Process Management

General Presentation and Linux System Calls

Adrian Coleșa

Technical University of Cluj-Napoca (UTCN)
Computer Science Department

April 3rd, 2019

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

The purpose of today's lecture

- Presents general aspects related to process management
- Give examples and details about Linux system calls for processes

Bibliography

- A. Tanenbaum, *Modern Operating Systems*, 2nd Edition, 2001, Chapter 2, Processes, pg. 71 – 100, pg. 132 – 151

Outline

Outline

Process Definition

- Longman dictionary's definition of **process**
  - *a series of actions that are done in order to achieve a particular result*
- **a program in execution** $\Leftrightarrow$ an user application
  - a **sequential** stream of **execution** in its own **memory address space**
  - including the current **values of CPU's registers** (e.g. IP)
- **OS abstraction for using the computer**
  - composed by all that is needed to run a program: CPU, memory, I/O devices etc.
  - it is a **virtualization concept** $\rightarrow$ **virtualizes an entire system** (computer)
  - $\Rightarrow$ **isolation mechanism**, i.e. isolates one execution (process) by another

Process Definition

- Longman dictionary's definition of **process**
  - *a series of actions that are done in order to achieve a particular result*
- **a program in execution** $\Leftrightarrow$ an user application
  - a **sequential** stream of **execution** in its own **memory address space**
  - including the current **values of CPU's registers** (e.g. IP)
- **OS abstraction for using the computer**
  - composed by all that is needed to run a program: CPU, memory, I/O devices etc.
  - it is a **virtualization concept** $\rightarrow$ **virtualizes an entire system** (computer)
  - $\Rightarrow$ **isolation mechanism**, i.e. isolates one execution (process) by another

Process Definition

- Longman dictionary's definition of **process**
  - *a series of actions that are done in order to achieve a particular result*
- **a program in execution** $\Leftrightarrow$ an user application
  - a **sequential** stream of **execution** in its own **memory address space**
  - including the current **values of CPU's registers** (e.g. IP)
- **OS abstraction for using the computer**
  - composed by all that is needed to run a program: CPU, memory, I/O devices etc.
  - it is a **virtualization concept** $\rightarrow$ **virtualizes an entire system** (computer)
  - $\Rightarrow$ **isolation mechanism**, i.e. isolates one execution (process) by another

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Process Definition

- Longman dictionary's definition of **process**
  - *a series of actions that are done in order to achieve a particular result*
- **a program in execution** ⇔ an user application
  - a **sequential** stream of **execution** in its own **memory address space**
  - including the current **values of CPU's registers** (e.g. IP)
- **OS abstraction for using the computer**
  - composed by all that is needed to run a program: CPU, memory, I/O devices etc.
  - it is a **virtualization concept** → **virtualizes an entire system** (computer)
  - ⇒ **isolation mechanism**, i.e. isolates one execution (process) by another

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Process Definition

- Longman dictionary's definition of **process**
  - *a series of actions that are done in order to achieve a particular result*
- **a program in execution** $\Leftrightarrow$ an user application
  - a **sequential** stream of **execution** in its own **memory address space**
  - including the current **values of CPU's registers** (e.g. IP)
- **OS abstraction for using the computer**
  - composed by all that is needed to run a program: CPU, memory, I/O devices etc.
  - it is a **virtualization concept** $\rightarrow$ **virtualizes an entire system** (computer)
  - $\Rightarrow$ **isolation mechanism**, i.e. isolates one execution (process) by another

Process Definition

- Longman dictionary's definition of **process**
  - *a series of actions that are done in order to achieve a particular result*
- **a program in execution** $\Leftrightarrow$ an user application
  - a **sequential** stream of **execution** in its own **memory address space**
  - including the current **values of CPU's registers** (e.g. IP)
- **OS abstraction for using the computer**
  - composed by all that is needed to run a program: CPU, memory, I/O devices etc.
  - it is a **virtualization concept** $\rightarrow$ **virtualizes an entire system** (computer)
  - $\Rightarrow$ **isolation mechanism**, i.e. isolates one execution (process) by another

Process Definition

- Longman dictionary's definition of **process**
  - *a series of actions that are done in order to achieve a particular result*
- **a program in execution** $\Leftrightarrow$ an user application
  - a **sequential** stream of **execution** in its own **memory address space**
  - including the current **values of CPU's registers** (e.g. IP)
- **OS abstraction for using the computer**
  - composed by all that is needed to run a program: CPU, memory, I/O devices etc.
  - it is a **virtualization concept** $\rightarrow$ **virtualizes an entire system** (computer)
  - $\Rightarrow$ **isolation mechanism**, i.e. isolates one execution (process) by another

