Processes

OPS Lecture 3, G53OPS/G52OSC

Geert De Maere
(Jason Atkin – OSC)
Geert.DeMaere@Nottingham.ac.uk

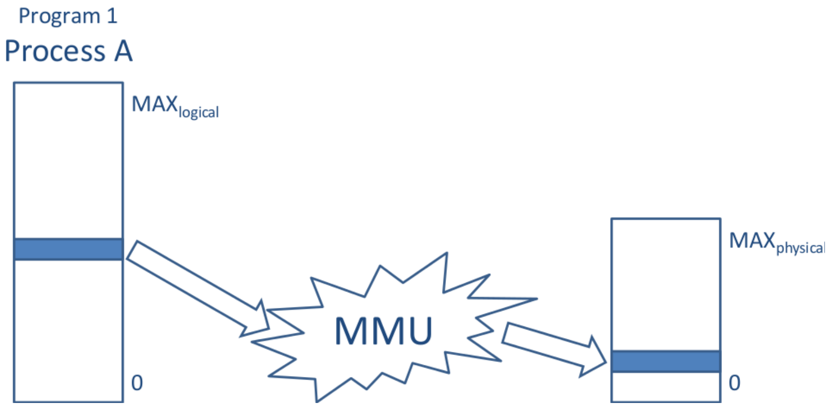
University Of Nottingham
United Kingdom

2015

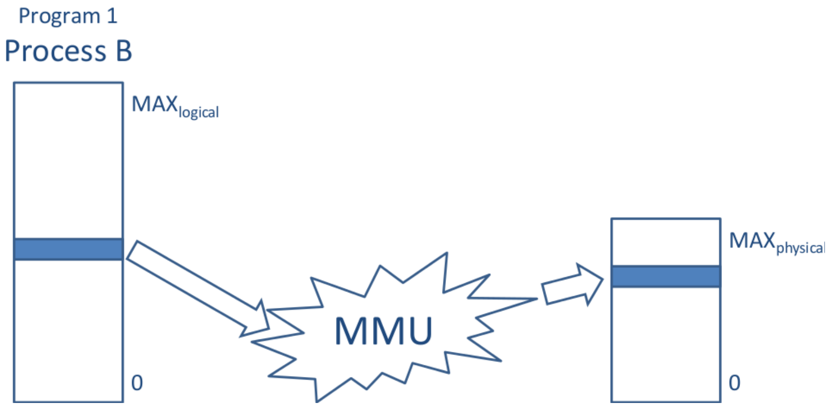
Recap
Last Lecture

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

Recap
Last Lecture



Recap
Last Lecture



Goals for Today
Overview

- 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

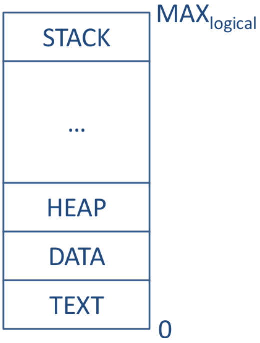
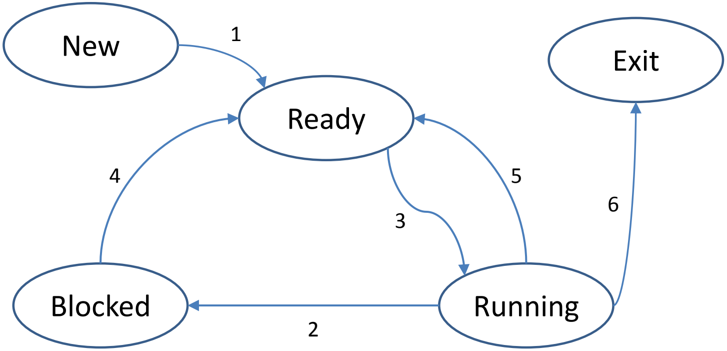


Figure: Representation of a process in memory

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)

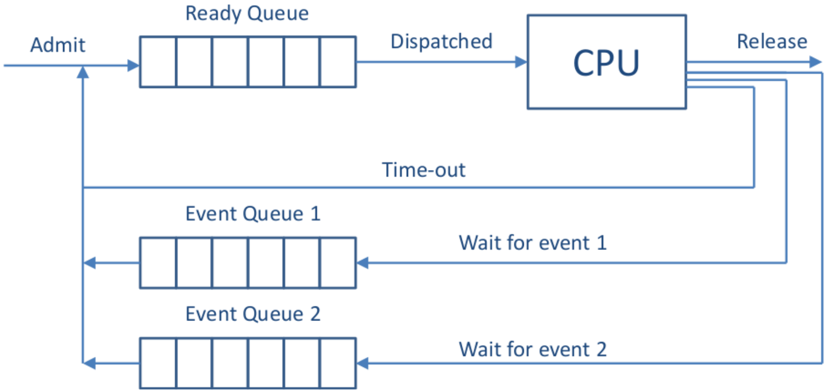
Process States and Transitions

Transitions

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

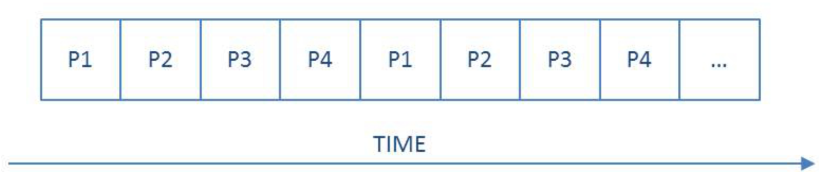
Exam 2013-2014: List the 5 process states and explain the transitions between them

