



# Sistemas de Operação / Fundamentos de Sistemas Operativos

## Processes and threads

Artur Pereira <artur@ua.pt>

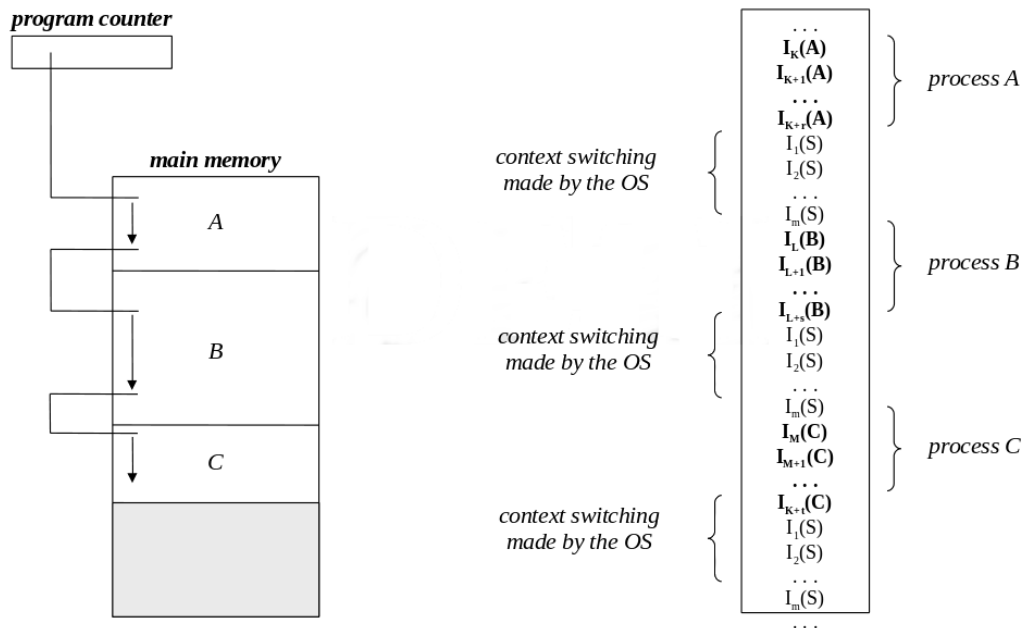
DETI / Universidade de Aveiro

## Outline

- 1 Process model
- 2 Context switching
- 3 Process control table
- 4 Process state diagram
- 5 Typical Unix state diagram
- 6 Threads and multithreading
- 7 Bibliography

# Process

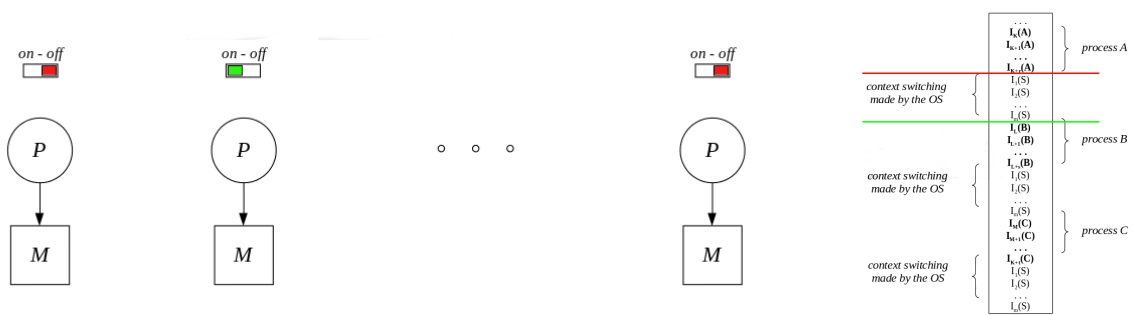
## Execution in a multiprogrammed environment



