Creative Commons License

This work is under a Creative Commons Attribution 4.0 Unported License

The details of this license are publicly available at https://creativecommons.org/licenses/by-nc-nd/4.0
# Table of Contents

- Operating System
- Basic Concepts
- Access to Kernel functions
- Process management
- Memory subsystem
- Other important OS concepts and tasks
What is an Operating System?

- The OS is a software that manages hardware and software resources.
Management of a distributed environment

- Multiple nodes have usually one copy of the OS each
  - With a shared filesystem
Design Principles

• The OS also plays a key role for user interaction

• It offers a **usable** environment
  • Abstracts the user from the different kind of “systems” and hardware

• It offers a **safe/robust/protected** execution environment
  • Safe from the point of view of accessing HW correctly and protected from the point of view of user’s interaction

• It offers an **efficient** execution environment
  • Fine grain management of the HW
  • Allows many users/programs sharing resources, ensuring a good resource utilization
Dealing with the system

• Using a graphical environment / a command shell in text mode
  • Allow a high-level view of the system services and resources
    • Services
    • Storage
    • Networking
  • On top of the kernel system calls
System Management

• Intermediary between applications and hardware

  • **Kernel internals**: define data structures to manage HW and algorithms to decide how to use it

  • **Kernel API**: offers a set of functions to ask for **system services**
    • **System calls**
The System: Programs + Kernel

- User 1
- User 2
- Shell
- Editors
- Compilers
- Data Bases
- Web Browsers
- System calls
- Kernel
Overall main steps of the OS

Boot
- Executed when the system is switched on, the kernel code is loaded in memory
- Interrupts and basic HW configurations are initialized
- It starts the system access mechanism: daemons, login, shell, etc

Usage
- User sessions / execution of applications, services
- Support development of new applications...

Shutdown
- Executed when system is switched off
- Saves persistent information, stop services and devices, etc
Access to Kernel functions

• Execution modes
  • Hardware support to guarantee security
    • The CPU must be able to differentiate when it is executing instructions coming from normal (non-privileged) user code or instructions coming from the kernel code
  • Two or more levels of privilege
    • User vs. kernel modes
    • Privileged levels: 0 (most-privileged) – 1 – 2 – 3 (user-level)
When is the kernel code executed? (I of III)

• When an **interrupt** occurs:
  • Interrupts are generated by HW devices
    • Interrupt: asynchronous and involuntary notification
  • Clock interrupts: there are several clocks in a system
    • **OS Tick**: configured during boot period by the OS (e.g. 10ms) to execute management code
  • Storage/network...
Example of Interrupts

<table>
<thead>
<tr>
<th>IRQ</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>system timer (cannot be changed)</td>
</tr>
<tr>
<td>1</td>
<td>keyboard controller (cannot be changed)</td>
</tr>
<tr>
<td>2</td>
<td>cascaded signals from IRQs 8–15</td>
</tr>
<tr>
<td>3</td>
<td>second RS-232 serial port (COM2 in Windows)</td>
</tr>
<tr>
<td>4</td>
<td>first RS-232 serial port (COM1 in Windows)</td>
</tr>
<tr>
<td>5</td>
<td>parallel port 2 and 3 or sound card</td>
</tr>
<tr>
<td>6</td>
<td>floppy disk controller</td>
</tr>
<tr>
<td>7</td>
<td>first parallel port</td>
</tr>
<tr>
<td>8</td>
<td>real-time clock</td>
</tr>
<tr>
<td>9</td>
<td>open interrupt</td>
</tr>
<tr>
<td>10</td>
<td>open interrupt</td>
</tr>
<tr>
<td>11</td>
<td>open interrupt</td>
</tr>
<tr>
<td>12</td>
<td>PS/2 mouse</td>
</tr>
<tr>
<td>13</td>
<td>math coprocessor</td>
</tr>
<tr>
<td>14</td>
<td>primary ATA channel</td>
</tr>
<tr>
<td>15</td>
<td>secondary ATA channel</td>
</tr>
</tbody>
</table>
When is the kernel code executed? (II of III)

