Computer Systems
8. The OS level — Processes

Spyros Voulgaris

Vrije Universiteit
1 Introduction

2 System Calls

3 Processes

4 Inter-process communication
The Operating System level

- Assembly programs
  - Direct execution
  - System calls
  - OS level
- ISA level
  - Interpretation
- Micro-architecture level
  - Direct execution
- Digital logic level

Introduction
What is an operating system?

An operating system is an **extended machine**
- Implements higher-level operations than the ISA level
- New operations: system calls
- The OS is a hybrid layer: programs can also use the ISA level directly

An operating system is a **resource manager**
- Makes sure several programs can run simultaneously
- Makes sure to organize device access between multiple programs (keeps track of resource usage, grants/retracts access to resources, arbitrates conflicts, etc.)
Processes

- A process is a program in execution
  - Private memory address space (isolated from other processes)
  - Private stack, PC & SP registers, etc.
  - A process is always created by another process

- The OS takes care of process scheduling
  - At one instant of time, only one process can execute on a given CPU
  - The OS makes sure that each process gets a fair share of the CPU

- Processes are isolated from each other
  - For a process it feels like it has the whole machine for itself
  - Communication between processes must be done through explicit inter-process communication
Processes want to exchange data with external devices

Every device is different!
- Different types (disk, keyboard, network, etc.)
- Different speeds
- Different manufacturers
- Etc.

The OS must provide:
- A homogeneous access interface to all sorts of different devices
- It must also arbitrate between multiple processes which want to use the same devices at the same time
Memory management

- Processes want to use memory
  - But we want to isolate the memory from one process from the others
  - Better: give each process the illusion that it can access the entire computer’s memory without being disturbed by the others
  - Even better: give each process the illusion that it can use the entire address space (bytes 0 – $2^{32}$) regardless of installed physical memory

- The OS must provide:
  - Memory isolation
  - Paging (i.e., move pieces of memory to disk to make space for others)
  - Memory mapping (i.e., give each process the entire address space)
Is there anybody here who does not know what a file is? :-)

- How do we create files, directories, permissions, and so on
- Using block storage on disk

Multiple file systems can be mounted into a single hierarchy

Before mounting

Root

Floppy

(a)

After mounting

Root

Floppy

(b)

Special files

- Many OS concepts are represented as files
- Devices, inter-process communication mechanisms, etc.
System calls

- System calls are *the way processes can request an operation to be performed by the OS*

- System calls for *process management*:

```
pid = fork()                     Create a child process identical to the parent
pid = waitpid(pid, &statloc, opts)  Wait for a child to terminate
s = wait(&status)                 Old version of waitpid
s = execve(name, argv, envp)      Replace a process core image
exit(status)                      Terminate process execution and return status
size = brk(addr)                  Set the size of the data segment
pid = getpid()                    Return the caller’s process id
pid = getpgrp()                   Return the id of the caller’s process group
pid = setsid()                    Create a new session and return its process group id
l = ptrace(req, pid, addr, data)  Used for debugging
```
System calls for **signal management**:

- `s = sigaction(sig, &act, &oldact)` | Define action to take on signals
- `s = sigreturn(&context)` | Return from a signal
- `s = sigprocmask(how, &set, &old)` | Examine or change the signal mask
- `s = sigpending(set)` | Get the set of blocked signals
- `s = sigsuspend(sigmask)` | Replace the signal mask and suspend the process
- `s = kill(pid, sig)` | Send a signal to a process
- `residual = alarm(seconds)` | Set the alarm clock
- `s = pause()` | Suspend the caller until the next signal
System calls for file management:

- `fd = creat(name, mode)`
- `fd = mknod(name, mode, addr)`
- `fd = open(file, how, ...)`
- `s = close(fd)`
- `n = read(fd, buffer, nbytes)`
- `n = write(fd, buffer, nbytes)`
- `pos = lseek(fd, offset, whence)`
- `s = stat(name, &buf)`
- `s = fstat(fd, &buf)`
- `fd = dup(fd)`
- `s = pipe(&fd[0])`
- `s = ioctl(fd, request, argp)`
- `s = access(name, amode)`
- `s = rename(old, new)`
- `s = fcntl(fd, cmd, ...)`

Obsolete way to create a new file
Create a regular, special, or directory i-node
Open a file for reading, writing or both
Close an open file
Read data from a file into a buffer
Write data from a buffer into a file
Move the file pointer
Get a file’s status information
Get a file’s status information
Allocate a new file descriptor for an open file
Create a pipe
Perform special operations on a file
Check a file’s accessibility
Give a file a new name
File locking and other operations
System calls for directory and file system management:

- `s = mkdir(name, mode)` Create a new directory
- `s = rmdir(name)` Remove an empty directory
- `s = link(name1, name2)` Create a new entry, name2, pointing to name1
- `s = unlink(name)` Remove a directory entry
- `s = mount(special, name, flag)` Mount a file system
- `s = umount(special)` Unmount a file system
- `s = sync()` Flush all cached blocks to the disk
- `s = chdir(dirmame)` Change the working directory
- `s = chroot(dirmame)` Change the root directory
System calls for protection:

- `s = chmod(name, mode)` Change a file’s protection bits
- `uid = getuid()` Get the caller’s uid
- `gid = getgid()` Get the caller’s gid
- `s = setuid(uid)` Set the caller’s uid
- `s = setgid(gid)` Set the caller’s gid
- `s = chown(name, owner, group)` Change a file’s owner and group
- `oldmask = umask(complmode)` Change the mode mask
System calls

- System calls for **time management**:

  \[
  \text{seconds} = \text{time}(&\text{seconds}) \\
  s = \text{stime}(tp) \\
  s = \text{utime}(\text{file}, \text{timep}) \\
  s = \text{times}(\text{buffer})
  \]

  - Get the elapsed time since Jan. 1, 1970
  - Set the elapsed time since Jan. 1, 1970
  - Set a file’s "last access" time
  - Get the user and system times used so far
How can a single CPU execute several processes simultaneously?