Process States and Transitions
OS Queues



Context Switching
Multi-programming

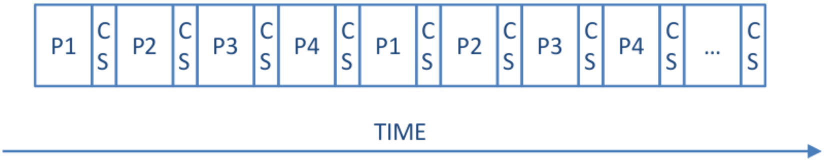
- Modern computers are **multi-programming** systems:
- Assuming a **single processor system**, the instructions of individual processes are executed **sequentially**
    - Multi-programming goes back to the **“MULTICS”** age
    - Multi-programming is achieved by **alternating** processes and **context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



Context Switching

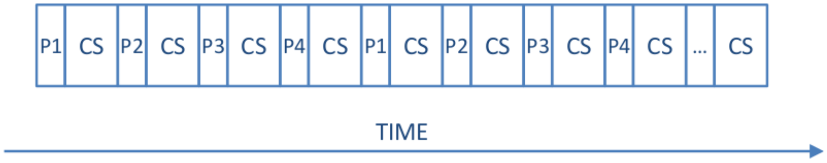
Multi-programming (Cont'ed)

- 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 **"responsiveness"** and **"overhead"**
- The OS uses **process control blocks** and a **process table** to manage processes and maintain their information



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** $\Rightarrow$ the process control block is **updated**
  - **(Re-)started** $\Rightarrow$ the process control block **read**
- A **trade-off** exists between **"responsiveness"** and **"overhead"**
- The OS uses **process control blocks** and a **process table** to manage processes and maintain their information

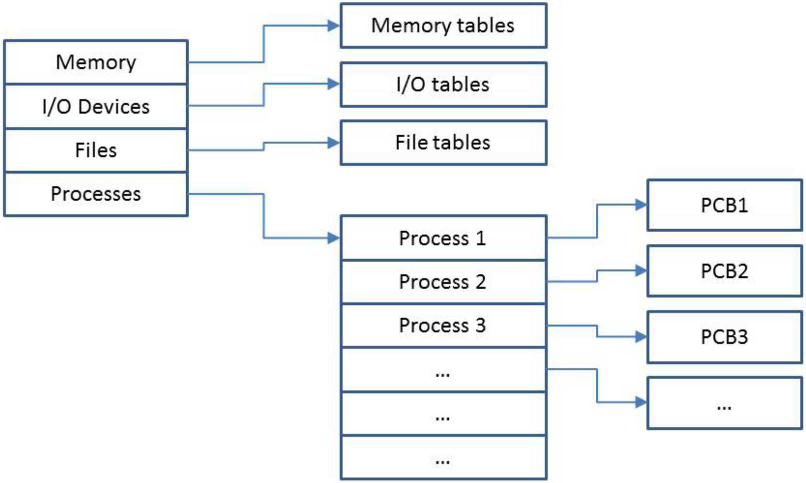


Context Switching

Multi-programming (Cont'ed)

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

Process Implementation
Tables and Control Blocks



Process Implementation

Tables and Control Blocks

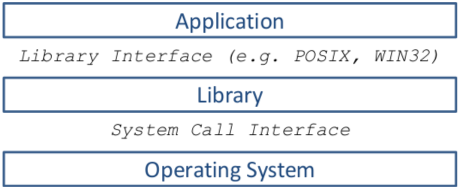
- An operating system **maintains information** about the status of "resources" in **tables**
  - **Process tables** (process control blocks)
  - **Memory tables** (memory allocation, memory protection, virtual memory)
  - **I/O tables** (availability, status, transfer information)
  - **File tables** (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)
3. Move PCB to appropriate queue (ready/blocked)
4. Run scheduler, select new process
5. Update to running state in PCB
6. Update memory structures
7. Restore process
```

System Calls
Process Creation

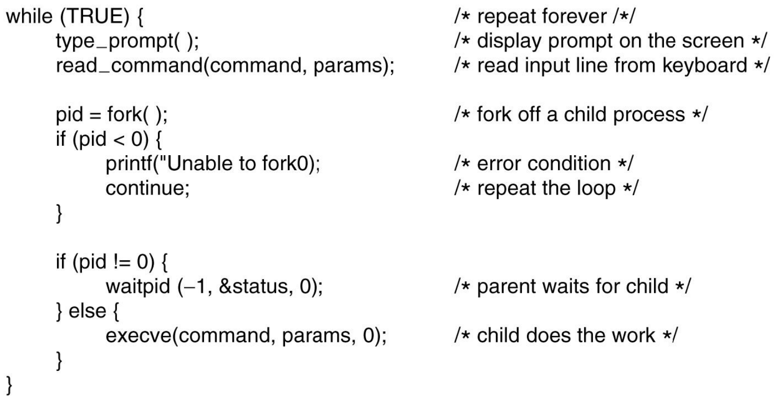
- The true system calls are “**wrapped**” in the **OS libraries** (e.g. `libc`) following a well defined interface (e.g. `POSIX`, `WIN32` API)
- System calls for **process creation**:
  - Unix: **fork()** generates and exact copy of parent $\Rightarrow$ **exec()**
  - Windows: **NTCreateProcess()**
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

Processes
Process Creation in Linux



```
while (TRUE) {                                   /* repeat forever /*/
    type_prompt( );                              /* display prompt on the screen */
    read_command(command, params);               /* read input line from keyboard */

    pid = fork( );                               /* fork off a child process */
    if (pid < 0) {
        printf("Unable to fork0);                /* error condition */
        continue;                                /* repeat the loop */
    }

    if (pid != 0) {
        waitpid (−1, &status, 0);                /* parent waits for child */
    } else {
        execve(command, params, 0);              /* child does the work */
    }
}
```

Figure: Use of the fork() and exec() system calls (Tanenbaum)

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

Next Lecture

Content

- Next Lecture: Process Scheduling