# Processes

## Process model

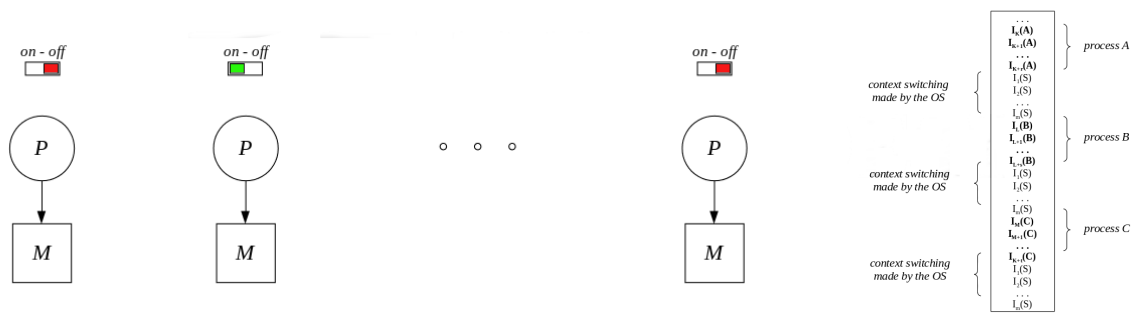
- In **multiprogramming** the activity of the processor, because it is switching back and forth from process to process, **is hard to perceive**
- Thus, it is better to assume the existence of a number of virtual processors, one per existing process
  - **number of active virtual processors  $\leq$  number of real processors**
  - **Turning off one virtual processor and turning on another, corresponds to a **context switching****



# Processes

## Process model (2)

- The **context switching**, and thus the switching between virtual processors, can occur for different reasons, possible not controlled by the running program
- Thus, to be viable, this process model requires that
  - the execution of any process must not be affected by the **instant in time** or the **location in the code** where the switching takes place
  - no restrictions are imposed on the total or partial execution times of any process



# Processes

## Context switching

- Current processors have two functioning modes:
  - supervisor mode** – all instruction set can be executed
    - is a privileged mode
  - user mode** – only part of the instruction set can be executed
    - input/output instructions are excluded as well as those that modify control registers
    - it is the normal mode of operation
- Switching from user mode to supervisor mode is only possible through an **exception** (for security reasons)
- An exception can be caused by:
  - I/O interrupt**
    - external to the execution of the current instruction
  - illegal instruction (division by zero, bus error)
    - associated with the execution of the current instruction, but not intended
  - trap instruction (software interruption)
    - associated with the execution of the current instruction, and intended