Process Definition

- Longman dictionary's definition of **process**
  - *a series of actions that are done in order to achieve a particular result*
- **a program in execution** $\Leftrightarrow$ an user application
  - a **sequential** stream of **execution** in its own **memory address space**
  - including the current **values of CPU's registers** (e.g. IP)
- **OS abstraction for using the computer**
  - composed by all that is needed to run a program: CPU, memory, I/O devices etc.
  - it is a **virtualization concept** $\rightarrow$ **virtualizes an entire system** (computer)
  - $\Rightarrow$ **isolation mechanism**, i.e. isolates one execution (process) by another

Process Definition

- Longman dictionary's definition of **process**
    - *a series of actions that are done in order to achieve a particular result*
- **a program in execution** $\Leftrightarrow$ an user application
    - a **sequential** stream of **execution** in its own **memory address space**
    - including the current **values of CPU's registers** (e.g. IP)
- **OS abstraction for using the computer**
    - composed by all that is needed to run a program: CPU, memory, I/O devices etc.
    - it is a **virtualization concept** $\rightarrow$ **virtualizes an entire system** (computer)
    - $\Rightarrow$ **isolation mechanism**, i.e. isolates one execution (process) by another

Process vs Program

- program's source code $\xrightarrow{\textit{compilation}}$ program's executable $\xrightarrow{\textit{launching}}$ process
- **program = static (inactive) entity**
- **process = active entity**
- a process
  - is an activity of some kind
  - is created from a program loaded in memory
  - is allocated system resources (memory, file descriptors, CPU etc.)
  - has input, output, and a state
- the two **parts of a process**
  - **sequential execution**: no concurrency inside a process; everything happens sequentially
  - **process state**: everything that process interacts with (registers, memory, files etc)

Process vs Program

- program's source code $\xrightarrow{compilation}$ program's executable $\xrightarrow{launching}$ process
- **program = static (inactive) entity**
- **process = active entity**
- a process
  - is an activity of some kind
  - is created from a program loaded in memory
  - is allocated system resources (memory, file descriptors, CPU etc.)
  - has input, output, and a state
- the two **parts of a process**
  - **sequential execution**: no concurrency inside a process; everything happens sequentially
  - **process state**: everything that process interacts with (registers, memory, files etc)

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Process vs Program

- program's source code $\xrightarrow{\textit{compilation}}$ program's executable $\xrightarrow{\textit{launching}}$ process
- **program = static (inactive) entity**
- **process = active entity**
- a process
  - is an activity of some kind
  - is created from a program loaded in memory
  - is allocated system resources (memory, file descriptors, CPU etc.)
  - has input, output, and a state
- the two **parts of a process**
  - **sequential execution**: no concurrency inside a process; everything happens sequentially
  - **process state**: everything that process interacts with (registers, memory, files etc)

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Process vs Program

- program's source code $\xrightarrow{\textit{compilation}}$ program's executable $\xrightarrow{\textit{launching}}$ process
- **program = static (inactive) entity**
- **process = active entity**
- a process
    - is an activity of some kind
    - is created from a program loaded in memory
    - is allocated system resources (memory, file descriptors, CPU etc.)
    - has input, output, and a state
- the two **parts of a process**
    - **sequential execution**: no concurrency inside a process; everything happens sequentially
    - **process state**: everything that process interacts with (registers, memory, files etc)

Process vs Program

- program's source code $\xrightarrow{compilation}$ program's executable $\xrightarrow{launching}$ process
- **program = static (inactive) entity**
- **process = active entity**
- a process
  - is an activity of some kind
  - is created from a program loaded in memory
  - is allocated system resources (memory, file descriptors, CPU etc.)
  - has input, output, and a state
- the two **parts of a process**
  - **sequential execution**: no concurrency inside a process; everything happens sequentially
  - **process state**: everything that process interacts with (registers, memory, files etc)

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Process vs Program

- program's source code $\xrightarrow{\textit{compilation}}$ program's executable $\xrightarrow{\textit{launching}}$ process
- **program = static (inactive) entity**
- **process = active entity**
- a process
  - is an activity of some kind
  - is created from a program loaded in memory
  - is allocated system resources (memory, file descriptors, CPU etc.)
  - has input, output, and a state
- the two **parts of a process**
  - **sequential execution**: no concurrency inside a process; everything happens sequentially
  - **process state**: everything that process interacts with (registers, memory, files etc)