(a) Multiprogramming: alternate between several programs
(b) Conceptually, each program has its own program counter and executes in isolation
(c) But only one process is active at any instant
Process creation and deletion

A process is created:
- At system initialization
- When another process calls the system call for process creation

A process terminates:
- Upon normal program exit (voluntary)
- Upon error program exit (voluntary)
- Upon fatal error (involuntary)
- When killed by another process (involuntary)
A process can be in 3 main possible states:

- **Running** (actually using the CPU at that instant)
- **Waiting** (Runnable but temporarily stopped to let another process run)
- **Blocked** (Unable to run until some external event happens)
Concretely, a process is composed of:

- An image: the executable code of the program it is running
- Some memory: to keep a copy of the executable code, variables, stack, etc.
- Descriptors of system resources currently used by the process (files)
- Security attributes: ID of the process owner, set of permissions, etc.
- Process state:
  - Content of the registers
  - Program counter
  - Stack pointer
  - Pending signals
  - And a few more...

Details vary from one OS to the next...
Processes are either CPU-bound or I/O bound

CPU bound:

I/O bound:
Process scheduling

- When more than one processes are ready to execute, the **process scheduler** must choose which one to execute.

- Many **scheduling disciplines** are possible:
  - **Round-Robin** Each process is given an opportunity to execute for a given **time quantum**. If a process cannot use its full quantum, we move to the next.
    - How long should the quantum be? Responsiveness vs. efficiency
  - **Priority** Give each process a **priority level**. The waiting process with the highest priority is selected.
    - Low-priority processes may starve!
**Fair-share** Tag processes with the ID of their user, make sure each user gets a fair share of the CPU regardless of the number of processes they create.

**Real-time** Give strong guarantees to certain processes that they will execute at the time they need (periodic or aperiodic).

- The system can sustain $m$ periodic events with period $P_i$ that require $C_i$ seconds each time if

$$\sum_{i=1}^{m} \frac{C_i}{P_i} \leq 1$$

- Mostly used for multimedia systems
Context switch

- **Cooperative context switch:**
  - Happens when the process voluntarily initiates a context switch
  - Happens every time a process issues a system call

- **Preemptive scheduling:**
  - Happens after a process has used up all its time slice
  - Triggered by a timer interrupt (setup by the kernel before scheduling the process)
Table of Contents

1. Introduction
2. System Calls
3. Processes
4. Inter-process communication
Processes run in full isolation from each other
- Private memory address space
- Private PC, stack, etc.
- Each process has the illusion of being alone in the system

BUT...

Processes also often need to communicate with each other
- To synchronize their actions in time
- To exchange data
IPC for inter-process data exchange

- Files
- Signals
- Pipes
  - FIFO communication channel between two processes
- Message passing (mostly in $\mu$-kernels)
  - send(destination, message);
  - message = receive(source);
- Shared memory (mostly in monolithic kernels)
  - Multiple processes simultaneously access to the same memory region
  - Efficient but dangerous because of race conditions!
Race conditions can happen when several processes share some storage that they can read and write.

Example: a print spooler directory
- Files are numbered in slots
- Two processes decide simultaneously to add a file to the spooler
Mutual exclusion

- We need to make sure that only one process at a time manipulates the spool directory.
  1. No two processes may be simultaneously inside their critical regions.
  2. No assumptions may be made about speeds or the number of CPUs.
  3. No process running outside its critical region may block other processes.
  4. No process should have to wait forever to enter its critical region.
Mutual exclusion done wrong: busy waiting

This implements strict alternation: process A cannot enter its critical section twice in a row.

Busy waiting: this wastes CPU resources!

(a)

while (TRUE) {
    while (turn != 0) /* loop */;
    critical_region();
    turn = 1;
    noncritical_region();
}

(b)

while (TRUE) {
    while (turn != 1) /* loop */;
    critical_region();
    turn = 0;
    noncritical_region();
}
Mutual exclusion done better: the TSL instruction

- All good ISAs support a Test and Set Lock instruction: TSL RX,LOCK
  - Copies the contents of LOCK into register RX
  - Writes an 1 into the LOCK variable
  - These two actions are indivisible: no other process or CPU can access the LOCK variable during that time

```plaintext
enter_region:
tsl register,lock          | copy lock to register and set lock to 1
cmp register,#0           | was lock zero?
jne enter_region          | if it was non zero, lock was set, so loop
ret                        | return to caller; critical region entered

leave_region:
move lock,#0               | store a 0 in lock
ret                        | return to caller
```

- **Good:** we solved the strict alternation problem
- **Bad:** we still have busy waiting
Mutual exclusion done right: mutex

mutex_lock:
  TSL REGISTER,MUTEX  | copy mutex to register and set mutex to 1
  CMP REGISTER,#0     | was mutex zero?
  JZE ok              | if it was zero, mutex was unlocked, so return
  CALL thread_yield   | mutex is busy; schedule another thread
  JMP mutex_lock      | try again later
ok: RET | return to caller; critical region entered

mutex_unlock:
  MOVE MUTEX,#0       | store a 0 in mutex
  RET | return to caller

- No more busy waiting! The process checks the lock once each time it gets scheduled
- Extended version of Mutexes: semaphores
Read chapters 1 (Introduction) and 2 (Processes) of the Wikibook on operating systems (see Course Documents on Blackboard)