ProSO

Operating Systems Project (PROSO)
Barcelona School of Informatics (FIB)
Universitat Politècnica de Catalunya (UPC)
2012-2013

Goals

- Get a good O.S. internal understanding
  1. Be able to program low level O.S. code, developing from scratch the basic system components
  2. 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: tuesdays)
    – Additional, on your own (5h/week)

• Intermediate code deliveries & meetings with your advisor to follow the project evolution

• Teams of two students
  – Use the forum to put your group (deadline: 17/09)
  – Name; DNI; Lab group to assist
Grading

• Project 0 → Just to set up the environment
• Project 1 (ZeOS) → 70%
  – Intermediate delivery P1.1 → 10%
  – Intermediate delivery P1.2 → 25%
  – Final Project delivery & Global evaluation P1.3 → (25% + 10%)
• Project 2 (Linux modules) → 30%
  – Final Project delivery → 30%

Deliveries

• Deliveries are not mandatory
  – But the corresponding grade will be 0
• Intermediate deliveries
  – Evaluation & Feedback
    • You must meet with your advisor
• Important dates
  – P1.1: September 28th (Friday)
  – P1.2: October 31st (Wednesday)
  – P1.3: November 30th (Friday)
  – P2: December 21st (Friday)
Initial scheduling (can be modified)

<table>
<thead>
<tr>
<th>Monday</th>
<th>Tuesday</th>
<th>Wednesday</th>
<th>Thursday</th>
<th>Friday</th>
</tr>
</thead>
<tbody>
<tr>
<td>10 September</td>
<td>LAB</td>
<td>LAB</td>
<td>LAB</td>
<td>18 September</td>
</tr>
<tr>
<td>17 September</td>
<td>LAB</td>
<td>LAB</td>
<td>LAB</td>
<td>25 September</td>
</tr>
<tr>
<td>24 September</td>
<td>LAB</td>
<td>LAB</td>
<td>LAB</td>
<td>29 September</td>
</tr>
<tr>
<td>01 October</td>
<td>LAB</td>
<td>LAB</td>
<td>LAB</td>
<td>06 October</td>
</tr>
<tr>
<td>08 October</td>
<td>LAB</td>
<td>LAB</td>
<td>LAB</td>
<td>13 October</td>
</tr>
<tr>
<td>15 October</td>
<td>LAB</td>
<td>LAB</td>
<td>LAB</td>
<td>20 October</td>
</tr>
<tr>
<td>22 October</td>
<td>LAB</td>
<td>LAB</td>
<td>LAB</td>
<td>27 October</td>
</tr>
<tr>
<td>29 October</td>
<td>LAB</td>
<td>LAB</td>
<td>LAB</td>
<td>03 November</td>
</tr>
<tr>
<td>05 November</td>
<td>LAB</td>
<td>LAB</td>
<td>LAB</td>
<td>10 November</td>
</tr>
<tr>
<td>12 November</td>
<td>LAB</td>
<td>LAB</td>
<td>LAB</td>
<td>17 November</td>
</tr>
<tr>
<td>19 November</td>
<td>LAB</td>
<td>LAB</td>
<td>LAB</td>
<td>24 November</td>
</tr>
<tr>
<td>26 November</td>
<td>LAB</td>
<td>LAB</td>
<td>LAB</td>
<td>01 December</td>
</tr>
<tr>
<td>09 December</td>
<td>LAB</td>
<td>LAB</td>
<td>LAB</td>
<td>18 December</td>
</tr>
<tr>
<td>16 December</td>
<td>LAB</td>
<td>LAB</td>
<td>LAB</td>
<td>25 December</td>
</tr>
</tbody>
</table>

Material

- [http://studies.ac.upc.es/FIB/PROSO](http://studies.ac.upc.es/FIB/PROSO)
- **RACO**
  - Deliveries
  - Forum
  - Noticeboard
- **Main source of documentation**
  - PROSO web page
  - P1 documentation (in web page)
  - P1 basic files and tests (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)
What are you going to implement?

