UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona

Process Management

COMPUTER ARCHITECTURE AND OPERATING SYSTEMS

Bioinformatics

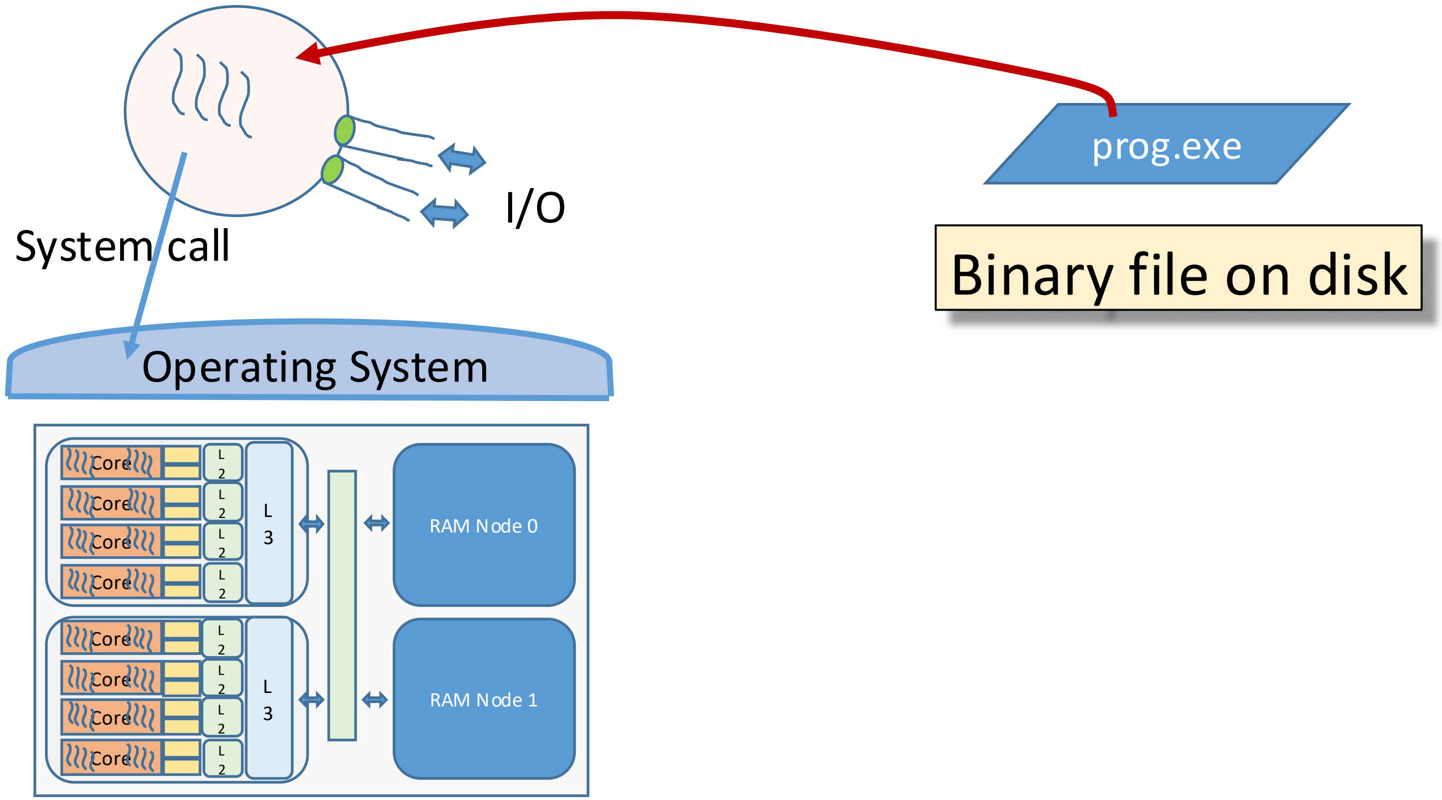
2025/26 Spring Term

Jordi Fornés

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Departament d'Arquitectura de Computadors

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 on 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 hardware
  - Address space
    - Description of the memory regions of the process: code, data, stack,…
  - Execution context
    - SW: PID, scheduling information, information about 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

- Parallelism: N processes run at a given time in N CPUs

Time(each CPU executes one process)

CPU0	Proc. 0
CPU1	Proc. 1
CPU2	Proc 2

- 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)

CPU0	**Proc. 0**	**Proc. 1**	**Proc. 2**	**Proc. 0**	**Proc. 1**	**Proc. 2**	**Proc. 0**	**Proc. 1**	**Proc. 2**	**...**

Execution Flows (Threads)

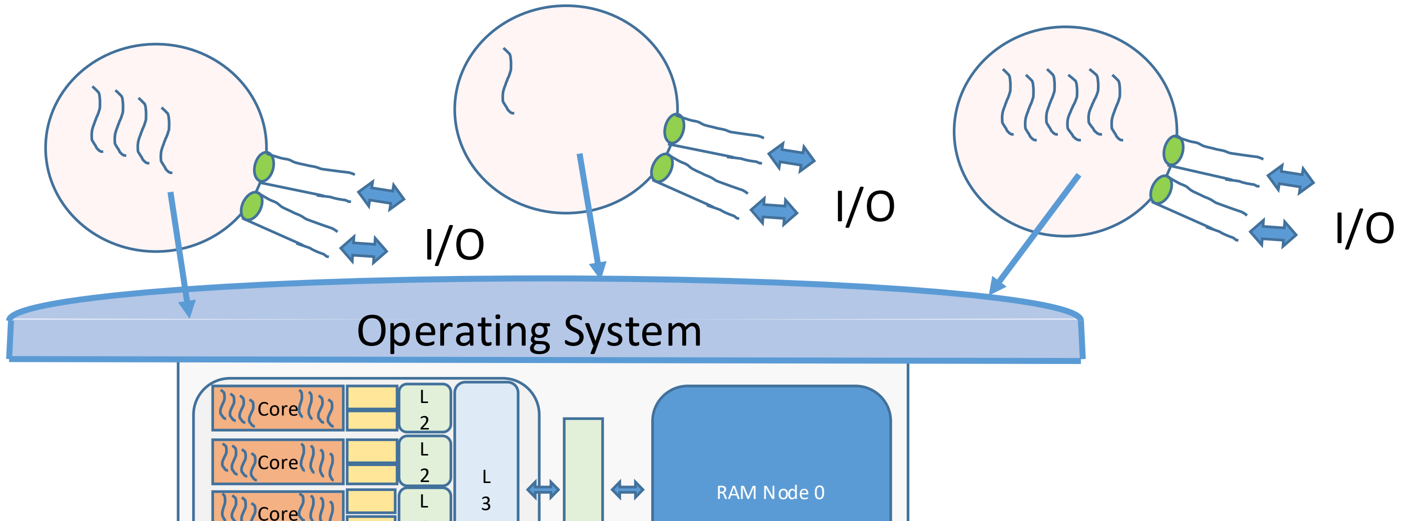
- Analyzing the concept of Process…
  - the OS representation of a program during its execution

