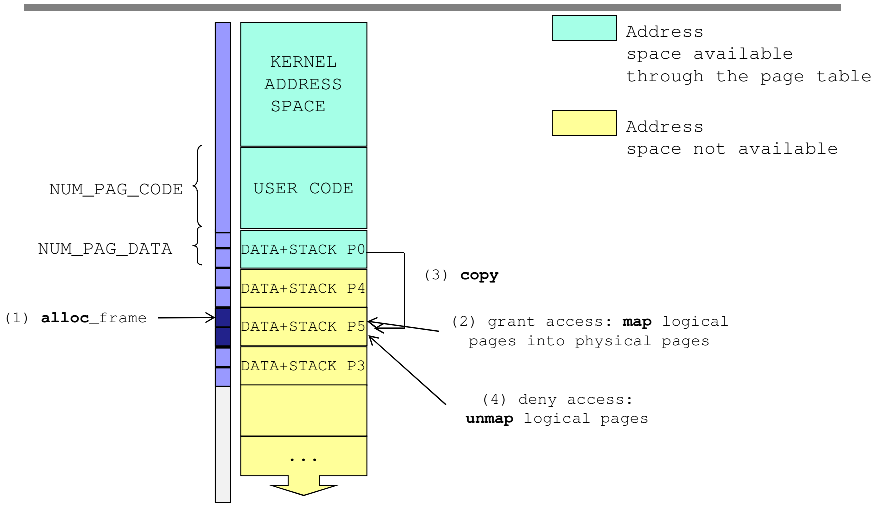
System data



Fork: Inherit user data

- The user address space is private
- If we have to copy some parts we have to modify the address space of the father
- **Code:** shared
  - no actions are required
- **Data+stack: private**
  - Allocate free frames for child data+stack
  - Grant access from parent to the child frames
    - **modify the memory page table**
  - Copy data+stack from parent to child
  - Deny access from parent to the child frames

User address space: Ex. P0 creates P5



Exit

- Update task_struct data
- Free allocated structures
  - Deallocate data+stack frames
  - Sempahores
  - …
- Delete process for the runqueue list
- Free entry of tasks table
- *Schedule*

Page fault management

- An example of exceptions management
  - Kill the process that causes the exception
    - Similar to the exit system call
  - Show a message in the screen
  - Continue with the execution of the rest of the processes

Process Scheduling

- The policy:
  - Evaluates if a change must be performed
  - And selects the *next* process
- The kernel performs a *context switch*
- Goals:
  - Fast response time
  - Good throughput
  - Prevent process starvation
  - Priority management
  - etc.

Process Scheduling: Round Robin

- Each process receives a (per process) unit of execution: One quantum= N tics
- When the quantum expires or process blocks
  - New process is selected for execution
  - First process of runqueue is always selected
    - Before selection, current process must be dequeued and enqueued
- When a process is selected to run, a whole quantum is assigned to it
- You must:
  - Define & implement required scheduler functions
  - Define & implement required round robin functions

Context switch

```
Scheduling()
{
        Task next;
        UpdateSchedulingData();
        if (MustChangeProcess()){
                Next=SelectNextProcess();
                Context_switch(next);
        }
}
Context_switch(P)
{
        SAVE(CURRENT)
        RESTORE(P)
}
```


Context switch

- The kernel suspends the execution of the running process and resume the execution of some "previously" suspended process
  - SAVE the execution context of the running process and RESTORE the execution context of the suspended one
    - Address space
    - TSS
    - Kernel stack
    - Hardware context

Context

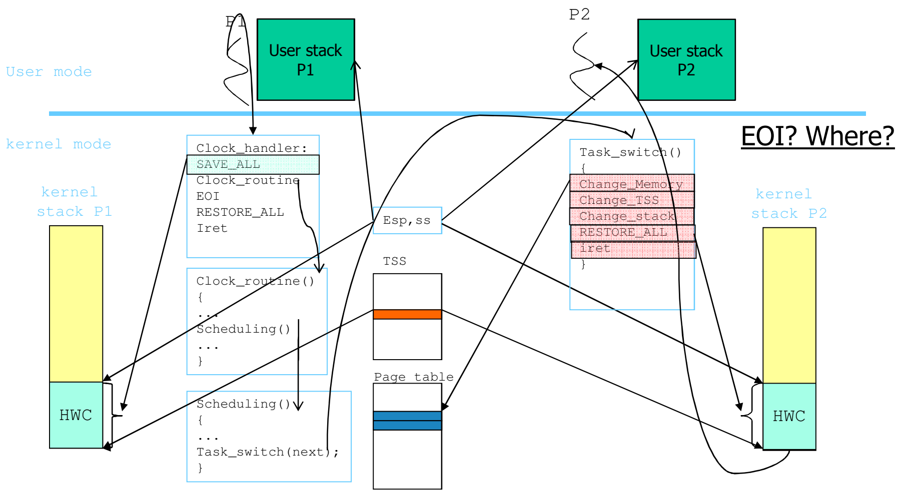
Context switch:SAVE

- Where/which is the context of current process saved?
  - Address space
    - Content: private, no need to save it
    - However, the info must be stored in task_struct
  - TSS (kernel stack address)
    - replaced each time, no need to save it
    - Address known, no need to save it
  - Kernel stack
    - Content:private, no need to save it
    - Address known, no need to save it
  - Hardware context
    - **It is saved in the kernel stack when entering the kernel (SAVE_ALL macro)**
    - **Where? Fixed position in kernel stack**