• When an **exception** occurs:
  • exceptions are generated by the CPU when some problem occurs during the execution of one instruction
    • Exception: synchronous, but involuntary notification

![Diagram showing exceptions and their causes]

- Floating point
  - Division by 0
  - Overflow
  - Underflow
  - Inexact
- Bad address (segmentation fault/bus error)
- Page fault
- Breakpoint
- Invalid instruction
- Alignment check
- ...
When the kernel code is executed? (III of III)

• When a program requests a service from the OS through a **system call**

- open, close, read, write, ioctl, fnctl, stat, fstat...
- mount, umount...
- fork, exec, exit, clone...
- socket, bind, listen, accept, connect...
- ...

OpenFile, ReadFile, WriteFile...
CreateProcess, CreateProcessEx...
...

... up to tens of syscalls
System Calls

• When a program requests a service from the OS through a **system call**
• A system call has stronger requirements than a “simple” function call
  • **Requirements**
    • The kernel code MUST be executed in privileged mode
    • For security, the “jump” implicit in the call instruction and the execution mode change must be done with a single instruction
    • The memory address of a system call could change from one kernel version to another, and it must be compatible → we need an instruction different from a “call”
  • **To take into account**
    • Changes in the execution mode imply that some HW resources are not shared
      • for instance, the stack
System Library

• To hide all these details to the user, the system provides a library to be linked with user codes
  • This is automatically done by the compile (gcc / g++ for instance)

• It is called the system library, and translates from the high level system call API for the specific language (C, C++, etc.) to the assembler code where all the requirements are taken into account
  • For C and C++, system calls are included in the C support library (libc)
# System Calls

- The implementation depends on the architecture and the OS itself

<table>
<thead>
<tr>
<th></th>
<th>Function call</th>
<th>System call</th>
</tr>
</thead>
<tbody>
<tr>
<td>Argument passing</td>
<td>stack / registers</td>
<td>stack / registers</td>
</tr>
<tr>
<td>Function invocation</td>
<td>call</td>
<td>syscall (depend)</td>
</tr>
<tr>
<td>At function start</td>
<td>save registers (sw)</td>
<td>save registers (sw)</td>
</tr>
<tr>
<td>Accessing arguments</td>
<td>stack</td>
<td>stack / registers</td>
</tr>
<tr>
<td>Before return</td>
<td>restore registers (lw)</td>
<td>restore registers (lw)</td>
</tr>
<tr>
<td>Return values</td>
<td>registers</td>
<td>stack / registers</td>
</tr>
<tr>
<td>Return function</td>
<td>ret</td>
<td>sysexit (depend)</td>
</tr>
</tbody>
</table>
Example: C/C++ code (from Comp.&Libs Lesson)

- From high-level code to program execution

```c
#include <unistd.h>
#include <stdio.h>

int global = 4;

int main(int argc, char **argv){
    char buf[512];
    int num;

    num = sprintf(buf, "Hello World %d!\n", global);
    write(1, buf, num);

    return 0;
}
```

- **Sprintf** is a language library call
- **Write** is a system library call
Non-privileged vs privileged execution mode
Summary: Step-by-Step kernel code invocation

- Save user context

→ Change from user mode to kernel mode
  - Restore system context
  - Retrieve user parameters
  - Identify service
  - Execute service
  - Return result

← Change from kernel mode to user mode
  - Restore user context

This procedure involves an overhead.
Linux i386 (32bits) vs Linux x64 (64bits)

**Linux i386**

...  
movl $4, %eax ; use the write syscall  
movl $1, %ebx ; write to stdout  
movl $msg, %ecx ; use string "Hello World!\n"  
movl $14, %edx ; write 14 characters  
int $0x80 ; make syscall  
...  

