Operating Systems

II. Processes

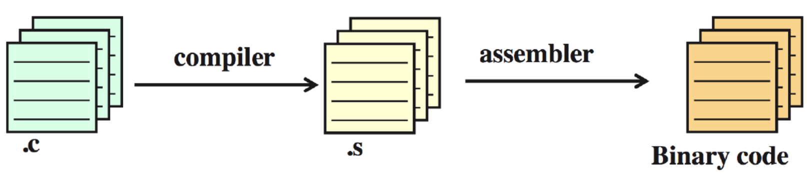
Ludovic Apvrille
ludovic.apvrille@telecom-paris.fr
Eurecom, office 470

perso.telecom-paris.fr/apvrille/OS/

Program

Abstraction

- Program is usually written in a high level language
- Compilers / interpreters convert high level languages into binary code



Process Definition

Definition of a process

Program in execution

Programs and processes

- One execution of a sequential program = one process
- Two executions of the same program = two processes

Computer system = set of processes

- Operating System processes
- User processes

Users

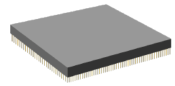
User: Definition

Person authorized to run processes

Features

- UserId, groupId
- Access rights to resources
  - On processes
  - On files
  - ...
- Super-user
  - *Administrator* (Windows), *root* (Unix)

CPU Protection



Goal

The OS must be sure to periodically gain control

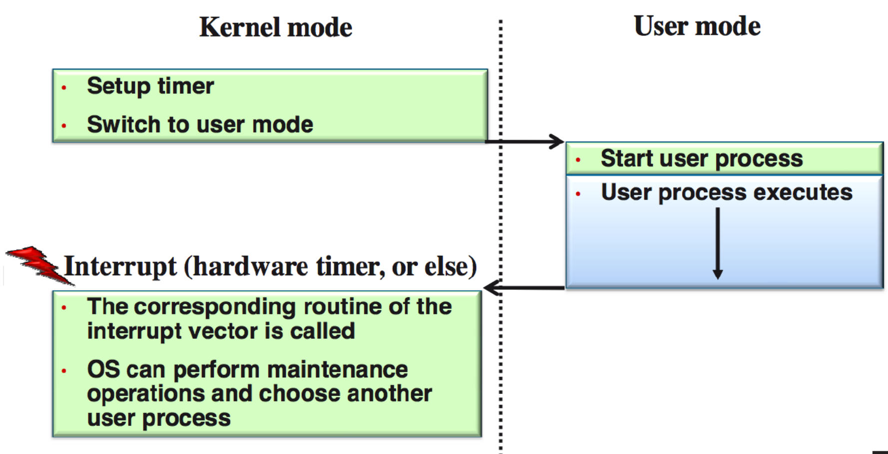
- Ensure CPU fairness between processes
- Prevent a process from stucking the system
  - e.g., infinite loop

Example of mechanisms

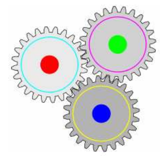
1. A hardware timer is set before a process is given the CPU
2. The timer interrupts the process after a specified period

