# Operating Systems and Concurrency

## Lecture 3: Processes

G52OSC/COMP2007



Geert De Maere
(Isaac Triguero)
{Geert.DeMaere,Isaac.Triguero}@Nottingham.ac.uk

University Of Nottingham
United Kingdom



2018

- The **hardware** and the **operating system interact** closely



- The **operating system** must have in depth **knowledge** of the **hardware**
- Examples of **interrupts** and **address translation**

- Introduction to **processes** and their **implementation**



- Process **states** and state **transitions**
- **System calls** for process management

# Processes
## Definition



- The simplified definition: *"a process is a **running instance** of a program"*
  - A program is **passive** and "sits" on a disk
  - A process has **control structures** associated with it, may be **active**, and may have **resources** assigned to it (e.g. I/O devices, memory, processor)



- A process is registered with the OS using its **"control structures"**: i.e. an entry in the OS's **process table** to a **process control blocks** (PCB)



- The **process control block** contains all information necessary to **administer the process** and is **essential** for **context switching** in **multiprogrammed systems**

# Processes

## Memory Image of Processes

- A **process' memory image** contains:
  - The program **code** (could be shared between multiple processes running the same code)
  - A **data** segment, **stack** and **heap**
- Every process has its own **logical address space**, in which the **stack** and **heap** are placed at **opposite sides** to allow them to grow
- Some OS'es use **address space layout randomisation**

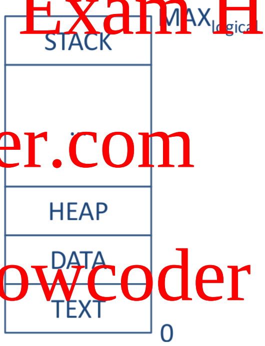
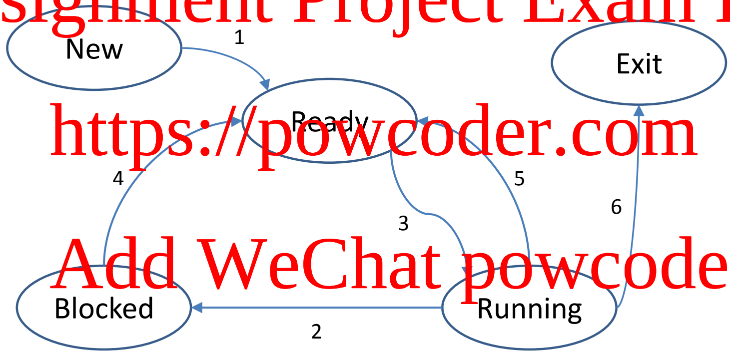



Figure: Representation of a process in memory

# Process States and Transitions

## Diagram



# Process States and Transitions
## States



- A **new** process has just been created (has a PCB) and is waiting to be admitted (it may not yet be in memory)
- A **ready** process is waiting for CPU to become available (e.g. unblocked or timer interrupt)
- A **running** process "owns" the CPU
- A **blocked** process cannot continue, e.g. is waiting for I/O
- A **terminated** process is no longer executable (the data structures - PCB - may be temporarily preserved

# Process States and Transitions
## States



- A **new** process has just been created (has a PCB) and is waiting to be admitted (it may not yet be in memory)
- A **ready** process is waiting for CPU to become available (e.g. unblocked or timer interrupt)



- A **running** process "owns" the CPU
- A **blocked** process cannot continue, e.g. is waiting for I/O



- A **terminated** process is no longer executable (the data structures - PCB - may be temporarily preserved
- A **suspended** process is swapped out (not discussed further)