**Linux x64**

...  
movq $1, %rax ; use the write syscall  
movq $1, %rdi ; write to stdout  
movq $msg, %rsi ; use string "Hello World!\n"  
movq $14, %rdx ; write 14 characters  
syscall ; make syscall  
...
Table of Contents

• Operating System
• Basic Concepts
• Access to Kernel functions
• Process management
• Memory subsystem
• Other important OS concepts and tasks
Program vs Process

- A process is the OS representation of a program during its execution
Program vs Process

• A process is the OS representation of a program during its execution

• The user **program** is static: it is just a sequence of bytes stored in a “disk”

• The user **process** is dynamic, and it consists of...
  • What regions of physical memory is using
  • What files is accessing
  • Which user is executing it (owner, group)
  • What time it was started
  • How much CPU time it has consumed
  • ...
Processes

• Assuming a general purpose system, multi-user
  • each time a user starts a program, a new (unique) process is created
  • The kernel assigns resources to it: physical memory, some slot of CPU time and allows file accesses

• In a general purpose system, we have a multiprogrammed environment
  • Multiprogrammed System: a system with multiple programs running at a time

• Process creation
  • The kernel reserves and initializes a new process data structure with dynamic information (the number of total processes is limited)
    • Each OS uses a name for that data structure, in general, we will refer to it as PCB (Process Control Block)
    • Each new process has a unique identifier (in Linux it is a number). It is called PID (Process Identifier)
Process Control Block (PCB)

- The PCB holds the information the system needs to manage a process

- The information stored on the PCB depends on the operating system and on the HW
  - Address space
    - Description of the memory regions of the process: code, data, stack,...
  - Execution context
    - SW: PID, scheduling information, information about the devices, accounting,...
    - HW: page table, program counter, ...
Process Control Block (PCB)

• Typical attributes are:
  • The process identifier (PID) and the parent process identifier (PPID)
  • Credentials: user and group
  • Environment variables, input arguments
  • CPU context (to save cpu registers when entering the kernel)
  • Process state: running, ready to run, blocked, stopped...
  • Data for I/O management
  • Data for memory management
  • Scheduling information
  • Resource accounting
  • ...

• We will deal with some of these attributes in the Lab
Multi-Process environment

• Usually there are many processes alive at a given time in a common OS

• Processes usually alternate using the CPU with other resource usage
  • In a multi-programmed environment the OS manages how resources are shared among processes

• In a general purpose system, the kernel alternates processes in the CPU
  • We have to alternate processes without losing the execution state
    • We will need a place to save/restore the processes execution state
    • We will need a mechanism to change from one process to another
  • We have to alternate processes being as much fair as possible
    • We will need a scheduling policy

• If the kernel makes this CPU sharing efficiently, users will have the feeling that a CPU is constantly assigned to the process
Parallelism vs Concurrency vs Sequential

- **Parallelism**: N processes run at a given time in N CPUs
  
  Time (each CPU executes one process)

- **Concurrency**: N processes could be potentially executed in parallel, but there are not enough resources to do it
  
  - The OS selects what process can run and what process has to wait

  Time (CPU is shared among processes)
Execution Flows (Threads)

• Analyzing the concept of Process...
  • the OS representation of a program during its execution
    ...we can state a **Process** is the resource allocation entity of a executing program (memory, I/O devices, threads)

• Among other resources, we can find the **execution flow/s (thread/s)** of a process
  • The execution flow is the basic scheduling entity the OS manages (CPU time allocation)
    • Every piece of code that can be independently executed can be bound to a thread
  • Threads have the required context to execute instruction flows
    • Identifier (Thread ID: TID)
    • Stack Pointer
    • Pointer to the next instruction to be executed (Program Counter),
    • Registers (Register File)
    • Errno variable
  • Threads **share** resources of the same process (PCB, memory, I/O devices)
Multi-threaded processes