Of course, instructions for settling the timer are privileged

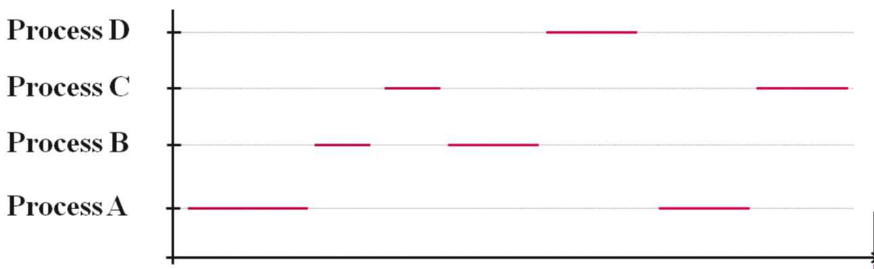
Example of CPU Protection



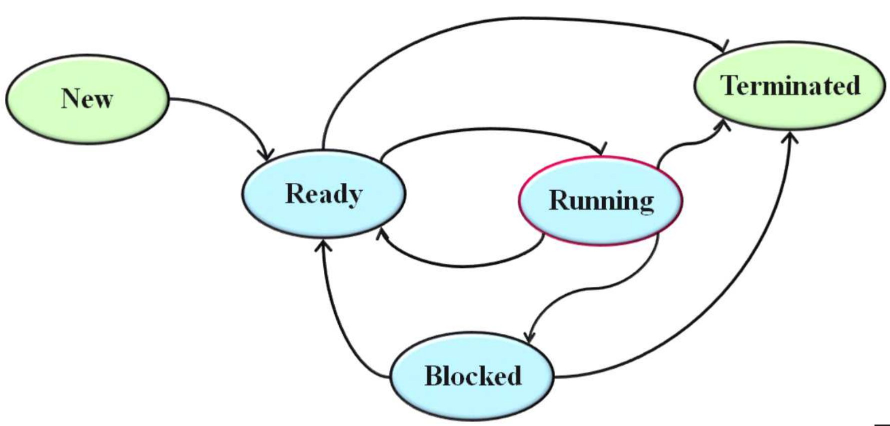
Running a Set of Processes



- Monoprocessor
  - Pseudo-parallelism: 1 process running at a time
    - So, either the OS or a user process is running
- Multiprocessor
  - A process can be running on each processor
- The OS scheduler dispatches processes on processors
  - Scheduling policy



States of Processes



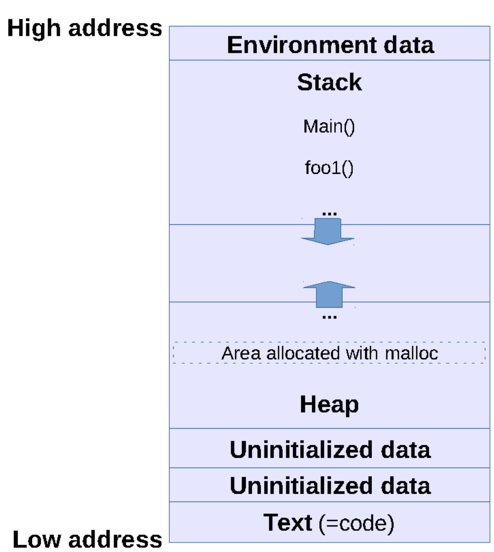
Process Data

- Data of processes are stored in memory

Various data of a process

- **Program code** = text section (static)
- **Current Activity**
  - Program counter = Processor's register
  - Next instruction to execute
- **Stack**: function calls are stored in a LIFO manner
  - Function parameters
  - Return address
  - Local variables
- **Heap**: Data section

Memory Layout of a C Program



Memory Allocation in C Programs

```
int a;

int funnyAllocation(char *buf, int b) {
  a = 5;
  b = b +1;
  strcpy(buf, "hello");

  return 7;
}

int main( int argc, char*argv[] ) {
  int b = 3;

  char *buf = (char *) ( malloc(sizeof(char) * 20));

  int returned = funnyAllocation(buf, b);

  printf("The returned value is: d\n", returned);
  printf("The value of b is: d\n", b);
  printf("The content of buf is: s\n", buf);
}
```


Memory Allocation in C Programs (Cont.)

```
$ gcc -Wall -o procmem procmem.c

$ ./procmem
The returned value is: 7
The value of b is: 3
The content of buf is: hello
```