# Process States and Transitions

## Transitions

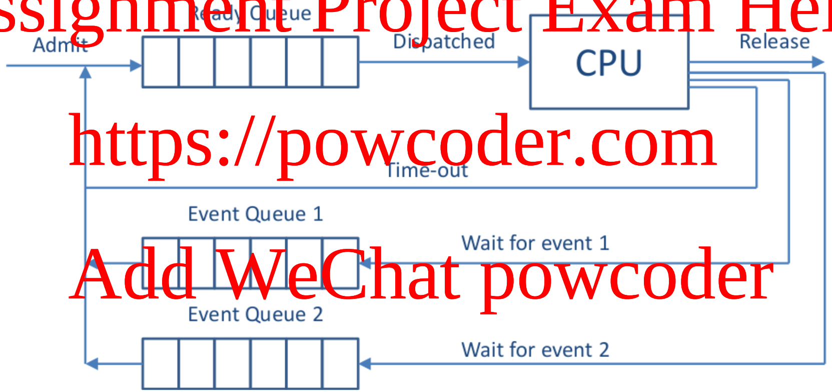


- State transitions include:
  1. **New** → **ready**: admit the process and commit to execution
  2. **Running** → **blocked**: e.g. process is waiting for input or carried out a system call
  3. **Ready** → **running**: the process is selected by the **process scheduler**
  4. **Blocked** → **ready**: event happens, e.g. I/O operation has finished
  5. **Running** → **ready**: the process is preempted, e.g., by a **timer interrupt** or by **pause**
  6. **Running** → **exit**: process has finished, e.g. program ended or exception encountered
- The **interrupts/traps/system calls** lie on the basis of the transitions

Exam 2013-2014: List the 5 process states and explain the transitions between them

# Process States and Transitions

## OS Queues



# Context Switching
## Multi-programming



- Modern computers are **multi-programming** systems
- Assuming a **single processor system**, the instructions of individual processes are executed **sequentially**
  - Multi-programming goes back to the "**MULTICS**" age
  - Multi-programming is achieved by **alternating** processes and **context switching**
  - **True parallelism** requires **multiple processors**

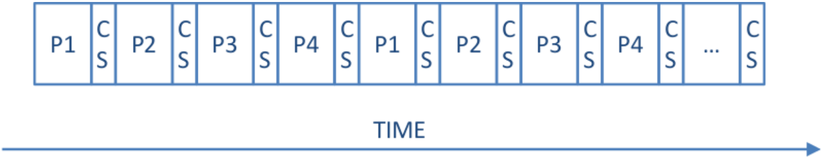




# Context Switching
## Multi-programming (Cont'ed)

- When a **context switch** takes place, the system **saves the state** of the old process and **loads the state** of the new process (creates **overhead**)
  - **Saved** ⇒ the process control block is **updated**
  - **(Re-)started** ⇒ the process control block **read**
- A **trade-off** exists between the length of the **time-slice** and the **context switch time**
  - **Short time slices** result in **good response times** but **low effective "utilisation"**
    - e.g.: 99 * (1 + 1) = 198ms
  - **Long time slices** result in **poor response times** but **better effective "utilisation"**
    - e.g.: 99 * (100 + 1) = 9999ms

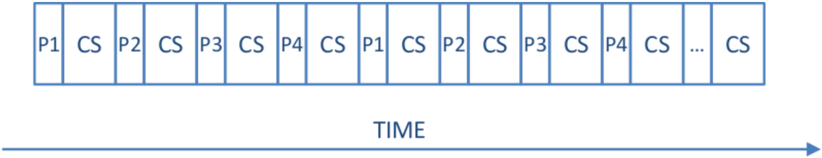




# Context Switching
## Multi-programming (Cont'ed)

- When a **context switch** takes place, the system **saves the state** of the old process and **loads the state** of the new process (creates **overhead**)
  - **Saved** ⇒ the process control block is **updated**
  - **(Re-)started** ⇒ the process control block **read**
- A **trade-off** exists between the length of the **time-slice** and the **context switch time**
  - **Short time slices** result in **good response times** but **low effective "utilisation"**
    - e.g.: 99 * (1 + 1) = 198ms
  - **Long time slices** result in **poor response times** but **better effective "utilisation"**
    - e.g.: 99 * (100 + 1) = 9999ms