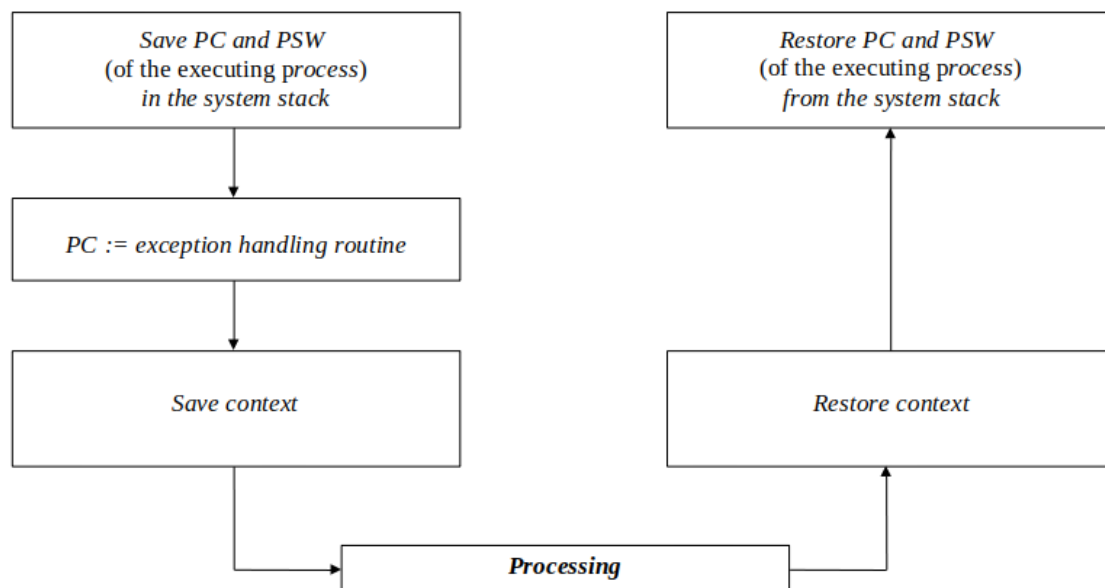
# Processes

## Context switching (2)

- The operating system should function in supervisor mode
    - in order to have access to all the functionalities of the processor
  - Thus kernel functions (including system calls) must be fired by
    - hardware (interrupt)
    - trap (software interruption)
  - This establishes a uniform operating environment: **exception handling**
- 
- **Context switching** is the process of storing the state of a process and restoring the state of another process
  - Context switching occurs necessarily in the context of an exception, with a small difference on how it is handle