Context switch:RESTORE

- Resuming the context of *next* process
  - Address space
    - User (data+stack) pages
    - System has to move to the required page table entries
  - TSS
    - It must point to the kernel stack of *next* process (esp0)
  - Hardware context
    - esp registers → must point to the saved context of *next* process
    - The rest of registers are restored from the *next* kernel stack

Process context switch:example



Process monitoring

- get_stats→ returns per process monitoring info.

struct stats {
int total_tics; /* Total tics executed by the process */
int total_trans; /* Total transitions ready → run */
int remaining_tics; /* Remaining tics to end the quantum */
};

int get_stats(int pid, struct stat *st)

Semaphores

FIB

Process Synchronization

- Semaphores
  - Creation: **sem_init**
  - Synchronization: **sem_wait** and **sem_signal**
  - Destruction: **sem_destroy**
  - 25-30 semaphores

- The use of semaphores can block and unblock processes (NEW!)

FIB

Creation

- int sem_init(int n_sem, unsigned int value)
  - n_sem: semaphore identificator
  - value: initial value for semaphore's counter
  - Returns -1 if error or 0 if ok
  - System call Id is 21
  - Initializes for semaphore 'n_sem':
    - counter to 'value'
    - blocked processes queue
    - Process that initializes the semaphore becomes the owner

sem_wait()

- int sem_wait(int n_sem)
  - n_sem: semaphore id
  - Returns: -1 if error or 0 if ok
  - System call Id is 22
  - If counter of semaphore 'n_sem' <= 0 →
    - Block current process
  - Else
    - Decrement counter

sem_signal()

- int sem_signal(int n_sem)
  - n_sem: semaphore id
  - Returns: -1 if error or 0 if ok
  - System call Id is 23
  - If no blocked process at semaphore 'n_sem' →
    - Increment counter
  - Else
    - Unblock First blocked process

Destruction

- int sem_destroy(int n_sem)
  - n_sem: semaphore id
  - Returns: -1 if error or 0 if ok
  - Unblocks blocked processors (sem_wait will return -1) and releases the semaphore
  - System call Id is 24
  - Errors:
    - semaphore not initialized
    - Calling process not the owner

Deliverable P1.3
Input/Output management

-Logical devices
-Virtual devices
- zeosFAT File system

FIB

Delivery 1.3 overview

- Processes needs to have access to/from devices
- OS features
  – Hides physical characteristics
  – Provides new "logical" devices
  – Manages devices efficiently, securely
  – Offers virtual devices to processes

FIB

Delivery 1.3 overview

- Some information must be persistent
  - Stored in disk
- File systems organize and manage data stored in disks
  - zeosFAT: File system based on FAT
- You have to:
  - Design new data structures
    - Logical/virtual devices
  - Implement system calls for input/output
  - Design/Implement zeosFAT

FIB 107 UPC

Devices

- Physical: Hardware devices
- Logical
  - known by the O.S
    - OS internal data representation + Device dependent functions
  - Abstract zero, one, or multiple physical devices
- Virtual (file descriptors)
  - Known by processes
  - Managed through system calls

FIB 108 UPC

Physical devices

- Keyboard
  - Read only
- Display
  - Write only

FIB

UPC

Logical devices

- Represents zero, one or multiple devices
  - Zero: ex. Pipes
  - One: keyboard
  - Multiple: console (keyboard+display)
- The OS defines a set of functions to make uniform the device access
  - New data type: file_operations
  - Each device can have its own data
- Files will represent logical devices

FIB

UPC

Logical devices: file_operations

- Define the common set of functions to access to devices
  - Open
  - Read
  - Write
  - Etc
- Same API to all the devices
  - The set of functions and parameters of each one are a superset

Logical devices: files

- Static characteristics
  - Name (to simplify, no multiples names for one logic device is allowed)
  - Access mode allowed: read, write, read&write
  - File_operations
- Dynamic characteristics
  - Depends on the specific accesses in course
    - Current offset
    - Open mode
    - …

Logical devices: files

- Define the required OS structures to guarantee (see system calls semantics):
  - Two processes accessing concurrently the same file
    - Concurrent but independent
  - Two processes sharing the access to the same file
    - Sharing dynamic data
  - One process accessing concurrently the same file multiple times
  - One process sharing the access to the same file multiple times

Logical devices: zeos name space

- Zeos specific: not persistent, only in memory
- Single level of directory
  - Not sub-directories are allowed
- Design the directory data type and the required functionality to initialize it at the start of zeos.