# Context Switching

## Multi-programming (Cont'ed)



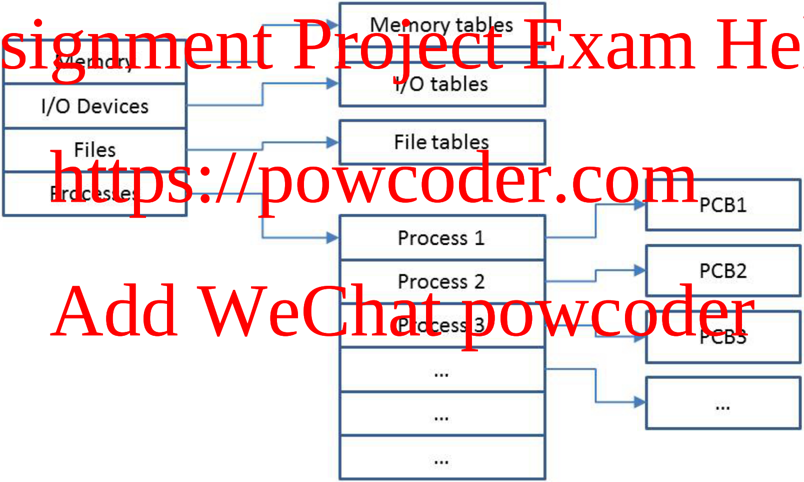
- A **process control block** contains three types of **attributes**:
  - **Process identification** (PID, UID, Parent PID)
  - **Process control information** (process state, scheduling information, etc.)
  - **Process state information** (user registers, program counter, stack pointer, program status word, memory management information, files, etc.)



- **Process control blocks** are **kernel data structures**, i.e. they are **protected** and only accessible in **kernel mode**!
  - Allowing user applications to access them directly could **compromise their integrity**
  - The **operating system manages** them on the user's behalf through **system calls** (e.g. to set **process priority**)

# Process Implementation

## Tables and Control Blocks



# Process Implementation

## Tables and Control Blocks



- An operating system **maintains information** about the status of "resources" in **tables**
  - **Process tables** (process control blocks)
  - **Memory tables** (memory allocation, memory protection, virtual memory)
  - **I/O tables** (availability, status, transfer information)
  - **File tables** (location, status)



- The **process table** holds a **process control block** for each process, allocated upon **process creation**



- Tables are maintained by the **kernel** and are usually **cross referenced**

# Context Switching
## Switching Processes

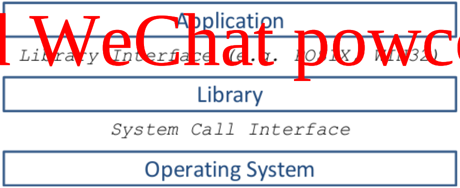


```
1. Save process state (program counter, registers)
2. Update PCB (running -> ready/blocked)
3. Move PCB to appropriate queue (ready/blocked)
4. Run scheduler, select new process
5. Update to running state in the new PCB
6. Update memory management unit (MMU)
7. Restore process
```

# System Calls
## Process Creation



- The true system calls are "**wrapped**" in the **OS libraries** (e.g. `libc`) following a well defined interface (e.g. `POSIX`, `WIN32` API)
- System calls for **process creation**:
  - Unix: **fork()** generates and exact copy of parent ⇒ **exec()**
  - Windows: **NTCreateProcess()**
  - Linux: **Clone()**



# System Calls
## Process Termination



- System calls are necessary to **notify the OS** that the **process has terminated**
  - Resources must be de-allocated
  - Output must be flushed
  - Process admin may have to be carried out
- A system calls for process termination:
  - UNIX/Linux: **exit()**, kill()
  - Windows: **TerminateProcess()**

# Fork
## Process Creation in Linux



- `fork()` creates an **exact copy** of the current process
  - The first instruction carried out by the child is the first one after the `fork` call