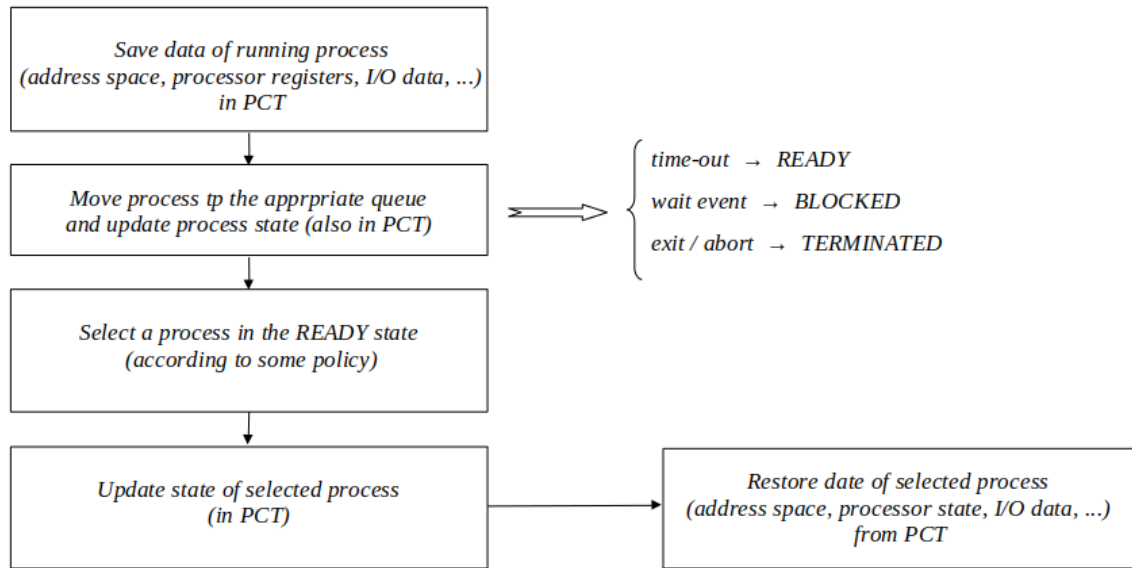
# Process

## Processing a (normal) exception



# Process

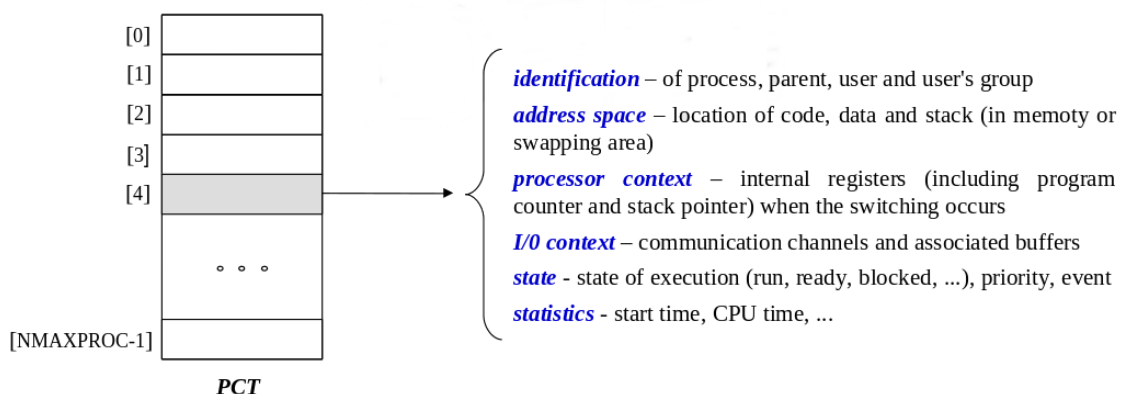
## Processing a process switching



# Processes

## Process control table

- To implement the process model, the operating systems needs a data structure to be used to store the information about each process – **process control block**
- The **process control table (PCT)**, which can be seen as an array of process control blocks, stores information about all processes



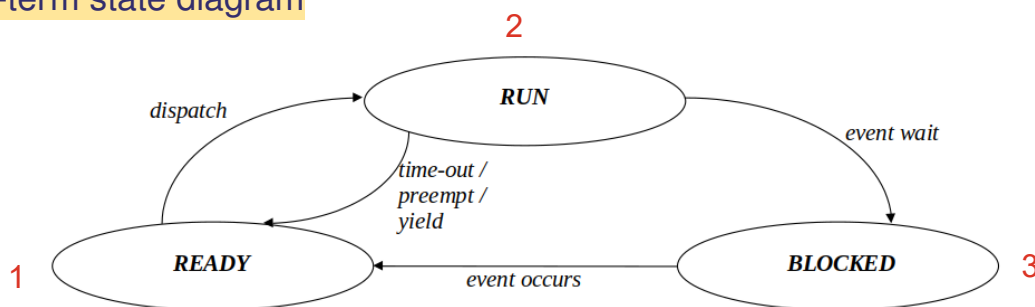
# Processes

## (Short-term) Process states

- A process can be **not running** for different reasons
  - so, one should identify the **possible process states**
- The most important are:
  - **RUN** – the process is in possession of a processor, and thus running
  - **BLOCKED** – the process is waiting for the occurrence of an external event (access to a resource, end of an input/output operation, etc.)
  - **READY** – the process is ready to run, but waiting for the availability of a processor to start/resume its execution
- Transitions between states usually result from external intervention, but, in some cases, can be triggered by the process itself
- The part of the operating system that handles these transitions is called the **(processor) scheduler**, and is an **integral part** of its kernel
  - Different policies exist to control the firing of these transitions
  - They will be covered later

# Processes

## Short-term state diagram



- **event wait** – the running process is prevented to proceed, awaiting the occurrence of an external event
- **dispatch** – one of the processes ready to run is selected and is given the processor
- **event occurs** – an external event occurred and the process waiting for it is now ready to be given the processor
- **time-out** – the time quantum assigned to the running process gets to the end, so the process is removed from the processor
- **preempt** – a higher priority process gets ready to run, so the running process is removed from the processor
- **yield** – the running process voluntarily releases control of the processor