Virtual devices

- Each time a process ask for access of a file (open's it), the OS returns a new per-process handler (file descriptor)
- The handler binds process, the logical device, and the dynamic characteristics associated with the open
- Default fd
  - stdin : fd=0
  - stdout: fd=1
  - stderr: fd=2

Virtual devices

- Each process has its own table of file descriptors
- When a process calls forks, the child process heritages the file descriptor table (a copy)
  - fd's of child process will SHARE dynamic information with the parent process

New System calls

- Open
- Read
- Dup
- Close

Syscall open

- int open(char *path, int mode)
  - path: name of the file to be opened.
  - mode can be:
    - O_RDONLY (0x1)
    - O_WRONLY (0x2)
    - O_RDWR (0x3)
  - Returns: -1 if error or a new fd if ok
  - System call Id is 5
  - <u>Creates an open file descriptor</u> that refers to the file and establishes a connection between it and the returned fd.

Syscall read

- int read(int fd, char *buff, int size)
  - fd: file descriptor to read from
  - buff: pointer where data will be copied to
  - size: size (in bytes) to read
  - Return: -1 if error or number of read bytes if OK
  - System call Id is 3

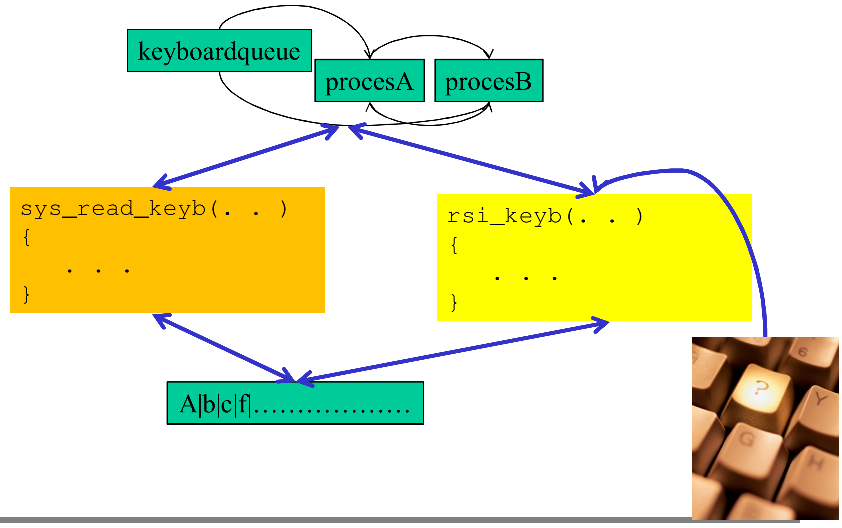
Note that:

- Buff must be in the user address space
- Read's are served in arrival order
- Calls the device dependent function

Syscall: dup/close

- int dup (int fd)
  - fd: file descriptor to duplicate
  - Return: -1 if error or the first fd available if OK
  - System call Id is 41
  - Duplicate an open file descriptor. The new fd SHARES the dynamic information
- int close (int fd)
  - fd: file descriptor to close
  - Return: -1 if error or 0 if OK
  - System call Id is 6
  - Unbinds file descriptor and its logical device

Reading from the keyboard



Sys_read_keyboard

- Shared buffer between this function and the keyboard rsi
  - Circular buffer (new type)
- If there are pending requests
  - BLOCK(keyboardqueue) the calling process
- Otherwise
  - If there are enough bytes to read → read and return
  - Otherwise→ BLOCK(keyboardqueue) the process
- Before blocking, we need to save the "request" data (which info?)

Rsi Keyboard

- Stores new characters in the shared buffer
- If there are blocked processes in the keyboardqueue
  - If there are enough bytes to finish the request OR the buffer is full
    - Copy data →Be careful, process A, executing the rsi, is copying data to user space of process B

ZeosFAT

File System

- Defines how files are stored in disk and how the disk space is managed
  - Allocates blocks
  - Organize blocks
  - Etc
- Per file system you have to decide
  - How to allocate new blocks
  - How to manage free blocks

zeosFAT

- In memory file system, we are not going to access to disk (not persistent)
- We will substitute the disk by a vectors of data blocks of 256 bytes
  - New data type
  - Design it and think about functionallity

How to allocate disk space?

- The unit of allocation will be blocks
- Blocks in a file will be linked
  - Each block will point to the next of the same file
- Pointers to next block will be separated from data
  - New data type: File Allocation Table (FAT)
- We need to know the first block of data per file

How to manage free blocks?

- Using the same FAT
  - Free blocks are linked
  - We need a pointer to the first free block

Directory

- Single level of directory

Existing System calls

- open
  - Add new flags to create a file:
    - O_CREAT (0x4)
    - O_EXCL (0x8)
- write/read/dup/close
  - Should remain without modifications

System call to add

- int unlink(const char *path)
  - Path: name of the file to be removed
  - Return: -1 if error (not abled to remove) or 0 if OK
  - System call Id is 10
  - Remove a file. A file only can be removed if no processes are using it: you should maintain the number of active references

zeosFAT

- Design the new data types