…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

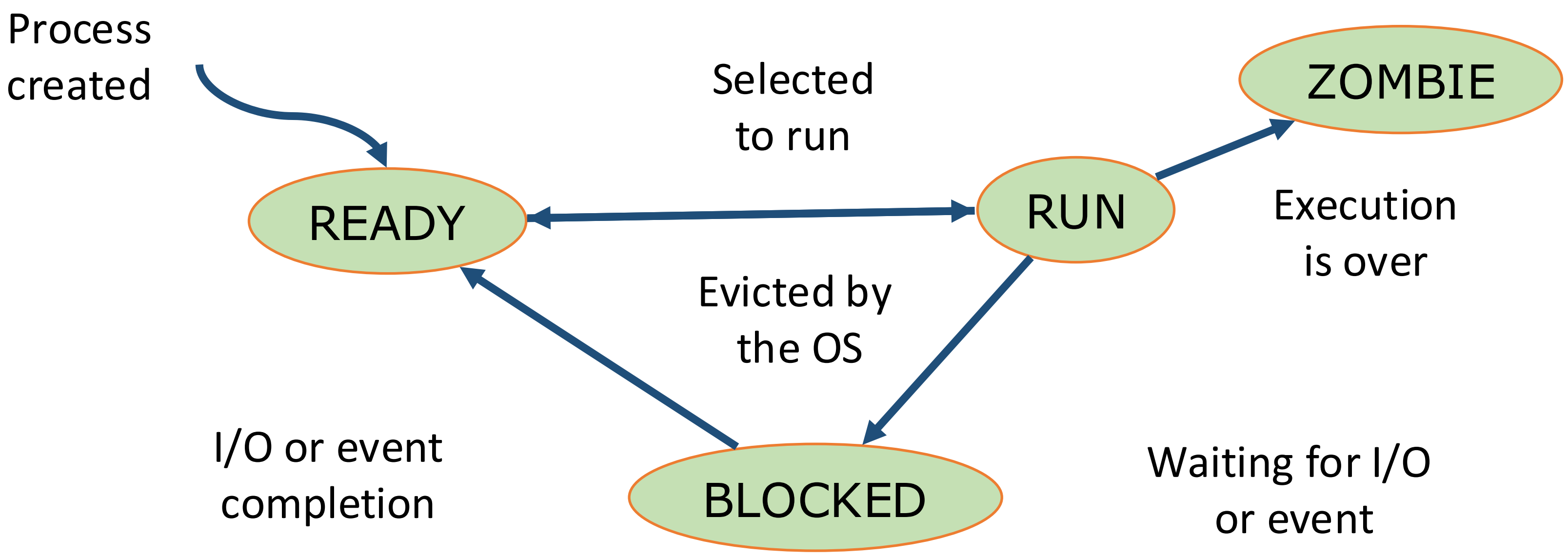
Process State

- 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, hast 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

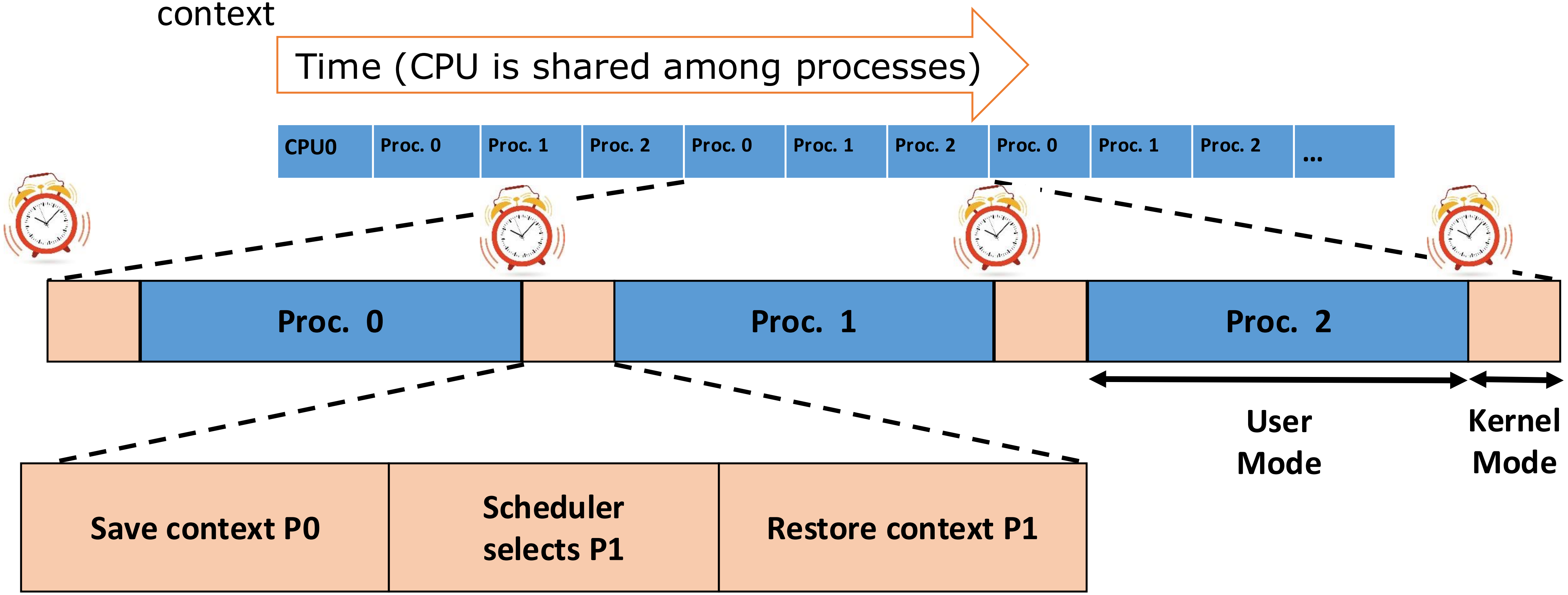
- **Long** term (*optional*): every start/end of a process
  - What is the maximum number of processes suitable to run in the system
    - It controls the multiprogrammed level of the system

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



Performance/Efficiency of a Scheduling Policy

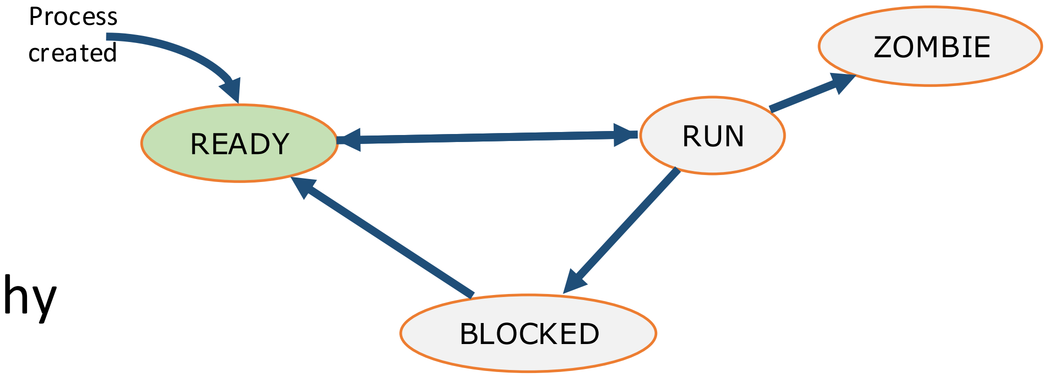
- What is the main goal of the system?
  - Real-time systems versus High Performance Computing
    - It is not the same the device that manage the ABS of a car than a node in a Supercomputer
- The definition of “optimal” scheduling policy depends on the purpose of the system
  - Different metrics to 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
- End of process execution
  - A process notifies to the kernel that it finishes its execution
- Wait for a child process to finish
  - And allow the system to release its data structures (PCB, kernel stack…)
- Get process identifiers
  - Get the Process ID (os.getpid()) and the Parent Process ID (os.getppid())
- Execute a new program
  - The process changes the program that is executing

Process Creation

os.fork()