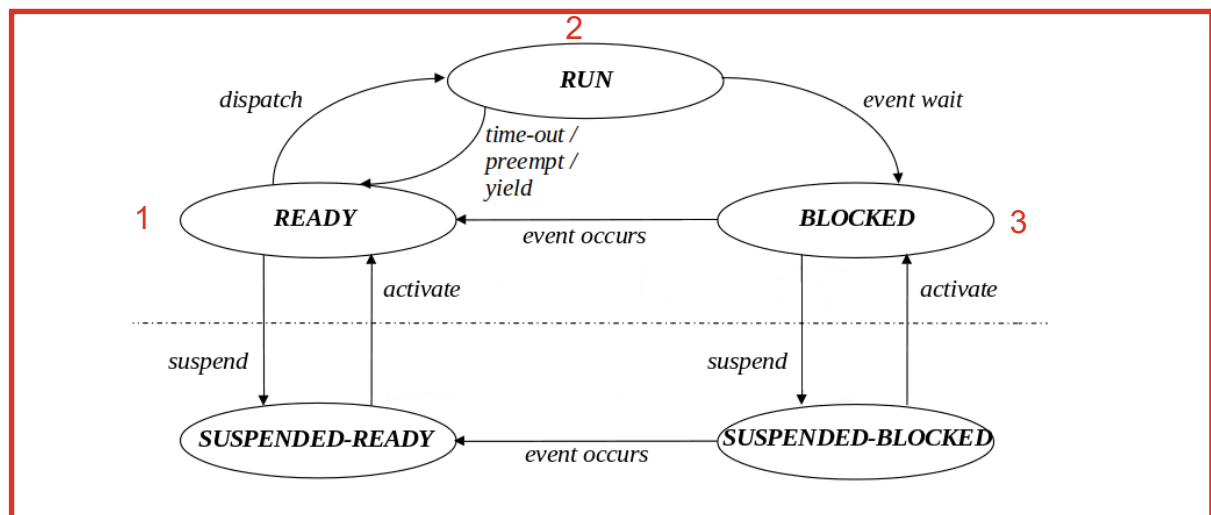
# Processes

## Medium-term states

- The main memory is finite, which limits the number of coexisting processes
- A way to overcome this limitation is to use an area in secondary memory to extend the main memory
  - This is called **swap area** (can be a disk partition or a file)
  - A non running process, or part of it, can be **swapped out**, in order to free main memory for other processes
  - That process will be later on **swapped in**, after main memory becomes available
- Two new states should be added to the process state diagram to incorporate these situations:
  - **suspended-ready** – the process is ready but swapped out
  - **suspended-blocked** – the process is blocked and swapped out

# Processes

## State diagram, including short- and medium-term states



- Two new type of transitions appear:
  - **suspend** – the process is *swapped out*
  - **activate** – the process is *swapped in*