- Implement a minimal O.S. kernel (ZeOS) Linux based

1. Mechanisms for entering the system (P1.1)
   - Interruptions, exceptions, system calls

2. Process Management (P1.2)

3. 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 *)

* See book: “Understanding The Linux Kernel”
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, ...
- Basic test libraries → see web page
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)

How to work

- Execute options
  a) Boot ZeOS from a floppy
     • Slowdown the development process
  b) Boot ZeOS on a machine emulator: Bochs
Tools

- Makefile (DONE in SO courses)
- nm
  - To see where functions/data are mapped
  - Read man pages
- objdump
  - To see code memory addresses
  - Read man pages
- Bochs emulator \(\rightarrow\) see the documentation

Bochs emulator (v2.3)

Boot Process: General Overview

- **Bootstrap**: initialize 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

ZeOS image 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 does not have a program loader
ZeOS image file scheme

12 bytes reserved

512 bytes System area
User area

128 bytes reserved

system.c
entry.S
io.c
sys.c
device.c
sched.c
mm.c
libc.c
user.c

main() (in system.c)

- Initializes a minimum execution context
  - Prepares init process environment
  - Initializes Segment Registers (memory management)
  - Initializes init process kernel stack
  - Sets gdtr and idtr from GDT and IDT address
- Initializes IDT (set_idt)
  - There are not any defined interrupts
- Copy user/code data to final destination in memory
- Jumps to user space (main() function in user.c)
P0: Preliminary assignment

- 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

P0: TO DO

- Understand the basic ZeOS files
- User code modifications
  - ONLY to get used to the working environment
  - NOT part of P1
- System code modifications
  - Modify printk to consider ‘\n’ characters
  - Include some additional messages when booting
  - Output in the Ubuntu console (DONE)
Deliverable P1.1
Mechanisms for entering the system

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

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 **if needed**
  - See enable_int function
  - Only for masked interrupts
Overview: \textit{management code}

- Entry point → handler (entry.S)
  - Assembly code
  - Basic \textit{hardware context} management

- Service routine (depends on the interrupt)
  - C code
  - Specific algorithm

Example: clock interrupt behavior

User mode (user.c, libc.c, etc)

```
... ...
```

kernel mode

```
clock_handler:
call clock_routine
iret
```

```
clock_routine()
/* clock interrupt code */
```

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 exiting the system

- Restore hardware context
  - general purpose registers (RESTORE_ALL macro)
  - ss, esp, psw, cs & eip (iret instruction)

- Switch execution mode
  - Kernel mode → User mode (iret instruction)
Exceptions

Exceptions list

<table>
<thead>
<tr>
<th>0. Divide error</th>
<th>10. Invalid TSS*</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Debug</td>
<td>11. Segment not present*</td>
</tr>
<tr>
<td>2. Not used</td>
<td>12. Stack segment*</td>
</tr>
<tr>
<td>3. Breakpoint</td>
<td>13. General protection*</td>
</tr>
<tr>
<td>4. Overflow</td>
<td>14. Page fault*</td>
</tr>
<tr>
<td>5. Bounds check</td>
<td>15. Reserved</td>
</tr>
<tr>
<td>6. Invalid opcode</td>
<td>16. Floating point error</td>
</tr>
<tr>
<td>7. Device not available</td>
<td>17. Alignment check*</td>
</tr>
<tr>
<td>8. Double fault*</td>
<td>18 to 31. Reserved</td>
</tr>
<tr>
<td>9. Coprocessor segment overrun</td>
<td>* hardware error code (4 bytes)</td>
</tr>
</tbody>
</table>
**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**

User mode

- ... exception generated while executing application program ...
- gpe_handler:
  - ... call gpe_routine ...
  - iret

Kernel mode

- gpe_routine() {
  - ...
  - ...
}
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.
    setInterruptHandler(0,divide_error_handler ,KERNEL_LEVEL)

DEFINE KERNEL_LEVEL & USER_LEVEL CONSTANTS

Handling exceptions: handler

- Save hardware context (SAVE_ALL)
- call exception service routine
- Restore hardware context (RESTORE_ALL)
  - Remove (if any) the exception hardware error code from the kernel stack
  - Return to user (iret)

YOU HAVE TO IMPLEMENT IT
Handling exceptions: service routine

- Exceptions not resolved in delivery 1

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

Interrupts

- clock
- keyboard
Handling interrupts: overview

User mode

... ...