Logical Organization of Processes

UNIX

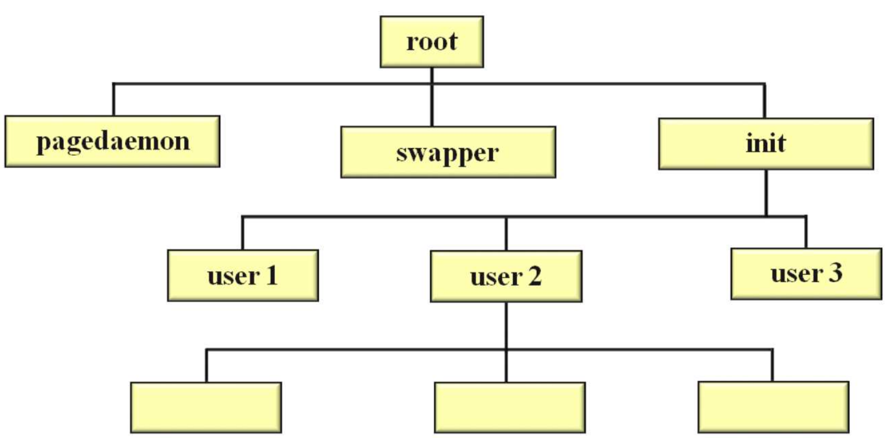
- Process hierarchy
  - A parent can possibly have many children
  - A child has exactly one parent
- Group of a process = this process + children + further descendants
- All processes belong to the group of the *init* process (root)

Windows

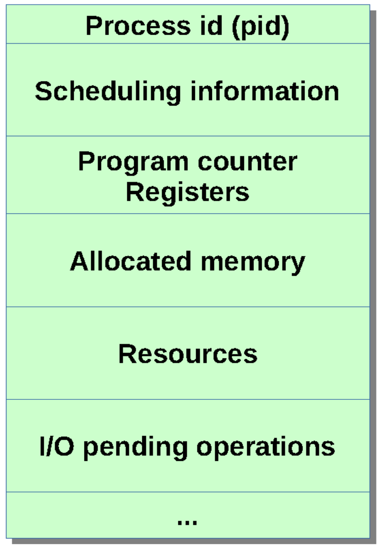
- No process hierarchy
- When a parent creates a child, it is given a special token (called a *handle*)
  - A handle can be passed to other processes
  - $\neq$ UNIX: processes cannot disinherit their children

Concepts | Process management | Boot sequence of an OS | Managing processes | And one last question!