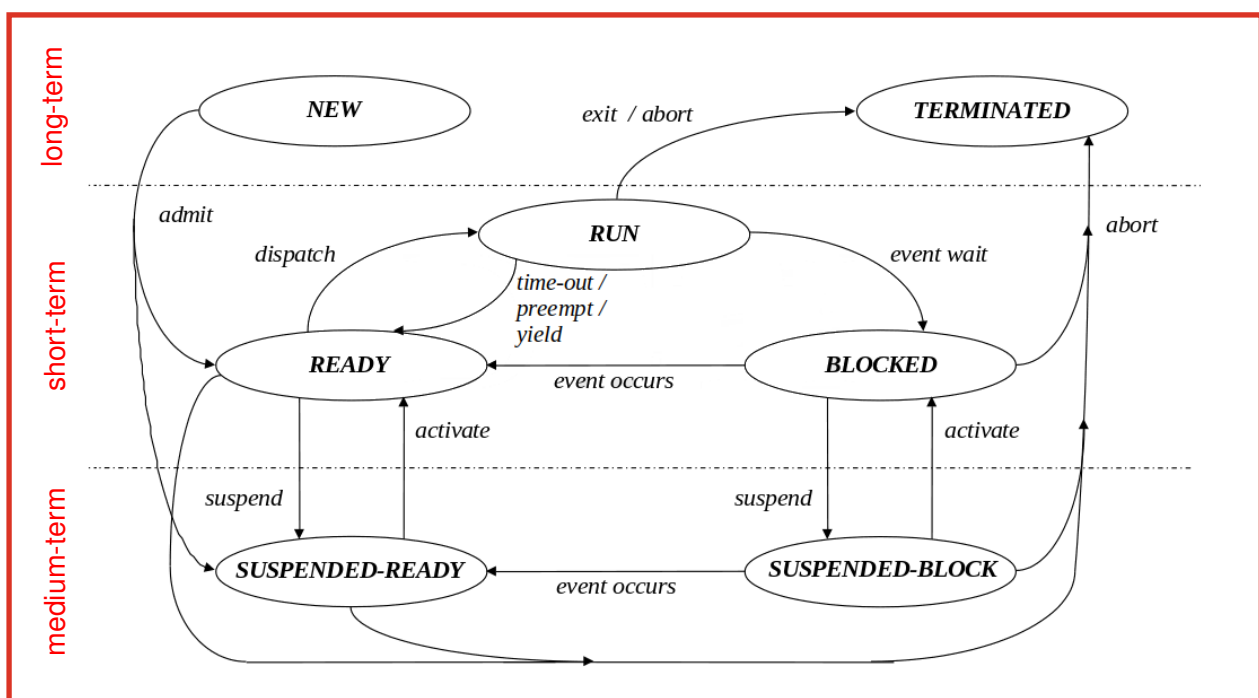
# Processes

## Long-term states and transitions

- The previous state diagram assumes processes are timeless
  - Apart from some system processes this is not true
  - Processes are created, exist for some time, and eventually terminate
- Two new states are required to represent creation and termination
  - **new** – the process has been created but not yet admitted to the pool of executable processes (the process data structure is been initialized)
  - **terminated** – the process has been released from the pool of executable processes, but some actions are still required before the process is discarded
- three new transitions exist
  - **admit** – the process is admitted (by the OS) to the pool of executable processes
  - **exit** – the running process indicates the OS it has completed
  - **abort** – the process is forced to terminate (because of a fatal error or because an authorized process aborts its execution)