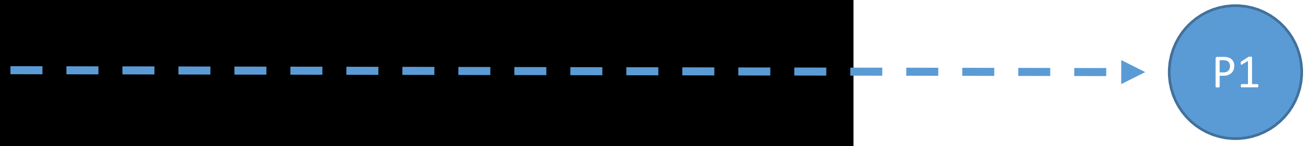


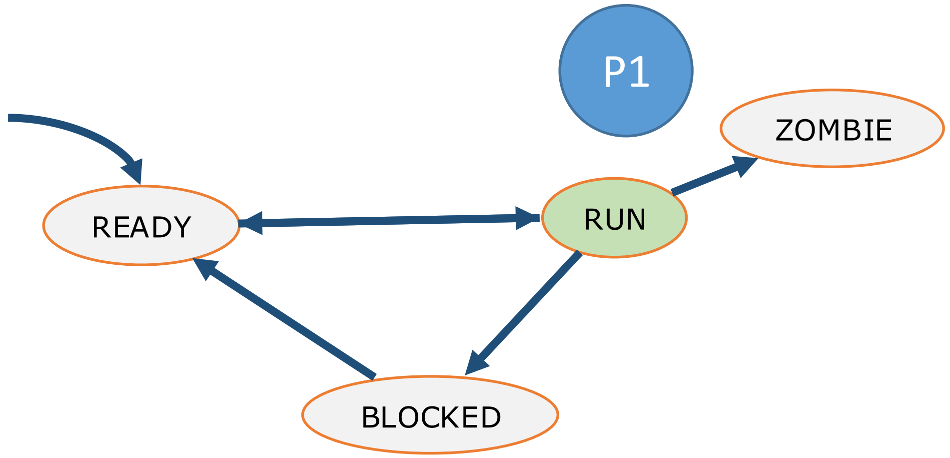
- The current process creates a child process
  - It is the base of the whole system process hierarchy
  - The child process is a clone of its parent
    - Most of the content is inherited
      - Such as memory regions, I/O devices, register file values
    - Some characteristics are not inherited
      - Such as PID, PPID (Parent's PID), stats (use of CPU…)

- Both processes keep executing from the very next instruction
- But both receive different return values
  - The parent receives the PID of the child process
  - The child receives 0

Example: Fork in Python

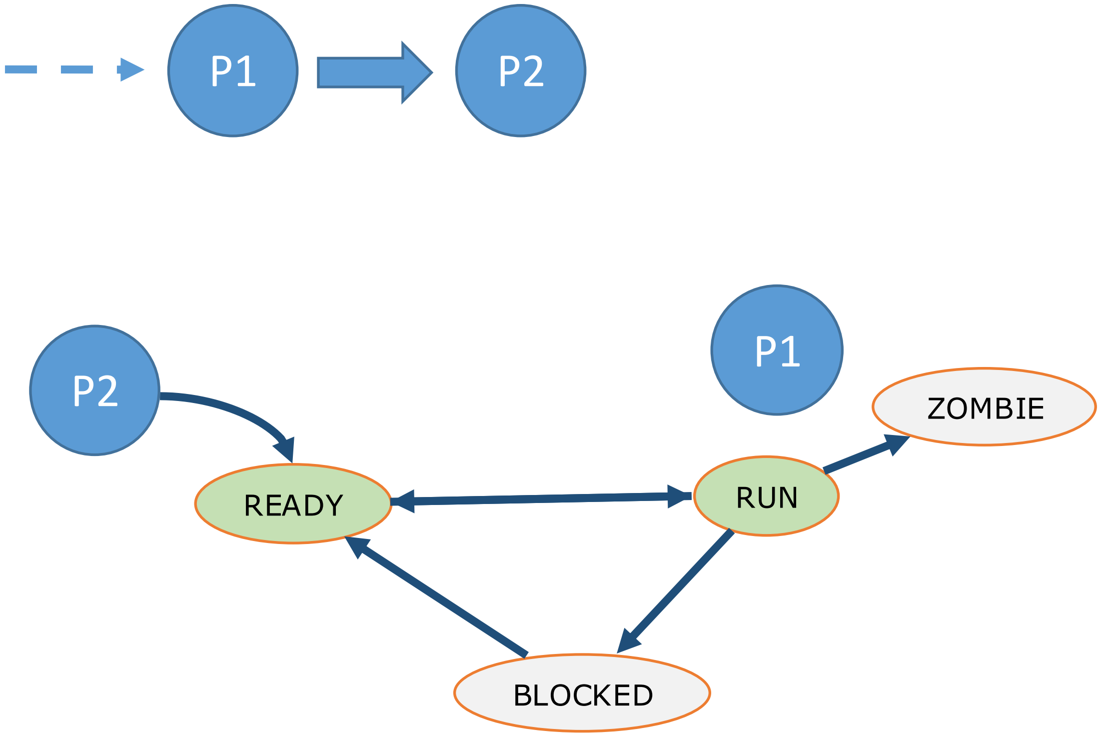
```
import os
count = 0
ret = os.fork()
if ret == 0:
  count -=1
  print("Child with counter = ",count)
else:
  count += 1
  print("Parent with counter = ",count)
```





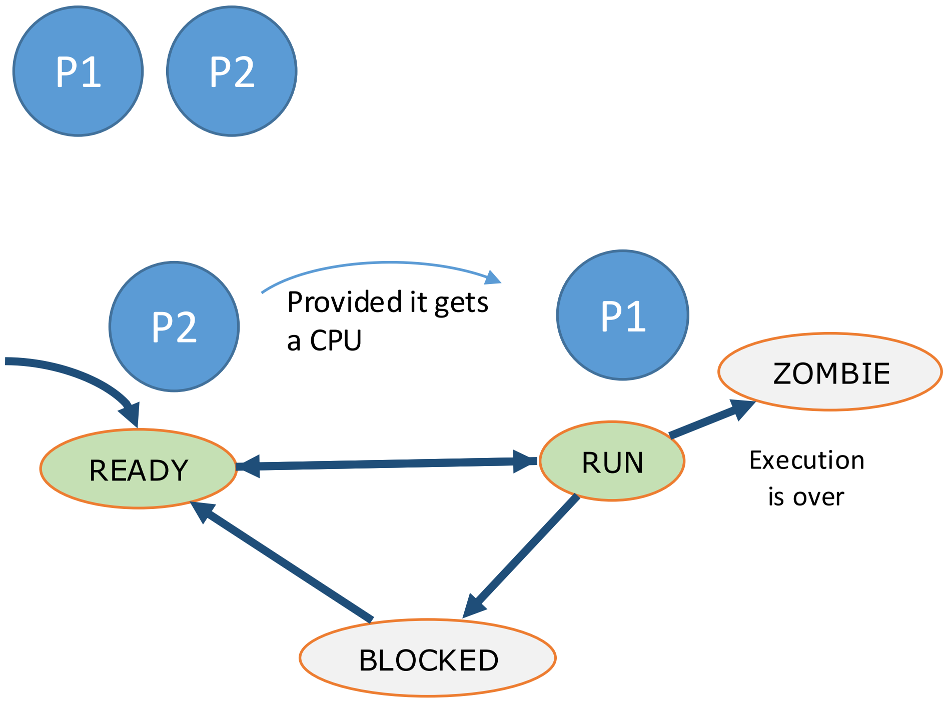
Example: Fork in Python

```python
import os
count = 0
ret = os.fork()
if ret == 0:
  count -=1
  print("Child with counter = ",count)
else:
  count += 1
  print("Parent with counter = ",count)
```



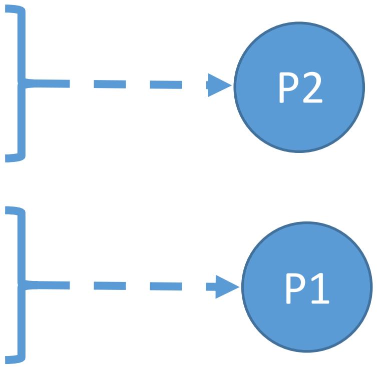
Example: Fork in Python

```python
import os
count = 0
ret = os.fork()
if ret == 0:
  count -=1
  print("Child with counter = ",count)
else:
  count += 1
  print("Parent with counter = ",count)
```