• A process has a single thread when it is launched
• A process can create a number of additional threads
  • E.g.: current high-performance videogames comprise >50 threads; Firefox/Chrome show >80 threads
• The management of multi-threaded processes depends on the OS support
  • User Level Threads vs Kernel Level Threads
Execution Flows (Threads)

• When and what are threads used for...
  • Parallelism exploitation (code and hardware resources)
  • Task encapsulation (modular programming)
  • I/O efficiency (specific threads for I/O)
  • Service request pipelining (keep required QoS)

• Pros
  • Threads management (among threads of the same process) has less cost than process management
  • Threads can exchange data without syscalls, since they share memory

• Cons
  • Hard to code and debug due to shared memory
Process State

• The PCB holds the information required to exactly know the current status of the process execution

• Processes do not always use the CPU
  • E.g.: Waiting for data coming from a slow device, waiting for an event...

• The OS classifies processes based on what their are doing, this is called the process state
  • It is internally managed like a PCB attribute or grouping processes in different lists (or queues)
Process State Graph

• This is a generic process state graph approach mostly used by kernels, but...
  • ...every kernel defines its own process state graph with slight modifications
Kernel Internals for Process Management

- Data structures to keep per-process information and resource allocation

- Data structures to manage PCB’s, usually based on their state
  - In a general purpose system, such as Linux, Data structures are typically queues, multi-level queues, lists, hash tables, etc.

- Scheduling algorithms to select the next process to run in the CPU
Schedulers

• Schedulers are critical for the proper performance of the system

• **Short** term: every OS Tick
  • What is the next process to run in the CPU

• **Medium** term: when the OS detects it is running out of resources (E.g. Memory)
  • What processes are candidate to **temporally** release resources to let other processes use them

• **Long** term (**optional**): every start/end of a process
  • What is the maximum number of processes suitable to run in the system
    • It controls the multiprogrammed level of the system
Short Term Scheduler

• Every OS tick the scheduler checks whether another process has to run in the CPU

• **Non-Preemptive Policies**: the scheduler cannot put a process out of the running state
  • Only the process itself can decide to release the CPU (e.g. blocking I/O call)

• **Preemptive Policies**: the scheduler can put a process out of the running state in order to enable another process run instructions in the CPU
  • **Quantum**: period of time the scheduler grants a process to run in a row in the CPU
  • Priority/non-priority based policies
    • E.g. Round-Robin

• Schedulers of current general purpose OSes are based on complex approaches
  • Multiple policies using multiple queues
Impact of context switch on performance

• Context Switch: changing the process that is running in the CPU
  • It involves an overhead due to kernel code execution and manage the save/restore of a process context

Time (CPU is shared among processes)

CPU0 | Proc. 0 | Proc. 1 | Proc. 2 | Proc. 0 | Proc. 1 | Proc. 2 | Proc. 0 | Proc. 1 | Proc. 2 | ...

Proc. 0 | Proc. 1 | Proc. 2

Save context P0 | Scheduler selects P1 | Restore context P1

User Mode | Kernel Mode
Performance/Efficiency of a Scheduling Policy

• What is the main goal of the system?
  • Real-time systems versus High Performance Computing
    • It is not the same the device that manage the ABS of a car than a node in a Supercomputer

• The definition of “optimal” scheduling policy depends on the purpose of the system
  • Different metrics to find out whether a scheduling policy is well chosen
    • E.g. Response time, throughput, efficiency, turnaround time
Syscalls related to Process Management

• Process creation
  • A process creates a new child process
    • Fork();

• End of process execution
  • A process notifies to the kernel that it finishes its execution
    • Exit(int);

• Wait for a child process to finish
  • And allow the system to release its data structures (PCB, kernel stack...)
    • Waitpid(pid, @, int);

• Get process identifiers
  • Get the Process ID ("getpid();") and the Parent Process ID ("getppid();")