# Processes

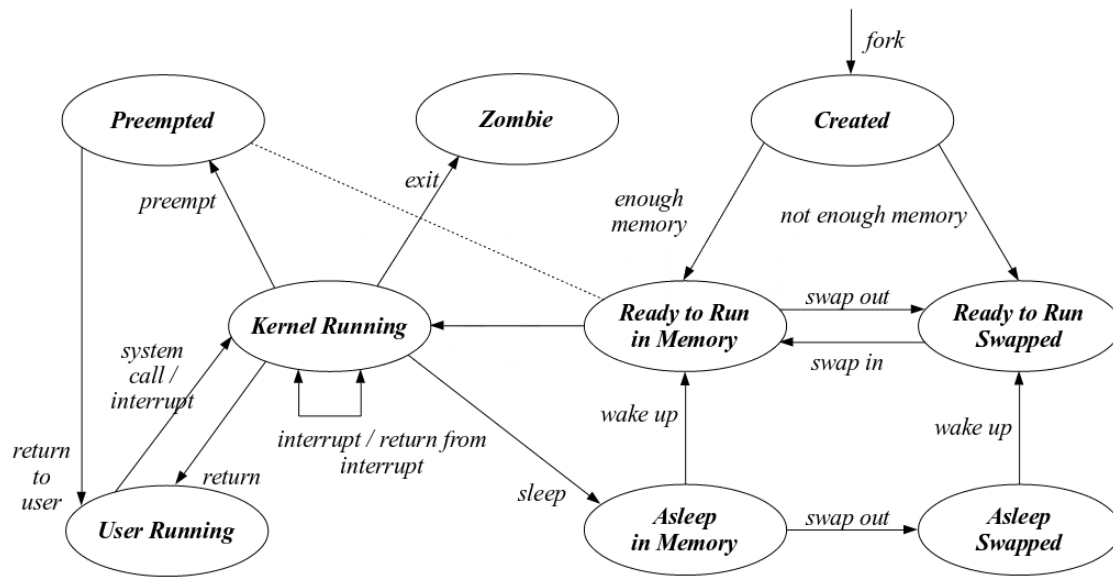
## Global state diagram





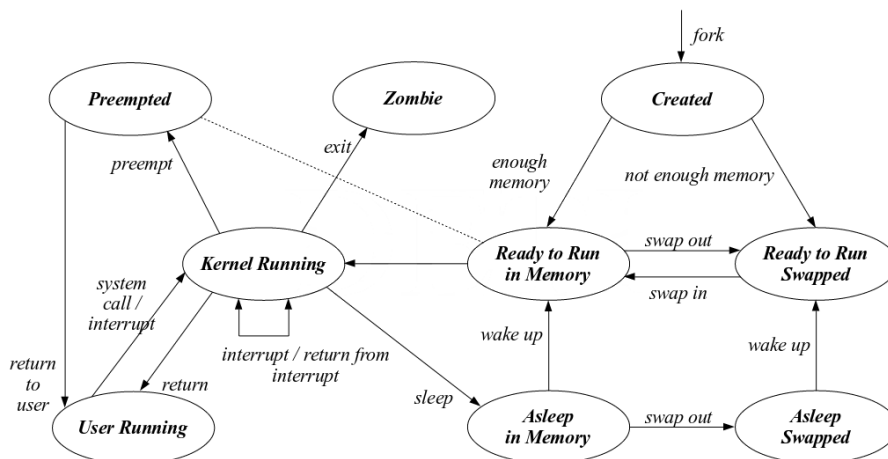
# Processes

## Typical Unix state diagram



# Processes

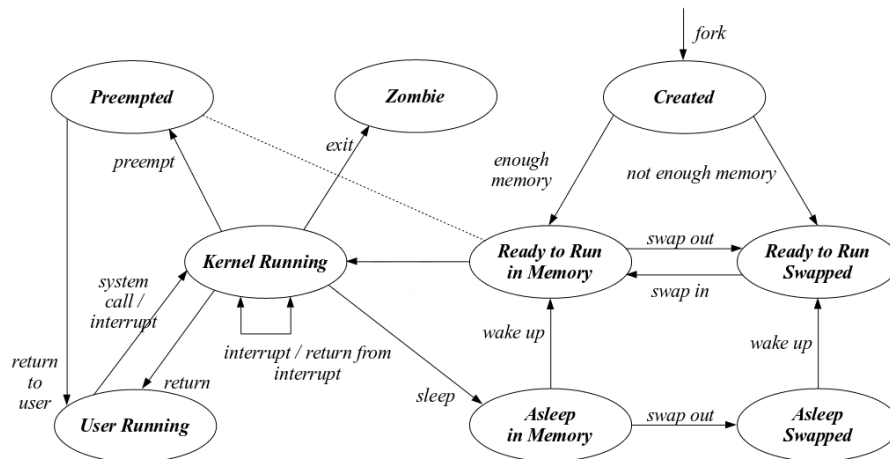
## Typical Unix state diagram (2)



- There are two run states, **kernel running** and **user running**, associated to the processor running mode, **supervisor** and **user**, respectively
- The ready state is also splitted in two states, **ready to run in memory** and **preempted**, but they are equivalent, represented by the dashed line

# Processes

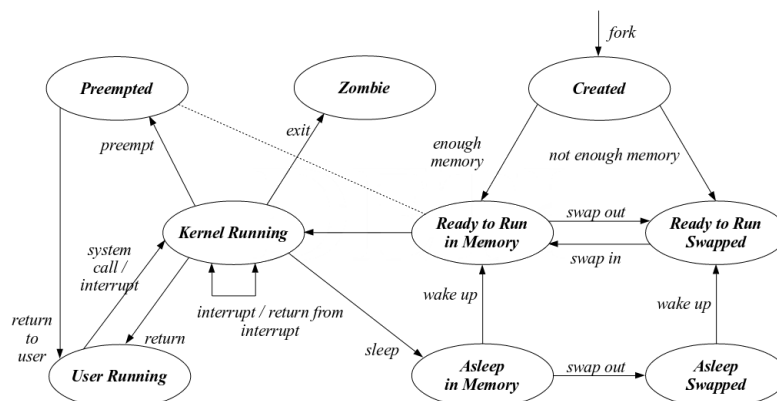
## Typical Unix state diagram (3)



- Only when a user process leaves supervisor mode, it can be preempted (because a higher priority process is ready to run)
- In practice, processes in **ready to run in memory** and **preempted** shared the same queue, thus they are treated as equal
- The **time-out** transition is covered by the preempt one

# Processes

## Typical Unix state diagram (4)

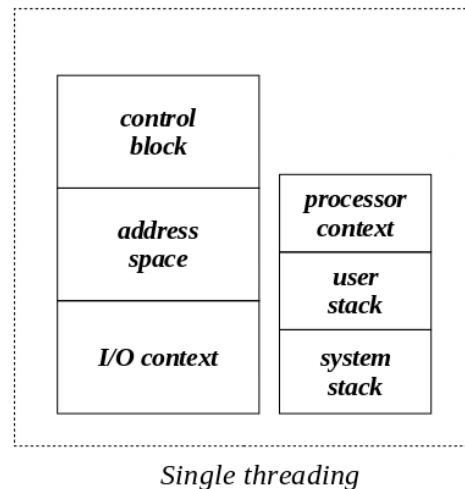


- Traditionally, execution in supervisor mode could not be interrupted (thus UNIX does not allow real time processing)
- In current versions, namely from SVR4, the problem was solved by dividing the code into a succession of atomic regions between which the internal data structures are in a safe state and therefore allowing execution to be interrupted
- This corresponds to a transition between the **preempted** and **kernel running** states, that could be called **return to kernel**