- `fork()` returns the **process identifier** of the child process **to the parent process** (iPID > 0)
- `fork()` **returns 0** to the **child process** (iPID = 0)

# Processes

## Process Creation in Linux



```
// PARENT CODE
#include <stdio.h>

void main(){
  int iStatus;
  int iPID = fork();
  if(iPID < 0)
  {
     printf("fork error\n");
  }
  else if(iPID == 0)
  {
     printf("hello from child\n");
     execl("/bin/ls", "ls", "-l", 0);
  }
  else if(iPID > 0)
  {
     waitpid(iPID, &iStatus, 0);
     printf("hello from parent\n");
  }
}
```

# Processes

## Process Creation in Linux



```c
// PARENT CODE
#include <stdio.h>

void main(){
  int iStatus;
  int iPID = fork();
  if(iPID < 0)
  {
     printf("fork error\n");
  }
  else if(iPID == 0)
  {
     printf("hello from child\n");
     execl("/bin/ls", "ls", "-l", 0);
  }
  else if(iPID > 0)
  {
     waitpid(iPID, &iStatus, 0);
     printf("hello from parent\n");
  }
}
```

```c
// CHILD CODE
#include <stdio.h>

void main(){
  int iStatus;
  int iPID = fork();
  if(iPID < 0)
  {
     printf("fork error\n");
  }
  else if(iPID == 0)
  {
     printf("hello from child\n");
     execl("/bin/ls", "ls", "-l", 0);
  }
  else if(iPID > 0)
  {
     waitpid(iPID, &iStatus, 0);
     printf("hello from parent\n");
  }
}
```

# Processes

## Process Creation in Linux



```
// PARENT CODE
#include <stdio.h>

void main(){
  int iStatus;
  int iPID = fork();
  if(iPID < 0)
  {
     printf("fork error\n");
  }
  else if(iPID == 0)
  {
     printf("hello from child\n");
     execl("/bin/ls", "ls", "-l", 0);
  }
  else if(iPID > 0)
  {
     waitpid(iPID, &iStatus, 0);
     printf("hello from parent\n");
  }
}
```



```
// CHILD CODE
#include <stdio.h>

void main(){
  int iStatus;
  int iPID = fork();
  if(iPID < 0)
  {
     printf("fork error\n");
  }
  else if(iPID == 0)
  {
     printf("hello from child\n");
     execl("/bin/ls", "ls", "-l", 0);
  }
  else if(iPID > 0)
  {
     waitpid(iPID, &iStatus, 0);
     printf("hello from parent\n");
  }
}
```

# Processes

## Process Creation in Linux



```c
// PARENT CODE
#include <stdio.h>

void main(){
  int iStatus;
  int iPID = fork();
  if(iPID < 0)
  {
    printf("fork error\n");
  }
  else if(iPID == 0)
  {
    printf("hello from child\n");
    execl("/bin/ls", "ls", "-l", 0);
  }
  else if(iPID > 0)
  {
    waitpid(iPID, &iStatus, 0);
    printf("hello from parent\n");
  }
}
```



```c
// CHILD CODE
#include <stdio.h>

void main(){
  int iStatus;
  int iPID = fork();
  if(iPID < 0)
  {
    printf("fork error\n");
  }
  else if(iPID == 0)
  {
    printf("hello from child\n");
    execl("/bin/ls", "ls", "-l", 0);
  }
  else if(iPID > 0)
  {
    waitpid(iPID, &iStatus, 0);
    printf("hello from parent\n");
  }
}
```