Kernel mode

interrupt arrives while executing application program

clock_handler:

... call clock_routine

... iret

clock_routine()

Handling interrupts: initialization

- Interrupts initialization
  - Init IDT entry
    - 32: clock,
    - 33: keyboard
  - Enable interrupts: 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
      make(0) / break(1)
      ```
  - 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

User mode

```
... xyz() ...
```

Wrapper routine in libc library

```
xyz() {
    ... int 0x80 ...
}
```

System call invocation in application program

Kernel mode

```
system_call:
    ... call sys_xyz ...
    ... iret

sys_xyz() {
    ... if error ret =ERR ...
}
```

System call handler

System call service routine

New for system calls
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

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 inline 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
- Steps to invoke the system call handler
  a) Pass parameters
  b) Identify the system call
  c) Generate the trap
- Return the result to the user code

Handling system calls: wrapper (II)

a) 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
b) Identify the system call
   - Copy system call number to eax
c) Generate Trap: int $0x80
d) Return result to user code
   - If error: returns -1 and updates the libc global **errno** variable
Handling system calls: handler

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

Handling system calls: service routines

- System call service routine
  - Check parameters
    - If error, returns the appropriate negative error code
  - 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
      - sys_write_console

PARAMETERS (which, where)?
TO DO

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

Deliverable P1.2
Process management
- Kernel data structures
- Memory organization
- Process scheduling
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
    - Ready, Running, Stopped, Zombie, etc.
  - Processes are grouped in lists
    - Running, Waiting for I/O, etc.
  - Hierarchical relationship
    - 1 father → N children
Concepts: process execution context

- The process execution context is:
  - The process address space: code, data, stack
  - The hardware context (values of the registers)
  - TSS (Task Segment Selector)
  - Kernel stack

- The hardware context is shared (only 1 set of registers)
  - We must SAVE and RESTORE them
    - When changing the execution mode: user $\rightarrow$ kernel
    - When performing a context switch: procA $\leftrightarrow$ procB

Kernel data structures: task_struct

"Understanding the Linux kernel" Chapter 3
Kernel data structures: **task_union**

```c
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.

```c
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

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)
Kernel data structures

- Main work to do:
  - Complete type definitions
    - task_struct
  - Implement kernel data needed
    - runqueue list
  - Implement ‘current’ function

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
    - Information about memory
    - ...
  - 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**
  - `dir_pages` in ZeOS (`mm.c`)
  - `pagusr_table` in ZeOS (`mm.c`)

---

**Organization in ZeOS**

- In ZeOS we only use the first directory entry
- Logical page $x$ → physical page (or frame) $z$
- Page number = $(address >>> 12)$
ZeOS logical address space: consecutive

- User mode
  - User Code
    - User Code: $L_{USER\_START}$
    - User Data + Stack: $DATA_{START} = L_{USER\_START} + (NUM\_PAG\_CODE \times PAGE\_SIZE)$
    - User Data + Stack: $DATA_{END} = DATA_{START} + (NUM\_PAG\_DATA \times PAGE\_SIZE)$

- Kernel mode
  - Kernel Code
    - Kernel Code: $KERNEL\_START$
    - Kernel Code includes:
      - Task Table
      - Kernel Stack
    - Kernel Code: $PH_{USER\_START}$
    - Kernel Code: $PH_{USER\_START} + (NUM\_PAG\_CODE \times PAGE\_SIZE)$

ZeOS physical memory layout

- User Code
  - User Code: $USER\_CODE$
  - User Data + Stack: $USERDATA\_s$ & $USER\_STACK\_s$
  - User Data + Stack: $USERDATA\_s + (NUM\_PAG\_DATA \times PAGE\_SIZE)$

- Kernel Code
  - Kernel Code: $KERNEL\_START$
  - Kernel Code: $PH_{USER\_START}$
  - Kernel Code: $PH_{USER\_START} + (NUM\_PAG\_CODE \times PAGE\_SIZE)$
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**

1. Do we have access to the data?
   - We have access to all the kernel data → no actions are required

2. Are the data shared by both processes?
   - Code → shared → no actions
   - Data → shared → no actions
   - Task_union (stack+task_struct) → private → we must copy from parent to child process
Fork: Inherit system data