# Threads

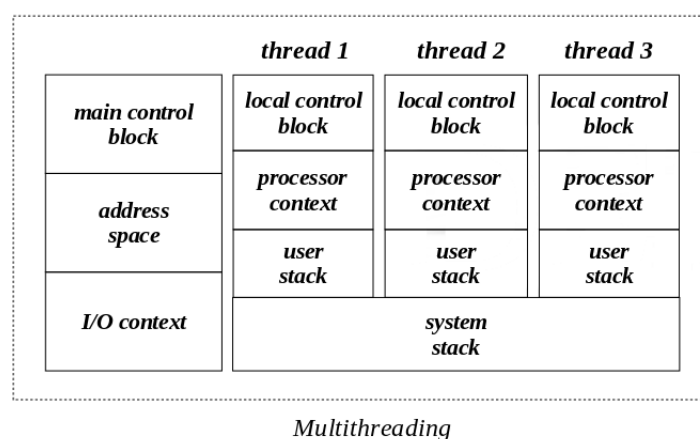
## Single threading

- In traditional operating system, a process includes:
  - an address space (code and data of the associated program)
  - a set of communication channels with I/O devices
  - a single thread of control, which incorporates the processor registers (including the program counter) and a stack
- However, these components can be managed separately
- In this model, **thread** appears as an execution component within a process



# Threads

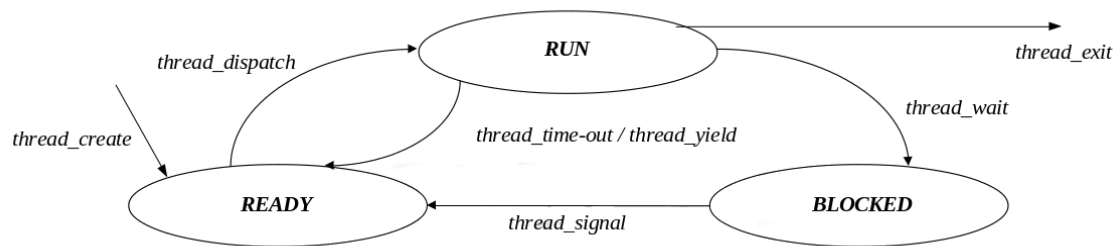
## Multithreading



- Several independent threads can coexist in the same process, thus sharing the same address space and the same I/O context
  - This is referred to as **multithreading**
- Threads can be seen as **light weight processes**

# Threads

## State diagram of a thread



- Only states concerning the management of the processor are considered (short-term states)
- states **suspended-ready** and **suspended-blocked** are not present:
  - they are related to the process, not to the threads
- states **new** and **terminated** are not present:
  - the management of the multiprogramming environment is basically related to restrict the number of threads that can exist within a process

## Bibliography

- Operating Systems: Internals and Design Principles, W. Stallings, Prentice-Hall International Editions, 7th Ed, 2012
  - Chapter 3: Process Description and Control (sections 3.1 to 3.5 and 3.7)
  - Chapter 4: Threads (sections 4.1, 4.2 and 4.6)
- Operating Systems Concepts, A. Silberschatz, P. Galvin and G. Gagne, John Wiley & Sons, 9th Ed, 2013
  - Chapter 3: Processes (sections 3.1 to 3.3)
  - Chapter 4: Threads (sections 4.1 and 4.4.1)
- Modern Operating Systems, A. Tanenbaum and H. Bos, Pearson Education Limited, 4th Ed, 2015
  - Chapter 2: Processes and Threads (sections 2.1 and 2.2)