Example: Fork in Python

```
import os
count = 0
ret = os.fork()
if ret == 0:
  count -=1
  print("Child with counter = ",count)
else:
  count += 1
  print("Parent with counter = ",count)
```



Output

Child with count = -1

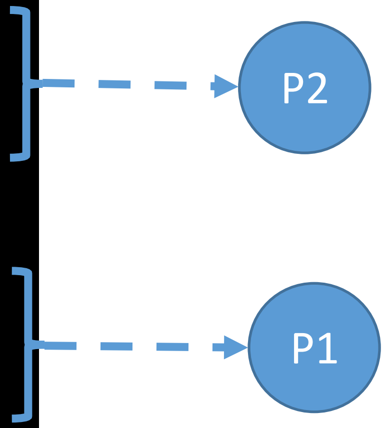
Parent with counter = 1

Notes:

- Memory regions are not shared between the processes
- Concurrent/parallel executions are possible

Example: Fork in C

```c
#include <unistd.h> // fork
#include <stdio.h> // printf
int main(int argc, char *argv[]) {
   int pid, status, code, i;
   int count = 0;
   int ret = fork();
   if (ret == 0) {
      count--;
      printf("Child with counter = %d\n", count);
   }
   else {
      count += 1;
      printf("Parent with counter = %d\n", count);
   }
}
```



Output

Child with count = -1

Parent with counter = 1

Notes:

- Memory regions are not shared between the processes
- Concurrent/parallel executions are possible

Concurrent vs Sequential Process Creation

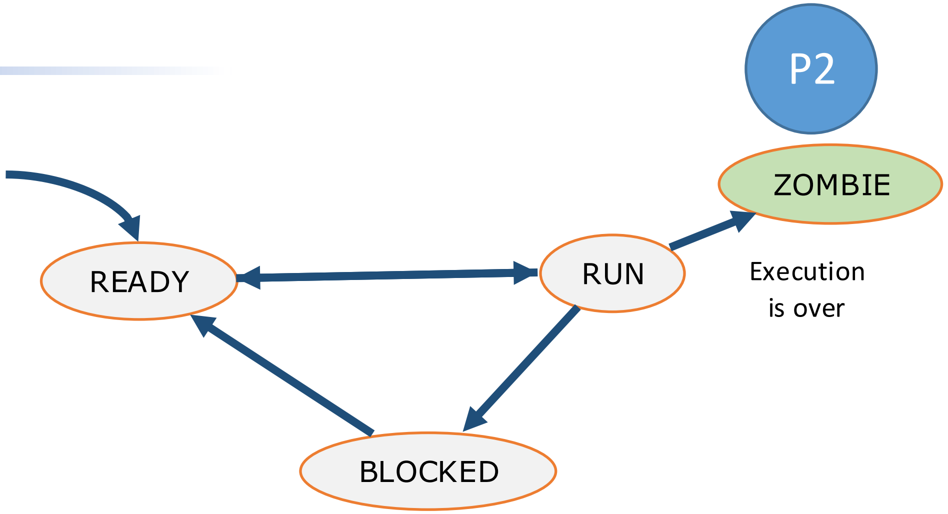
- A parent process can create multiple child processes

- Management of concurrent child process creations
  - The parent process does not wait to the death of a given child process to create the next one
  - Multiple child processes are alive at a time
    - More processes to be handled by the short term scheduler

- Management of sequential child process creations
  - The parent process waits to the death of a given child process before creating the next one
  - Only one child process is alive at a time
    - Only one additional child process to be handled by the short term scheduler

End of process execution

`sys.exit([arg])`



- The process ends the execution
  - It turns to Zombie status
    - All resources are released (e.g. memory),
    - but the PCB (PID and return value) is preserved
  - Parameters:
    - An integer* value that is the return value of the process execution (it is truncated to 1 Byte)
- The parent process has to release the *zombie* child process
  - Until that time, the PCB still exists and thus its PID

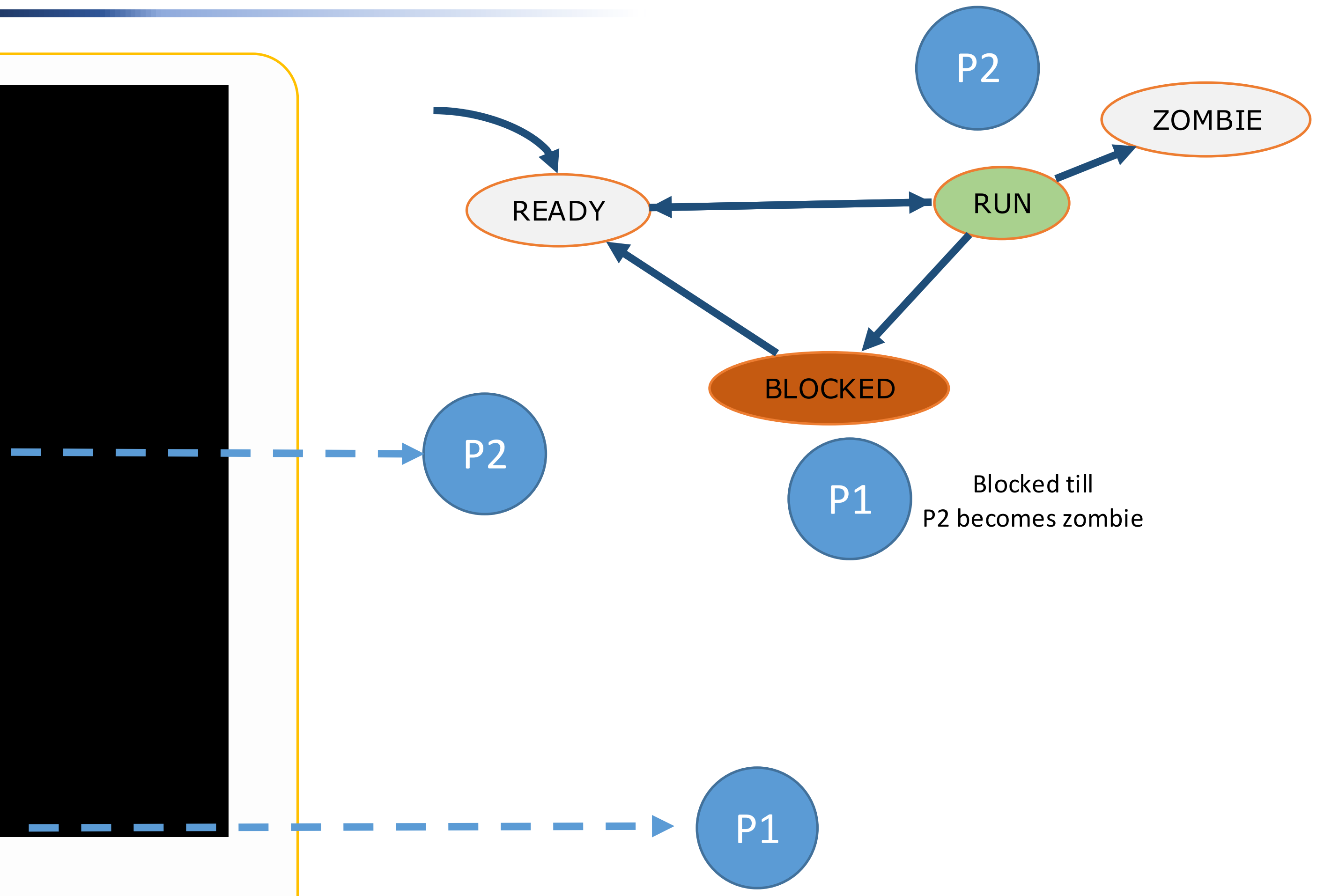
Wait for a child process to finish

[*pid*, *status*] = os.wait()