- Code kernel
- Task_union parent
- Data kernel
- Task_union child
- User space

Address space available through the page table

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**
    1) Allocate free frames for child data+stack
    2) Grant access from parent to the child frames
      - Modify the page table
    3) Copy data+stack from parent to child
    4) Deny access from parent to the child frames
User address space: Ex. P0 creates P5

Exit

- Free allocated structures
  - Deallocate data+stack frames
  - Semaphores
  - ...
- 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 to the next process
- Goals:
  - Fast response time
  - Good throughput
  - Prevent process starvation
  - Priority management
Process Scheduling: Round Robin

- Each process receives a (per process) unit of execution: One quantum = N clock 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

```c
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 resumes 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 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 set the required page table entries
  - TSS
    - esp0 must point to the kernel stack of **next** process
  - Hardware context
    - esp register → must point to the saved context of **next** process (in its kernel stack)
    - The rest of registers are restored from the **next** kernel stack

Process context switch: example

![Diagram showing process context switch example]
Process monitoring

- `get_stats` returns per process monitoring info

```c
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; 0 if process is not running */
};

int get_stats(int pid, struct stat *st);
```

Note that 'st' is a pointer to the user's address space

---

Semaphores
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!)

---

**Creation**

- `int sem_init (int n_sem, unsigned int value)`
  - n_sem: semaphore identifier
  - 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’
    - List of blocked processes
    - 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 and schedule
  - Else
    - Decrease 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’ →
    - Increase 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 processes (respective `sem_wait` will return -1) and releases the semaphore
    - System call Id is 24
    - Errors:
      - Semaphore not initialized
      - Calling process is not the owner

---

**Deliverable P1.3**

**Input/Output management**

- Logical devices
- Virtual devices
- ZeOSFAT File system
Delivery 1.3 overview

- Processes need to have access to/from devices
- OS features
  - Hides physical characteristics
  - Provides new “logical” devices
  - Manages devices efficiently, securely
  - Offers virtual devices to processes

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
## Devices

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

---

## Physical devices

- **Keyboard**
  - Read only
- **Display**
  - Write only
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

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 are allowed)
  - Access mode allowed: read, write, read&write
  - file_operations
  - No users accounts: no permission information is needed

- **Dynamic characteristics**
  - Depend on the specific accesses in course
    - Current offset
    - Open mode
    - ...

Logical devices: files

- Define the required OS structures to guarantee:
  - 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
  - No 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 asks for access of a file (opens it), the OS returns a new **per-process handler (file descriptor)**
- The fd binds process, the logical device, and the dynamic characteristics associated with the open
- Default fds
  - stdin: fd=0
  - stdout: fd=1
  - stderr: fd=2
Virtual devices

- Each process has its own table of file descriptors
- When a process calls fork, 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
  - Creates a file descriptor that refers to the file and establishes a connection between the file 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** a 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
  - Unbind file descriptor and its logical device

---

**Reading from the keyboard**

```
sys_read_keyb(. . )
{
  ....
}

keyb_routine(. . )
{
  ....
}
```

- `processA` and `processB` access the keyboard queue through the `sys_read_keyb` and `keyb_routine` functions, respectively.
sys_read_keyboard

- **Shared buffer** between this function and the keyboard service routine
  - 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?)

Keyboard service routine

- 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 to process → Be careful, process A, executing the service routine, is copying data to user space of process B → Page table must be updated
ZeOSFAT

File System

- Defines how files are stored in disk and how the disk space is managed
  - Allocate 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 vector of data blocks of 256 bytes
  - New data type
  - Design it and think about functionality
    - Having few blocks is enough (MAX_BLOCKS = 15)

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)
  - If we are creating the file, the access mode flag (O_RDONLY, O_WRONLY, O_RDWR) will indicate the valid access mode for that file during its whole lifetime.

- **write/read/dup/close**
  - Should remain without modifications

System call to be added

- **int unlink (const char *path)**
  - Path: name of the file to be removed
  - Return: -1 if error (not able 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
    - Removing a file includes freeing its data blocks