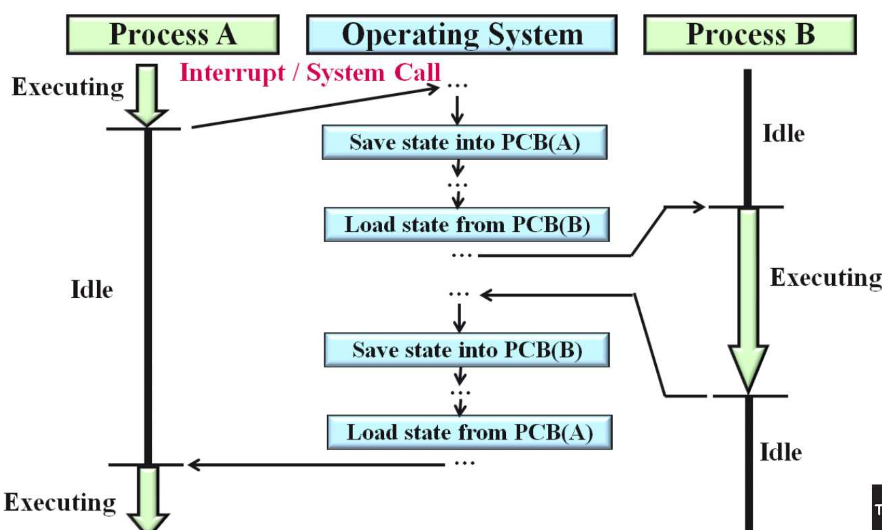
UNIX: Hierarchy of Processes



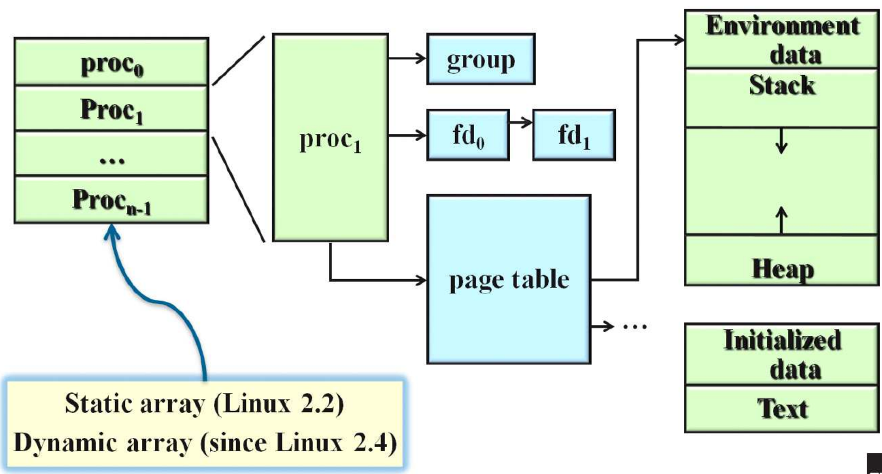
Process Control Block



Switching Between Processes



Processes Data Structure



Four Important Issues Regarding Processes

1. How is it possible to launch the first user process?

Boot sequence

2. How to manage processes from a programmer's point of view?

APIs

3. How to schedule processes (efficiently)?

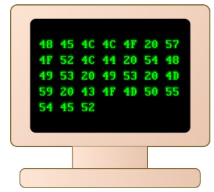
Scheduling policies

4. How can processes communicate with each others?

- Signals, shared memory, message passing, etc.
- Synchronization between processes

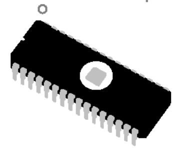
Boot, APIs: next slides ...
For other issues: attend next courses!

Steps of Boot Sequence



1. Reset of the hardware
   - All logic gates are reset to a known state
2. Diagnostic tests are run from the PROM Monitor
3. Boot manager
4. Starting of the OS kernel
5. User processes can execute!

Step 2: The PROM Monitor



Power On Self Test

- System hardware is initialized
- Miniature diagnostic program: Ensures a minimal operational base for the OS to run (memory, keyboard)
- $\Rightarrow$ Does not guarantee that the hardware is fully functional!

Scan of buses

To detect hardware devices connected to the system

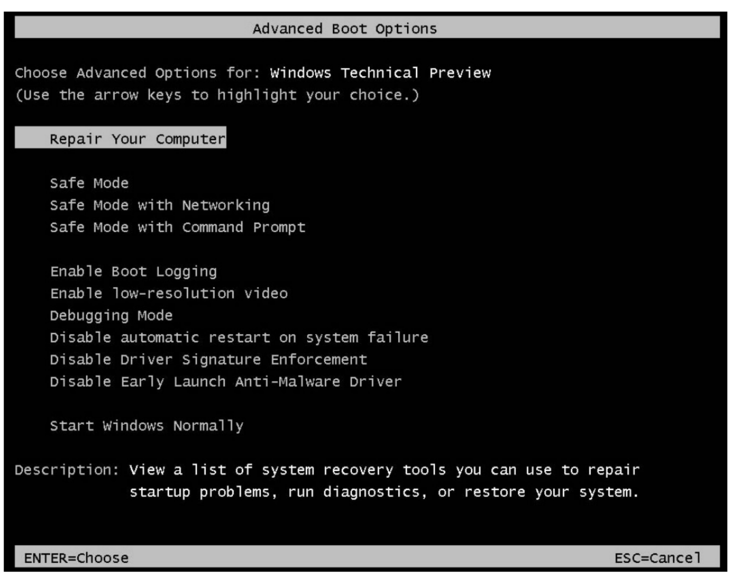
Loading and starting the boot manager

(See next slide)