- The parent process (caller) releases a zombie child process
  - Returns the PID of the released child process and the exit STATUS
    - **Status** is updated to hold the return value of the child process (or event that involved its finalization) in the high byte of a 16-bit number
    - **Release the zombie child process** means to release the PCB, PID and related structures
- The behavior
  - If there are child processes
    - If there is a zombie child process that matches with the “pid” parameter, it is released
    - Otherwise the parent process (caller) is blocked → this is a **blocking system call**
  - If there are no child processes, returns “-1”

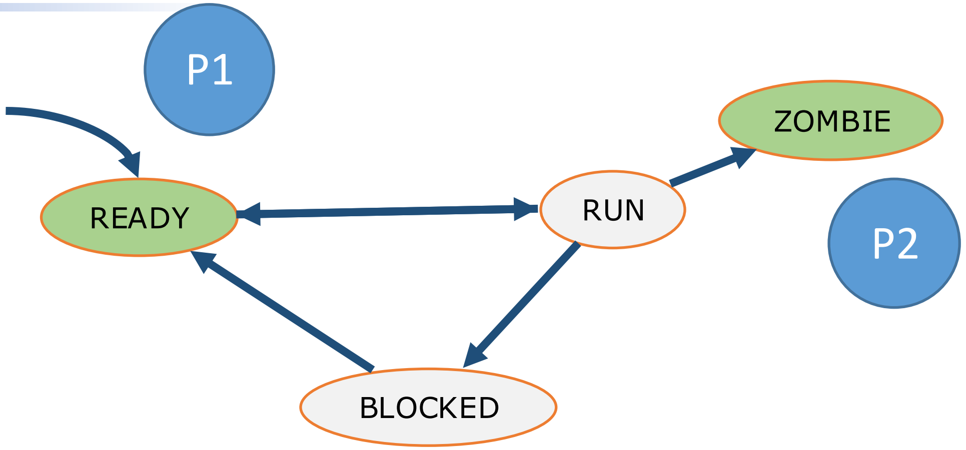
Example: exit & wait in Python

```
import os,sys
count = 0
ret = os.fork()

if ret == 0:
  print("Child with counter = ",count)
  for i in range (count, 255):
    count +=1
  sys.exit(count)

else:
  count += 1
  print("Parent with counter = ",count)
[pid,status] = os.wait()
```



Example: exit & wait in Python

```python
import os,sys
count = 0
ret = os.fork()

if ret == 0:
  print("Child with counter = ",count)
  for i in range (count, 255):
    count +=1
  sys.exit(count)

else:
  count += 1
  print("Parent with counter = ",count)
[pid,status] = os.wait()
```



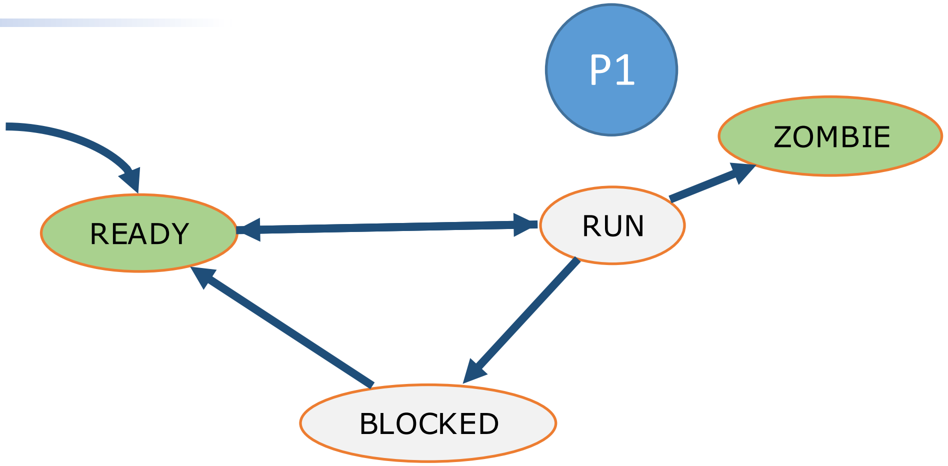
P2

count is 255 (0x**ff**)
exit (255);

P1

Example: exit & wait in Python

```python
import os,sys
count = 0
ret = os.fork()

if ret == 0:
  print("Child with counter = ",count)
  for i in range (count, 255):
    count +=1
  sys.exit(count)

else:
  count += 1
  print("Parent with counter = ",count)
[pid,status] = os.wait()
...
```



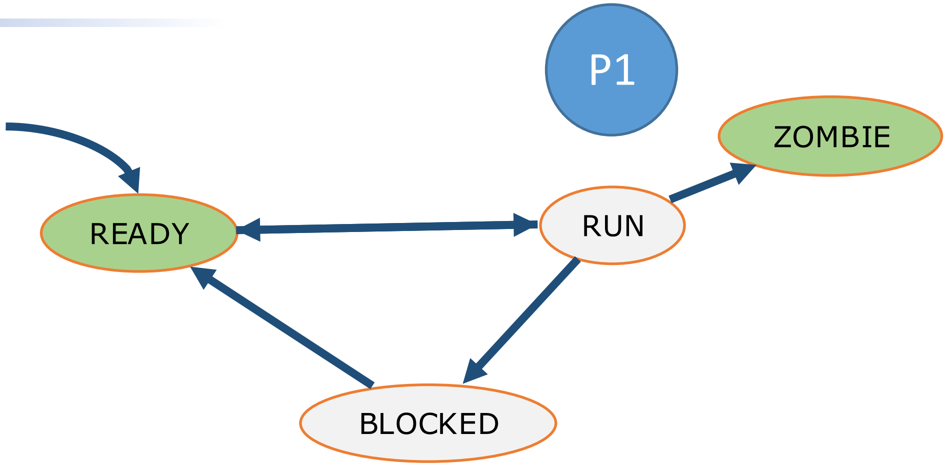
Status is 65280 (0xff00)

See check_status(pid, status)

Example: exit & wait in C

```
#include <unistd.h> // fork
#include <stdio.h> // printf
#include <sys/wait.h> // wait
#include <stdlib.h> // exit
int main(int argc, char * argv[]) {
   int i, pid, status, code;
   int count = 0;
   int ret = fork();
   if (ret == 0) {
      printf("Child with counter = %d\n",count);
      for (i=count;i<255;i++) {
         count ++;
      }
      exit(count);
   }
   else {
      count += 1;
      printf("Parent with counter = %d\n",count);
      pid = wait(&status);
      if WIFEXITED(status) {
         code = WEXITSTATUS(status);
         printf("Child %d is dead: %d\n",pid,status);
      }
   }
}
…
```

... check_status(pid, status)



P1

Status is 65280 (0xff00)

Example: exit conventions

- Error codes in exit(…) follow some common conventions
  - Code 0: program exited successfully
  - Code 1
    - Minor issues, e.g., grep returns 1 if no matching lines are found in any files
    - Errors occurred, e.g., find
  - Code 2 and above
    - Errors occurred, e.g. grep could not open at least one of the files provided
  - Usually, no negative numbers are returned

Execute a new program

`os.execlp(file, arg0, arg1, ...)`

`os.execvp(file, args)`

- Current process replaces the program (file) that is executing
  - A whole new memory contents and register values are loaded from “filename”
  - It performs dynamic linking, if necessary, and starts the program from its entry point
  - Parameters:
    - filename: indicates the name of the program to be loaded and executed (PATH is used to find it)
    - argX: hold the command line arguments for the program to be executed
    - args[]: same as argX, but in array format.
- Behavior
  - If the new program can be found, loaded and started, it never returns
    - **Once it is replaced, the previous memory contents (e.g. code, data) are not there any more**
  - On Unix, the new executable is loaded into the current process, and will have the same process id as the caller. In Python, errors will be reported as OSError exceptions.
    - E.g. the “filename” is wrong, the user has no permission to execute the “filename”, etc.