• Execute a new program
  • The process changes the program that is executing
    • Execlp(path, argvs, ..., NULL);
Error control

```c
#ifndef __ASM_GENERIC_ERRNO_BASE_H
#define __ASM_GENERIC_ERRNO_BASE_H

#define EPERM            1      /* Operation not permitted */
#define ENOENT           2      /* No such file or directory */
#define ESRCH            3      /* No such process */
#define EINTR            4      /* Interrupted system call */
#define EIO              5      /* I/O error */
#define ENXIO            6      /* No such device or address */
#define E2BIG            7      /* Argument list too long */
#define ENOEXEC          8      /* Exec format error */
#define EBADF            9      /* Bad file number */
#define ECHILD          10      /* No child processes */
#define EAGAIN          11      /* Try again */
#define ENOMEM          12      /* Out of memory */
#define EACCES          13      /* Permission denied */
#define EFAULT          14      /* Bad address */
#define ENOTBLK         15      /* Block device required */
#define EBUSY           16      /* Device or resource busy */
#define EEXIST         17      /* File exists */
... 
#define EHWPOISON       133     /* Memory page has hardware error */
```

• Common UNIX/Linux error codes
Table of Contents

• Operating System
• Basic Concepts
• Access to Kernel functions
• Process management
• Memory subsystem
• Other important OS concepts and tasks
Memory Management

• The CPU can only access directly to memory and the register bank
  • Instructions and data must be located in main memory

• The CPU sends out **logical addresses (logical @s)**
• The requested instructions/data are located in **physical addresses**
• Logical @s **may not directly match** the correspondent physical @s
  • The OS in conjunction with the Hardware manages this translation
    • logical @ → physical @

• The process uses **virtual memory** to become larger than main memory size
  • Logical addresses point to virtual memory locations
Program Loading

- The OS loads the program from the disk to Physical Memory
  1) Request & reserve space in main memory
  2) Load the program
  3) Start running

Binary file in the disk

Process Contents in Memory
Multiprogrammed OS

- Memory regions are private per process
  - In multiprogrammed systems (multiple processes are alive at a time) the OS must protect mem regions
  - Contents usually are distributed in remote physical memory locations
- The OS memory zone holds the kernel and all required internal data and routines
  - The OS memory space NEEDS protection

![Diagram of memory regions and processes]
Memory Management Unit (MMU)

- MMU: HW component which, at least, offers **address translation and memory access protection**. It can also support other management tasks.

- **OS is responsible for configuring the MMU** with the correct address translation values for the current process in execution
  - What logical @ are valid and what are their corresponding physical @
  - Guarantees that each process gets assigned only its own physical @

- **HW support to translation and protection between processes**
  - MMU receives a logical @ and translates it to the corresponding physical @ using its data structures
    - It throws an exception to the OS if the logical address is not marked as valid or if it has not associated a physical address
  - OS manage the exception according to the situation: **Segmentation Fault!!!**
    - **Exception**: involuntary issue triggered by the execution of an instruction
  - Optimization: Recent translations are kept inside a cache-like component
    - Translation Lookaside Buffer (TLB)
Multiprogrammed Systems

Process A and C are instances of the same program
Multiprogrammed Systems

Process A and C are instances of the same program
Process Logical Address Space

- Local variables, parameters and execution control
- Dynamic memory: runtime allocation
- Global variables
- Instructions

- Stack (max)
- Heap
- Data
- Code
### Process Contents in Memory

- **Stack**: dynamic mem
  - Function arguments
  - Local Variables
- **Shared libraries**: Code, data...
- **Heap**: dynamic mem
  - Mem allocated at runtime
- **Data**: .bss & .data
  - Global variables
- **Code**: .text
  - Instructions

![Diagram of physical and virtual memory](image)

- **Physical memory**
  - Memory blocks labeled P1, P2, P3, P4, P5
  - Memory addresses: 0x00000000 to 0xFFFFFFFF