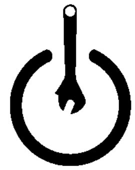
Step 3: Boot Manager

- Examples: BIOS, (U)EFI
- Initialization of some of the devices and annex memories
- Execution of boot loader
  - Kernel selection
  - Selection of boot parameters
    - Maintenance mode, multi-user mode, etc.
  - Loads the chosen kernel, and starts it
  - Example of boot loaders
    - Grub
    - Windows multi-boot loader
    - rEFInd

Boot Menu of Windows



Step 4: Starting the OS Kernel



- Allocation of memory
  - Kernel, data storage areas, I/O buffers
- Probe of devices
  - The kernel builds a device tree and loads corresponding devices
- Creation of first system processes
  - Swapper (sched), init, pagedaemon
- Start-up scripts are executed according to run-level
  - Start system services (e.g., *rlogin* daemon)
  - Windows runlevels
    - Multi-user, Safe mode, Safe mode with network

Controlling Processes

- Creation of new processes
- Management of processes
- Termination of processes

Starting and Terminating a Process

A C program starts its execution with a call to:

```
int main(int argc, char *argv[])
```

Termination

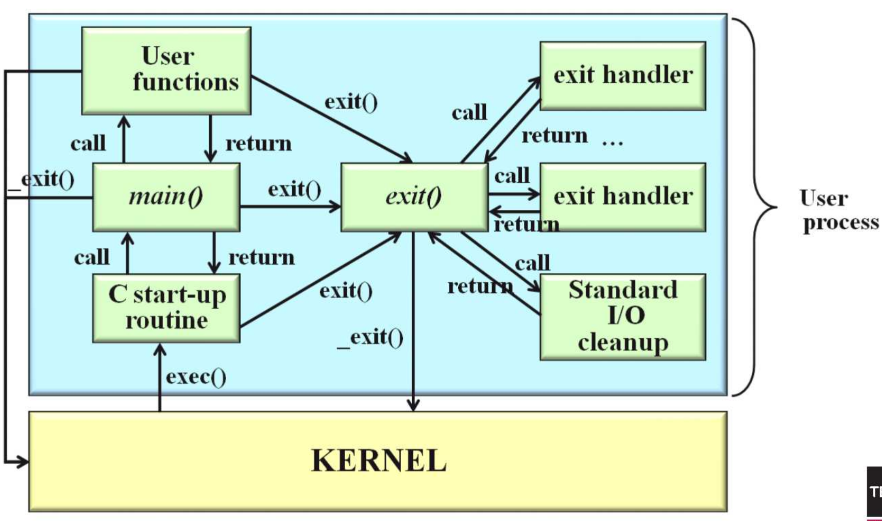
- Normal termination
  - Return from main
  - Call to *exit()* (or *_exit()*)
- Abnormal termination
  - Call to *abort()*
  - Allocated resources exceeded
  - Cascading termination
  - ...

```
EXIT(3)
#include <stdlib.h>
void exit(int status)

_EXIT(2)
#include <unistd.h>
void _exit(int status)

ABORT(3)
#include <stdlib.h>
 void abort(void);
```


"Life" of a C Program (Linux)



Call to *_exit()*

1. Process resources are freed

- File descriptors
- Shared memory segments
- Semaphores
- Message queues
- ...

2. The process becomes a zombie process

- Entry in the process allocation table is conserved
- Children's parent is set to the *init* process
- All children executing in foreground are terminated

UNIX: Creation of a New Process with *fork()*

- A running process calls *fork()*

fork()

```
#include <sys/types.h>
#include <unistd.h>

pid_t fork(void);
```

- The new process is a child process
- The function is called once but returns twice:
  - 0 is returned to the child process
  - The *pid* (process id) of the child is return to the parent process
- For more information on *fork()*

```
$man fork
```

fork(): Code Example

Code

```c
#include <sys/types.h>

int glob = 6;

int main(void) {
  pid_t ret;
  int var = 66;

  ret = fork();
  if (ret == 0) { /* Child */
    glob++; var++;
  } else {
    sleep(1);
  }
  printf("process ret = %d glob=%d var=%d\n", ret, glob, var);
  exit(0);
}
```

What happens when executing this code?
What must be improved in that code?

fork(): Drawback and Solution

Drawback: Created child is a clone of its parent

Recopy of data space, heap, stack, etc.
⇒ Not very efficient

A few solutions...