Example

- Child process replaces its code by the “ls” program
- Parent process waits until “ls” finishes

```
import os,sys

pid = os.fork()
if pid == 0:
  os.execlp("ls","-l", "-a")
else:
  os.wait()
print("Parent about to finish")
```

- [Relation between fork and exec]()

Interprocess communication (IPC)

Inter Process Communication (IPC)

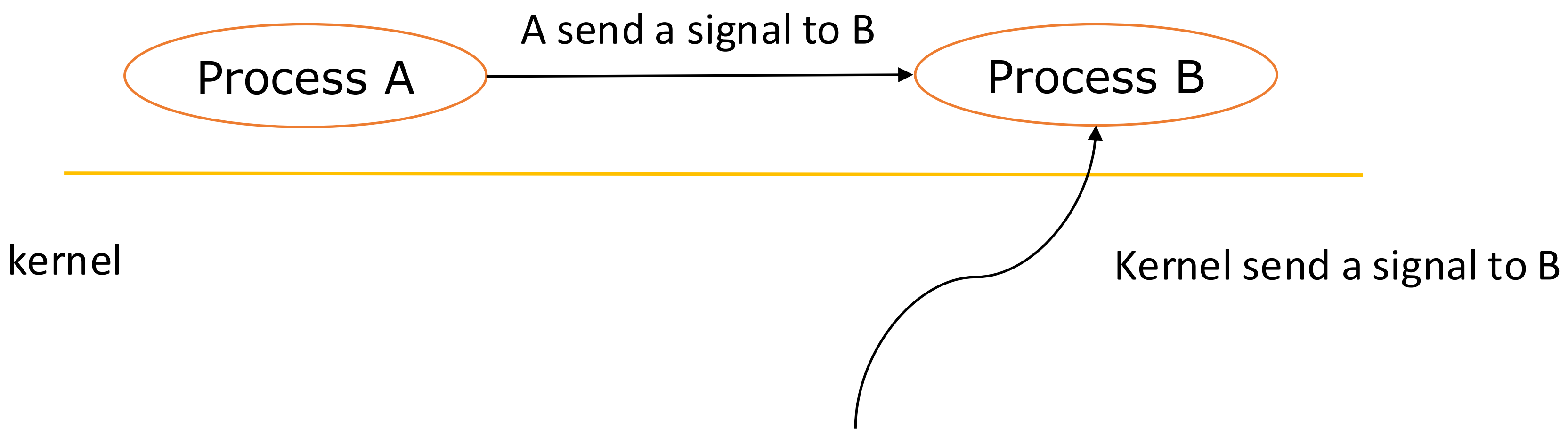
- A complex problem can be solved with several processes that cooperates among them. Cooperation means **communication**
  - Data communication: sending/receiving data
  - Synchronization: sending/waiting for events
- There are two main models for data communication
  - Shared memory between processes
    - Processes share a memory area and access it through variables mapped to this area
      - This is done through a system call, by default, memory is not shared between processes
  - Message passing (Unit 4)
    - Processes uses some special device to send/receive data
- We can also use regular files, but the kernel doesn't offer any special support for this case

IPC in Linux

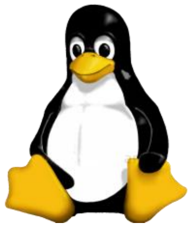
- **Signals – Events send by processes belonging to the same user or by the kernel**
- **Pipes, aka FIFOs: special devices designed for process communication. The kernel offers support for process synchronization** (Unit 4)
- Sockets – Similar to pipes but it uses the network
- Shared memory between processes – Special memory area accessible for more than one process

Signals: idea

- Signals: notification that an event has occurred
- Signals received by a process can be sent by the kernel or by other processes of the same user



Type of signals and management (I)



- Each type of event has an associated signal
  - Type of events and associated signals are defined by the kernel
    - The type of signal is a number, but there exists constants that can be used inside programs or in the command line
  - There are two signals that are not associated to any event, so the programmer can assign any meaning to them → SIGUSR1 y SIGUSR2
- Each process has associated a management to each signal
  - Default managements
  - A process can *catch* (change the associated management) to all type of signal except SIGKILL and SIGSTOP

Type of signals and management (2)

- Some signals

Name	Action	Event
SIGCHLD	IGNORE	Child stopped or terminated
SIGCONT	CONT	Continue if stopped
SIGSTOP	STOP	Stop a process
SIGINT	END	Interrupted from the keyboard (Ctrl-C)
SIGALRM	END	timer programmed by alarm has expired
SIGKILL	END	Finish the process
SIGSEGV	CORE	Invalid memory access
SIGUSR1	END	Defined by the process
SIGUSR2	END	Defined by the process

- Main uses in this course:
  - Process synchronization
  - Time control (alarms)

Type of signals and management (3)

- Reaction of a process to a signal delivering is similar to the reaction to an interrupt:
  - When a process receives a signal, it stops the code execution, executes the management associated to that signal and then (if it survives) continues with the execution of the code.
- Processes can block/unblock the delivery of each signal except SIGKILL and SIGSTOP (signals SIGILL, SIGFPE and SIGSEGV cannot be blocked when they are generated by an exception).
  - When a process blocks a signal, if that signal is sent to the process it will not be delivered until the process unblocks it.
    - The system marks the signal as pending to be delivered
    - Each process has bitmap of pending signals: it only can remember one pending delivery for each type of singal
  - When a process unblocks a signal, it will receive the pending signal and will execute the associated management

Signals basic interface

- Send a signal
  - `os.kill(`*`pid, sig`*`)`
- Catch a signal
  - `signal.signal(`*`signalnum, handler`*`)`
- Timer setting
  - `signal.alarm(`*`time`*`)`
  - To send a `signal.SIGALRM` to the process self
- Variables defined in the [signal](https://docs.python.org/3/library/signal.html) module:
  - https://docs.python.org/3/library/signal.html
  - Check also `man 7 signal`

Interface: send a signal

- `os.kill(`*`pid`*`, `*`sig`*`)`
  - Send signal *sig* to the process *pid*.
  - Constants for the specific signals available on the host platform are defined in the signal module.

```
import os, signal

os.kill(os.getppid(), signal.SIGUSR1)
```

Interface: catch a signal

- `signal.signal(`*`signalnum, handler`*`)`
  - Set the handler for signal signalnum to the function handler.
    - handler can be a callable Python object taking two arguments (signal number and current stack frame)
    - Or one of the special values signal.SIG_IGN or signal.SIG_DFL

```
import os, signal

def f(i):
 print ”Caught! Signum: “,i)

signal.signal(signal.SIGUSR1, f)
```

Example: signal , kill

```
#example signal-kill
import signal
def handler(signum, frame):
  print ("hi!:")
  return 0
signal.signal(signal.SIGUSR1,handler)
while True:
  pass
```

```
jfornes@tiacos:~/ProcMan$ python3 signal-kill.py &
[1] 15029
jfornes@tiacos:~/ProcMan$ kill -USR1 15029
hi!:
jfornes@tiacos:~/ProcMan$ kill -USR1 15029
hi!:
```