- **Virtual memory**
  - Memory blocks labeled Stack, Shared libraries, Heap, Data, Code
  - Memory addresses: 0x00000000 to 0xFFFFFFFF
Static linking

- Static compilation and linking phases

```c
#include <unistd.h>
#include <stdio.h>

int global = 4;

int main(int argc, char **argv){
    char buf[512];
    int num;

    num = sprintf(buf, "Hello World %d!", global);
    write(1, buf, num);

    return 0;
}
```

Files in the disk
Process with static libraries

Logical addresses refer to virtual memory addresses due to the OS support

---

Virtual memory space (process)

Virtual memory space (OS)

I/O to devices, disks...

Memory mapping (to OS and process)

Files in the disk
Dinamically linking

• Dynamic compilation and linking phases

```c
#include <unistd.h>
#include <stdio.h>

int global = 0;

int main(int argc, char **argv){
    char buf[512];
    int num;
    num = sprintf(buf, "Hello World %d!", global);
    write(1, buf, num);
    return 0;
}
```
Process with shared libraries (Dynamic linking)

Virtual memory space (process)

Virtual memory space (OS)

I/O to devices, disks...

Memory mapping (to OS and process)

Files in the disk
Combining Views

Process A and C are instances of the same program
There is a global variable (e.g. “int counter;”)
- The compiler has assigned the logical @ 1000
  - Since it is an integer, the four bytes of the variable go from @1000 to @1003
Combining Views

Process A and C are instances of the same program
There is a global variable (e.g. “int counter;”)
- The compiler has assigned the logical @ 1000
  - Since it is an integer, the four bytes of the variable go from @1000 to @1003
Syscalls related to Memory Management

• The Heap is mainly employed for dynamic data structures
  • E.g. Structures used only for a period of time, unknown memory size requirements, allocated and deallocated often

• Syscalls related to heap management
  • Memory allocation
    • malloc (C library)
    • new (C++ library)
  • Memory deallocation
    • free (C library)
    • delete (C++ library)
  • All above calls invoke a system call to modify the limit of the Heap Mem Zone (brk)
Memory Allocation

• (type) `malloc(int size); // C language`
  • Returns the starting @ of the newly allocated size Bytes in the HEAP

• `new type[size]; // C++ language`
  • Returns the starting @ of the sizeof(type)*size Bytes in the HEAP

• Behavior
  • Before allocating @, it checks whether the HEAP has space enough to hold size bytes
    • If not, the OS increases the limit of the HEAP (sbrk)
    • The HEAP size is increased by a configurable number of bytes to reduce the number of times it has to be increased
  • The object memory zone is allocated in the HEAP following a placement algorithm
    • Can group objects by their size...
Memory Allocation

Starting status

After calling:
```
int *ptr = (int) malloc(4*sizeof(int));
```

After calling:
```
char *string = (char) malloc(2*sizeof(char));
```

2 chars
2 Bytes
Memory Deallocation

• `free(pointer);` // C language
• `delete [] pointer;` // C++ language
  • In both cases the memory zone pointed by the pointer is released

• Behavior
  • Deallocation of the memory zone pointed by the input parameter
  • The OS will consider to reduce or not the HEAP memory space
  • A pair of lists are maintained by malloc/new/free/delete
    • List of objects allocated
    • List of objects deallocated (may be merged)
Memory Deallocation

Starting status

Stack

Heap

Data

Code

After calling:
free(string);

Stack

Heap

Data

After calling:
free(ptr);

Stack

Heap

Data

Code
Memory Assignment

• **Contiguous assignment**
  - Physical address space is contiguous
    - The whole process is loaded on a partition which is selected at loading time
    - It is not flexible and complicates to apply optimizations (as, for example, on-demand loading)

• **Non-contiguous assignment**
  - Physical address space is not contiguous
  - Flexible
  - Increases granularity to the memory management of a process
  - Increases complexity of OS and MMU