- *fork()* under Linux: Copy-On-Write technique
  - Memory pages shared by parent and child
  - Memory pages set to read-only for both
  - Copy of the memory page when a write operation is performed
- *vfork()*
  - Parent process is suspended until the child process makes a call to *exec()* or to *exit()* (Linux)
- *exec()*
  - Replaces the current process image with a new process image

vfork() and *exec()*: Code Example

Code

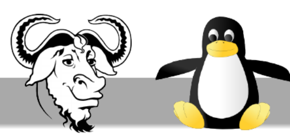
```
#include <sys/types.h>
#include <unistd.h>
int glob = 6;

int main(void) {
  int var = 66;

  pid_t ret = fork();
  if (ret == 0) { /* Child */
    if (execl("/bin/sh", "sh", "-c", "/bin/ls", NULL) <0) {
      exit(127); /* error */
    }
  } else {
    sleep(1);
  }
  printf("process ret = %d glob=%d var=%d\n", ret, glob, var);
  exit(0);
}
```

What happens when executing this program?

Linux: Excerpt of The Manual of *fork()*



fork()

fork() creates a new process by duplicating the calling process. The new process is referred to as the child process. The calling process is referred to as the parent process.
The child process and the parent process run in separate memory spaces. At the time of fork() both memory spaces have the same content. Memory writes, file map- pings (mmap(2)), and unmappings (munmap(2)) performed by one of the processes do not affect the other.
The child process is an exact duplicate of the parent process except for the following points:

- The child has its own unique process ID, and this PID does not match the ID of any existing process group (setpgid(2)).
- The child's parent process ID is the same as the parent's process ID.
- . . . (info on locks, etc.)

Linux: Excerpt of The Manual of *vfork()*

vfork()

vfork, just like fork(2), creates a child process of the calling process.
vfork() is a special case of clone(2). It is used to create new processes without copying the page tables of the parent process. It may be useful in performance sensitive applications where a child will be created which then immediately issues an execve.
vfork() differs from fork in that the parent is suspended until the child makes a call to execve(2) or exit(2). The child shares all memory with its parent, including the stack, until execve is issued by the child. The child must not return from the current function or call exit, but may call _exit().

Solaris: Excerpt of The Manual of *fork()* and *vfork()*

fork()

The fork() and fork1() functions create a new process. The new process (child process) is an exact copy of the calling process (parent process). ...

vfork()

The vfork() function creates new processes without fully copying the address space of the old process. This function is useful in instances where the purpose of a fork(2) operation would be to create a new system context for an execve() operation (see exec(2)). ...

POSIX: Excerpt of The Manual of *vfork()*

POSIX = Portable Operating System Interface based on UNIX

vfork()

The vfork() function has the same effect as fork(), except that the behavior is undefined if the process created by vfork() either modifies any data other than a variable of type pid_t used to store the return value from vfork(), or returns from the function in which vfork() was called, or calls any other function before successfully calling _exit() or one of the exec family of functions.

(*For the origin of the POSIX name, see http://stallman.org/articles/posix.html*)

vfork(): Another Code Example

Code

```
#include <sys/types.h>
int glob = 6;

int main(void) {
  pid_t ret;
  int var = 66;

  ret = vfork();  // previously was fork
  if (ret == 0) { /* Child */
    glob++; var++;
  } else { sleep(1);}
  printf("process ret = %d glob=%d var=%d\n", ret, glob, var);
  exit(0);
}
```

Execution

```
process ret = 0 glob=7 var=67
process ret = 7262 glob=7 var=67
```

Questions

When I need some information on a function provided by an Operating System, what should I do?

- Use google? wikipedia? Other sites?
- Use local manual pages?

Why?

Local manual pages

```
$ man -s2 fork
$ info fork
$ man -s3 abort
...
```