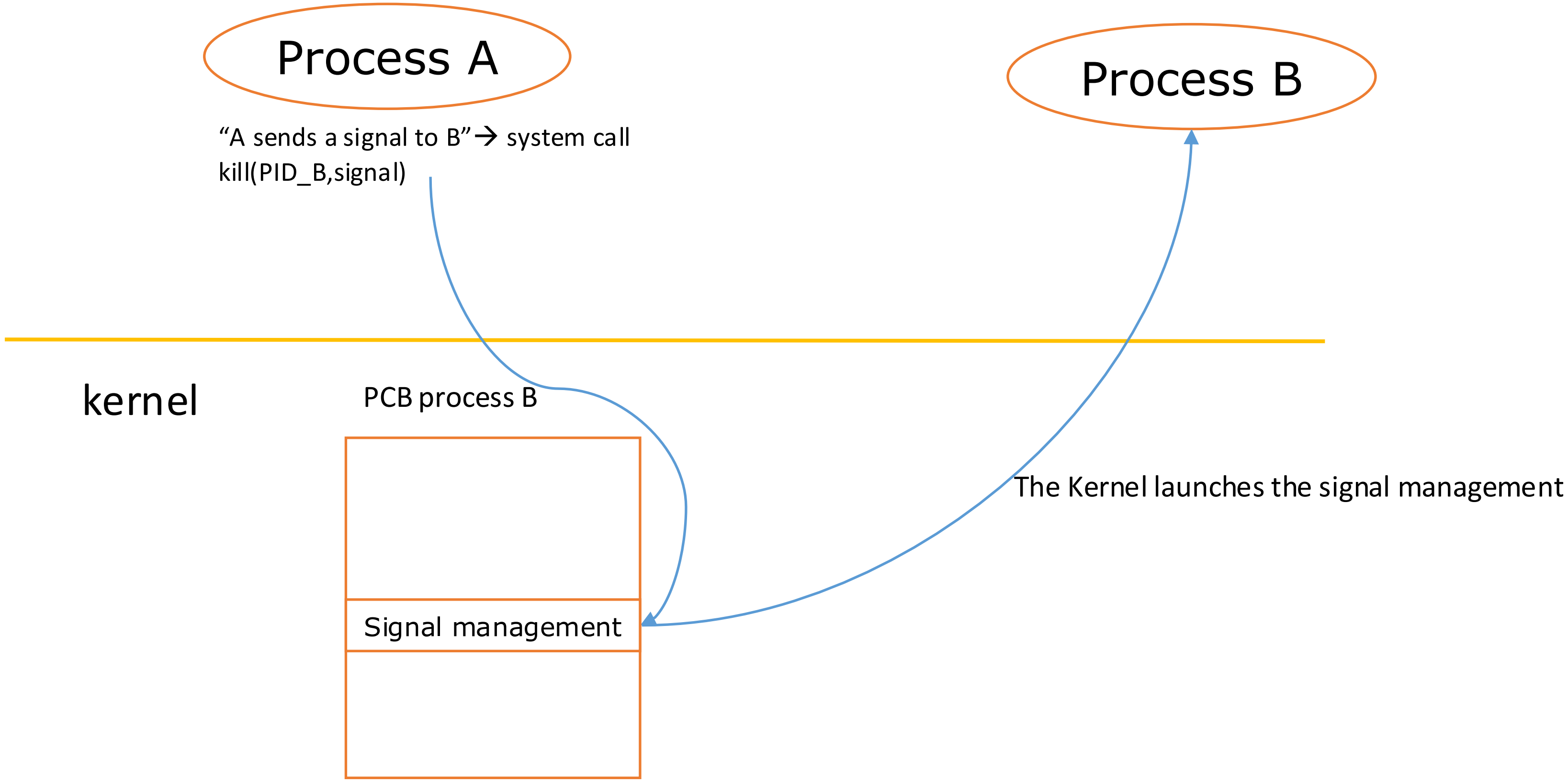
Example: signal , kill

```c
/* example signal-kill */
#include <signal.h>
#include <unistd.h>
void handler(int signum) {write(1,"hi!\n",4);}
int main(int argc, char* argv[]) {
    signal(SIGUSR1,handler);
    for(;;);
}
```

```
jfornes@CAOS:~/ProcMan$ cc -o signal-kill example_signal-kill.c
jfornes@CAOS:~/ProcMan$ ./signal-kill
[1] 15029
jfornes@CAOS:~/ProcMan$ kill -USR1 15029
hi!:
jfornes@CAOS:~/ProcMan$ kill -USR1 15029
hi!:
jfornes@CAOS:~/ProcMan$ kill -KILL 15029
[1]  + killed ./signal-kill
```

Signals: Sending and delivering

What really happens?: the kernel offers the service to pass the information.



Delivering a SIGCHLD signal

- When a child process terminates, the kernel sends a SIGCHLD to the parent
- Default disposition for SIGCHLD is ignore
- A child that terminates, but has not been waited for becomes a "zombie".
  - it will consume a slot in the kernel process table, and if this table fills, it will not be possible to create further processes.
- A parent call to `os.wait()` will block until all children have terminate
- Now, parent has a way to obtain information about the child without blocking

Delivering a SIGCHLD signal

```python
def check_status(pid, status):
    if os.WIFEXITED(status):
        code = os.WEXITSTATUS(status)
        print("Process ", pid, "ends  with exit code ", code)
    else:
        signum = os.WTERMSIG(status)
        print("Process ", pid, "ends  signal number ", signum)
    return 0

def child_handler(signum, frame):
    [pid, status] = os.wait()
    check_status(pid, status)
    return 0

signal.signal(signal.SIGCHLD, child_handler)
pid = os.fork()
...
```

Delivering a SIGCHLD signal

```
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
#include <signal.h>
#include <stdio.h>

int check_status(int pid, int status) {
   int code, signum;
   if WIFEXITED(status) {
      code = WEXITSTATUS(status);
      printf("Process %d ends because an exit with exit code %d\n", pid,code);
   }
   else {
         signum = WTERMSIG(status);
         printf("Process %d ends because a sginal number %d",pid, signum);
   }
   return 0;
}
void child_handler(int signum) {
   int pid, status;
   pid = waitpid(-1,&status,WNOHANG);
   check_status(pid, status);
}
int main(int argc, char* argv[]) {
   int pid;
   signal(SIGCHLD, child_handler);
```

Relation with fork and exec

- FORK: **new process**
  - Child inherits from parent
    - the signal table
    - the mask of blocked signals
  - Child resets
    - The bitmap of pending events
    - Timers
  - Events and timers are associated to a particular pid (the pid of the parent) and children are new processes with new pids.
- EXECLP: **same process**, new image
  - Process keeps
    - The bitmap of pending events
    - The mask of blocked signals
    - Timers
  - Process resets
    - The signal table→ the code of the process is different so the handle code of the signals is set to SIG_DFL

Error control

Error control

- It is extremely important to check for errors
  - On system calls
  - On library calls
- Manual pages describe the way system/library routines return errors

RETURN VALUE -- fork

On success, the PID of the child process is returned in the parent, and 0 is returned in the child. On failure, -1 is returned in the parent, no child process is created, and errno is set appropriately.

RETURN VALUE -- exit

These functions do not return.

Error control

- System calls usually return -1 on error and
  - Set the **errno** variable to contain the code of the specific error

RETURN VALUE -- wait

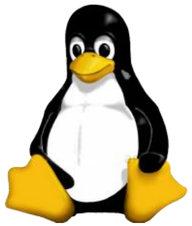
wait(): on success, returns the process ID of the terminated child; on error, -1 is returned.

ERRORS

ECHILD The process specified by pid does not exist or is not a child of the calling process.

EINVAL The options argument was invalid.