• Based on
  - **Paging** (fixed partitions)
  - **Segmentation** (variable partitions)
  - Combined schemes
    - For example, paged segmentation
Memory Assignment Issue: Fragmentation

• **Fragmentation problem**: when it is not possible to satisfy a given memory request although the system has enough memory to do it. There is free memory, but it cannot be assigned to a process
  • Also mentioned in disk management

• **Internal fragmentation**: memory assigned (to a process) that is not going to be used

• **External fragmentation**: free memory that cannot be used to satisfy any memory request because it is not contiguous and large enough for the requirements
  • It can be avoided compacting the free memory. It is necessary the system to support address translation at runtime.
Paging

- Logical address space is divided into fixed size partitions: **pages**
- Physical memory is divided into partitions of the same size: **frames**
- Assignment
  - For each page look for a free frame
    - List of free frames
    - Can cause internal fragmentation
- The frames assigned to a process go back to the free frames list when the process ends the execution
- **Page: working unit of the OS**
  - Facilitates on-demand loading
  - Enables page-level protection
  - Usually, a page belongs to just one memory region to match region protection requirements (code/data/heap/stack)
- Can cause internal fragmentation
• Logical address space divided into regions, considering the type of content (code, data, etc.)
• Logical address space divided into variable size partitions (segments), that fit the size that is really needed
  • At least 3 segments: one for code, one for stack and one for data
  • References to memory are composed of segment and offset
• Assignment: for each segment in a process
  • Look for a partition big enough to hold the segment
  • Possible policies: first fit, best fit, worst fit
  • Select from the partition just the amount of memory needed to hold the segment and the rest of the partition is kept in a free partitions list
• Can cause external fragmentation
Combined Scheme

• Paged Segmentation: 2 step translation

  - Logical address space is divided into segments
    - Code, data, heap, stack

  - Segments are divided into pages
    - Segment size is multiple of page size
    - Page is OS working unit
Memory Management

• Virtual memory
  • Split in pages (4KB usually, can be 2-4 MBytes)
  • A page can be
    • Valid and present
    • Valid and not present
    • Invalid

• Physical memory
  • Split in page frames
    • Same capacity as virtual pages
  • OS keeps information about
    • Available frames
    • Busy frames
When Memory is exhausted...

• OS constantly checks memory availability (used vs free) of the system

• If the memory becomes exhausted (less than a given availability threshold) OS starts an alternative support approach:
  • Difference between page replacement (paging) and process replacement (swapping)
  • Pages are selected->Written to the swap area [if modified]->Reallocate other virtual pages

• **Swap area**: special memory area, usually in a storage device (e.g. disk) to temporary hold contents of the main memory that have not space enough
  • It is considered as additional space to the physical memory
Table of Contents

• Operating System
• Basic Concepts
• Access to Kernel functions
• Process management
• Memory subsystem
• Other important OS concepts and tasks
Other important OS concepts and tasks

• The OS is in charge of many other management tasks
  • I/O subsystem
  • Filesystem
  • Inter-process communication

• Some of them will be addressed in this course

• New technologies are arising to provide new functionalities to the OS
  • Distributed systems
  • Virtualization technologies used for different management purposes
  • Accelerators (GPUs, FPGAs...) management

• Some of them will be addressed in next courses
Bibliography

• Computer Organization and Design (5th Edition)
  • D. Patterson and J. Hennessy
  • [http://cataleg.upc.edu/record=b1431482~S1*cat](http://cataleg.upc.edu/record=b1431482~S1*cat)
    • Introduces hardware support for OS

• Operating System Concepts (John Wiley & Sons, INC. 2014)
  • Silberschatz, A; Galvin, P. B; Gagne, G.
  • [http://cataleg.upc.edu/record=b1431631~S1*cat](http://cataleg.upc.edu/record=b1431631~S1*cat)
    • Introduces the presented concepts about OS