# Processes

## Process Creation in Linux



```c
// PARENT CODE
#include <stdio.h>

void main(){
  int iStatus;
  int iPID = fork();
  if(iPID < 0)
  {
    printf("fork error\n");
  }
  else if(iPID == 0)
  {
    printf("hello from child\n");
    execl("/bin/ls", "ls", "-l", 0);
  }
  else if(iPID > 0)
  {
    waitpid(iPID, &iStatus, 0);
    printf("hello from parent\n");
  }
}
```



```c
// CHILD CODE
#include <stdio.h>

void main(){
  int iStatus;
  int iPID = fork();
  if(iPID < 0)
  {
    printf("fork error\n");
  }
  else if(iPID == 0)
  {
    printf("hello from child\n");
    execl("/bin/ls", "ls", "-l", 0);
  }
  else if(iPID > 0)
  {
    waitpid(iPID, &iStatus, 0);
    printf("hello from parent\n");
  }
}
```

# Processes

## Process Creation in Linux



```c
// PARENT CODE
#include <stdio.h>

void main(){
  int iStatus;
  int iPID = fork();
  if(iPID < 0)
  {
    printf("fork error\n");
  }
  else if(iPID == 0)
  {
    printf("hello from child\n");
    execl("/bin/ls", "ls", "-l", 0);
  }
  else if(iPID > 0)
  {
    waitpid(iPID, &iStatus, 0);
    printf("hello from parent\n");
  }
}
```



```c
// CHILD CODE
#include <stdio.h>

void main(){
  int iStatus;
  int iPID = fork();
  if(iPID < 0)
  {
    printf("fork error\n");
  }
  else if(iPID == 0)
  {
    printf("hello from child\n");
    execl("/bin/ls", "ls", "-l", 0);
  }
  else if(iPID > 0)
  {
    waitpid(iPID, &iStatus, 0);
    printf("hello from parent\n");
  }
}
```

# Processes

## Process Creation in Linux



```
// PARENT CODE
#include <stdio.h>

void main(){
  int iStatus;
  int iPID = fork();
  if(iPID < 0)
  {
     printf("fork error\n");
  }
  else if(iPID == 0)
  {
     printf("hello from child\n");
     execl("/bin/ls", "ls", "-l", 0);
  }
  else if(iPID > 0)
  {
     waitpid(iPID, &iStatus, 0);
     printf("hello from parent\n");
  }
}
```

```
// CHILD CODE
#include <stdio.h>

void main(){
  int iStatus;
  int iPID = fork();
  if(iPID < 0)
  {
     printf("fork error\n");
  }
  else if(iPID == 0)
  {
     printf("hello from child\n");
     execl("/bin/ls", "ls", "-l", 0);
  }
  else if(iPID > 0)
  {
     waitpid(iPID, &iStatus, 0);
     printf("hello from parent\n");
  }
}
```

# Processes

## Process Creation in Linux



```c
// PARENT CODE
#include <stdio.h>

void main(){
  int iStatus;
  int iPID = fork();
  if(iPID < 0)
  {
    printf("fork error\n");
  }
  else if(iPID == 0)
  {
    printf("hello from child\n");
    execl("/bin/ls", "ls", "-l", 0);
  }
  else if(iPID > 0)
  {
    waitpid(iPID, &iStatus, 0);
    printf("hello from parent\n");
  }
}
```

```
// CHILD CODE
// REPLACED BY CODE FOR "/bin/ls"

total 16
-rwxrwxr-x. 1 pszgd pszgd 8648 Oct  4 15:33 a.out
-rw-rw-r--. 1 pszgd pszgd  358 Oct  6  2016 fork.c
```

# Processes

## Process Creation in Linux



```
// PARENT CODE
#include <stdio.h>

void main(){
  int iStatus;
  int iPID = fork();
  if(iPID < 0)
  {
    printf("fork error\n");
  }
  else if(iPID == 0)
  {
    printf("hello from child\n");
    execl("/bin/ls", "ls", "-l", 0);
  }
  else if(iPID > 0)
  {
    waitpid(iPID, &iStatus, 0);
    printf("hello from parent\n");
  }
}
```

```
// CHILD CODE
CHILD FINISHES
```

# Recap

## Take-Home Message



- **Definition of a process** and their **implementation** in operating systems
- **States**, state **transitions** of processes
- **Kernel structures** for processes and process management
- **System calls** for process management