Error control



►Common UNIX/Linux error codes

/usr/include/asm-generic/errno-base.h

```
#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 */
```

Error control

- Although in Python we could use the negative return value to handle low level errors, it is much better to use exceptions.
- Errors detected during execution are called *exceptions* and are not unconditionally fatal.
- If the exception is not handled by the program, it results in error messages:

```
>>> 10 * (1/0)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
```

Handling exceptions

- Each module defines its own exceptions
- In this unit we have to deal with OSError
  - *exception* OSError(*errno*, *strerror*[, *filename*[, *winerror*[, *filename2*]]])
  - This exception is raised when a system function returns a system-related error
  - *errno,* a numeric error code from the C variable errno.
- General case:

```
try:
   # syscall
except OSError:
   # manage error
else:
   # after syscall works
finally:
   # anyway
```

Handling exceptions

- Exemple:
  - A parent process creates a child and just after that waits for it
  - Is this code ok?

```
try:
    pid = os.fork()
    os.wait()
except OSError as err:
    print("OS error: {0}".format(err))
```

Handling exceptions

- Exemple:
  - A parent process creates a child and just after that waits for it
  - Is this code ok?

```
try:
    pid = os.fork()
    os.wait()
except OSError as err:
    print("OS error: {0}".format(err))
```

```
jfornes@tiacos:~/ProcMan$ python3 error_control.py
OS error: [Errno 10] No child processes
```

Handling exceptions

- Raising Exceptions
  - With the statement `raise` we can
    - force a specified exception to occur
    - Caught the exception, but not handle it
- Model-View-Controller pattern
  - Who tries?
    - The controller
  - Who raises the exception?
    - The model
  - Who prints the error message?
    - The view

Handling exceptions

- A `try` statement may have more than one except clause, to specify handlers for different exceptions.

```
try:
    pid = os.fork()
except OSError as err:
    print("OS error: {0}".format(err))
except:
    print ("Unknown error",sys.exc_info()[0])
    raise
else:
    if pid == 0:
        print ("Child ", os.getpid())
        sys.exit(0)
~
```

Error control

►Sample code to manage errors

```
#include <sys/wait.h>
#include <errno.h>
#include <stdlib.h>
#include <stdio.h>
int main() {
  int status;
  pid_t pid, mychild;

 ... mychild = ...

  pid = waitpid(mychild, &status, 0);
  if (pid < 0) {
    perror ("waitpid");
    exit(1);     // optional?
  }
...
```

If the pid returned is -1 ...

perror formats the error message:

waitpid: No child processes

- If the application cannot continue, issue the exit(...)
- If the application can continue, the user will just get the error message

Exercises

Exercises

- Analyse this code

```
try:
  pid = os.fork()
  print(“Hello”)

except OSError:
  print(“Error”)
```

- Output if fork success?
- Output if fork fails?
- Try it!!

Exercises

- Analyse this code

```
pid = fork();
printf("Hello");

if (pid==-1)
        printf("Error");
```

- Output if fork success?
- Output if fork fails?
- Try it!!

Exercises

- How many processes are created by this code?

```
os.fork()
os.fork()
os.fork()
```

- And by this one?

```
for _ in range (0,8):
  os.fork()
```

Exercises

- How many processes are created by this code?

```
fork();
fork();
fork();
```

- And by this one?

```
for (int i=0;i<8;i++)
  fork();
```

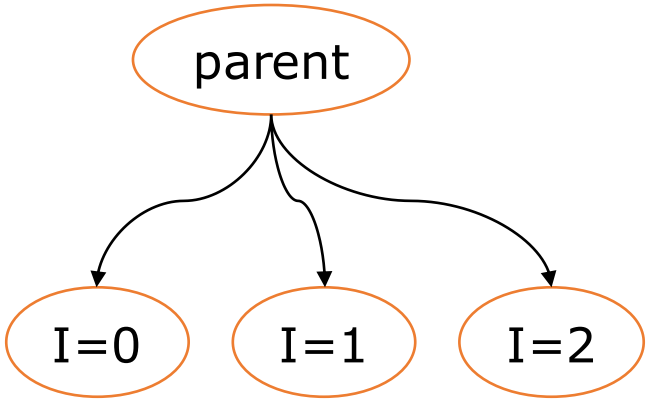
Exercises (exam)

- Draw the processes hierarchy
  - with `sys.exit(0)`
  - without `sys.exit(0)`

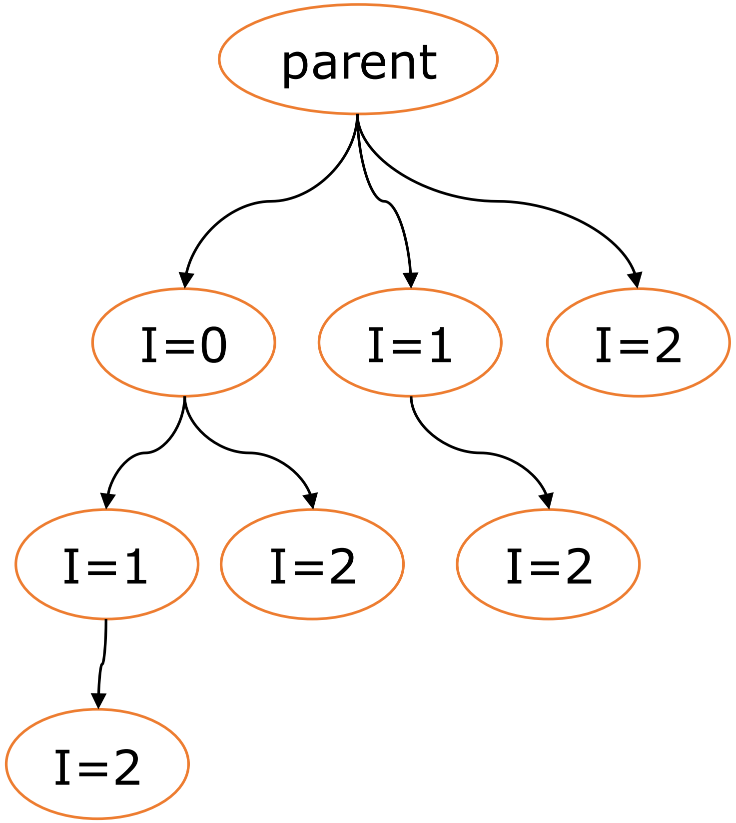
```
import os, sys
def work():
  print("My pid is ", os.getpid())
  sys.exit(0)

for _ in range (0,3):
  pid = os.fork()
  if pid ==0:
    work()
while True:
  pass
```

Process hierarchy

With exit system call

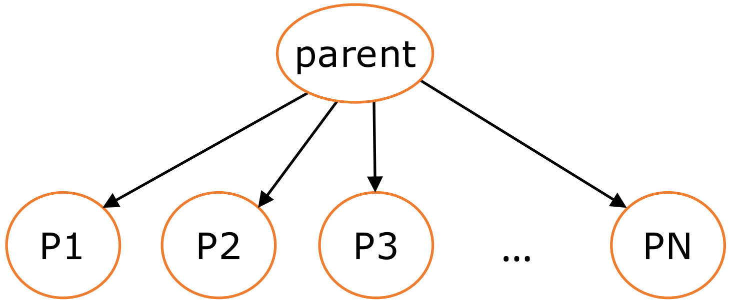


Without exit

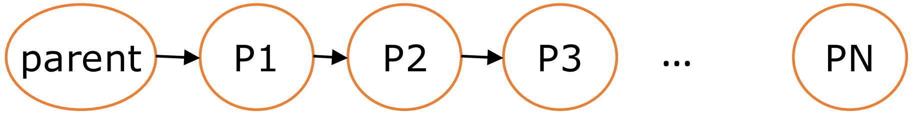


homeworks

- Write a Python program that creates this process scheme:



- Modify the code to generate this new one :



Bibliography

- Operating system concepts, Global edition. Hoboken: John Wiley & Sons, 2019.
  - A. Silberschatz, P. B. Galvin, and G. Gagne
  - https://discovery.upc.edu/permalink/34CSUC_UPC/18e7aks/alma991004148389706711
    - Introduces the presented concepts about OS
- Tools: Performance Analysis of Parallel Python Applications
  https://www.sciencedirect.com/science/article/pii/S1877050917307962
- Python documentation
  - https://docs.python.org/3/library/os.html
  - https://docs.python.org/3/library/sys.html
  - https://docs.python.org/3/library/signal.html
  - https://docs.python.org/3/tutorial/errors.html
- C documentation
  - The C Programming Language
    - Kernighan and Ritchie
    - https://discovery.upc.edu/permalink/34CSUC_UPC/1q393em/alma991000708719706711
  - Linux man pages