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Process vs Program

- program's source code $\xrightarrow{\textit{compilation}}$ program's executable $\xrightarrow{\textit{launching}}$ process
- **program = static (inactive) entity**
- **process = active entity**
- a process
  - is an activity of some kind
  - is created from a program loaded in memory
  - is allocated system resources (memory, file descriptors, CPU etc.)
  - has input, output, and a state
- the two **parts of a process**
  - **sequential execution**: no concurrency inside a process; everything happens sequentially
  - **process state**: everything that process interacts with (registers, memory, files etc)

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Process vs Program

- program's source code $\xrightarrow{\textit{compilation}}$ program's executable $\xrightarrow{\textit{launching}}$ process
- **program = static (inactive) entity**
- **process = active entity**
- a process
  - is an activity of some kind
  - is created from a program loaded in memory
  - is allocated system resources (memory, file descriptors, CPU etc.)
  - has input, output, and a state
- the two **parts of a process**
  - **sequential execution**: no concurrency inside a process; everything happens sequentially
  - **process state**: everything that process interacts with (registers, memory, files etc)

Process vs Program

- program's source code $\xrightarrow{compilation}$ program's executable $\xrightarrow{launching}$ process
- **program = static (inactive) entity**
- **process = active entity**
- a process
  - is an activity of some kind
  - is created from a program loaded in memory
  - is allocated system resources (memory, file descriptors, CPU etc.)
  - has input, output, and a state
- the two **parts of a process**
  - **sequential execution**: no concurrency inside a process; everything happens sequentially
  - **process state**: everything that process interacts with (registers, memory, files etc)

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Process vs Program

- program's source code $\xrightarrow{\textit{compilation}}$ program's executable $\xrightarrow{\textit{launching}}$ process
- **program = static (inactive) entity**
- **process = active entity**
- a process
  - is an activity of some kind
  - is created from a program loaded in memory
  - is allocated system resources (memory, file descriptors, CPU etc.)
  - has input, output, and a state
- the two **parts of a process**
  - **sequential execution**: no concurrency inside a process; everything happens sequentially
  - **process state**: everything that process interacts with (registers, memory, files etc)

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Process vs Program

- program's source code $\xrightarrow{\textit{compilation}}$ program's executable $\xrightarrow{\textit{launching}}$ process
- **program = static (inactive) entity**
- **process = active entity**
- a process
  - is an activity of some kind
  - is created from a program loaded in memory
  - is allocated system resources (memory, file descriptors, CPU etc.)
  - has input, output, and a state
- the two **parts of a process**
  - **sequential execution**: no concurrency inside a process; everything happens sequentially
  - **process state**: everything that process interacts with (registers, memory, files etc)

The Considered Context

- single-processor systems
- multiprogramming and time (processor) sharing
  - pseudo-parallelism
  - switching among processes
  - scheduling algorithm

The Considered Context

- single-processor systems
- multiprogramming and time (processor) sharing
  - pseudo-parallelism
  - switching among processes
  - scheduling algorithm

The Considered Context

- single-processor systems
- multiprogramming and time (processor) sharing
  - pseudo-parallelism
  - switching among processes
  - scheduling algorithm

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

The Considered Context

- single-processor systems
- multiprogramming and time (processor) sharing
  - pseudo-parallelism
  - switching among processes
  - scheduling algorithm

The Considered Context

- single-processor systems
- multiprogramming and time (processor) sharing
  - pseudo-parallelism
  - switching among processes
  - scheduling algorithm

Process Creation

- automatically by the OS (uncommon case)
  - at system initialization
  - reacting to different events
  - usually run as background processes (vs. foreground processes)
- **by another process** (common case)
  - there is a **system call** provided for process creation
  - situations
  - leads to a **process hierarchy** based on the **parent-child** relationship

Process Creation

- automatically by the OS (uncommon case)
  - at system initialization
  - reacting to different events
  - usually run as background processes (vs. foreground processes)
- **by another process** (common case)
  - there is a **system call** provided for process creation
  - situations
    - when a process needs help doing some computation
    - when a user action occurs: e.g. interaction with the shell
  - leads to a **process hierarchy** based on the **parent-child** relationship

Process Creation

- automatically by the OS (uncommon case)
  - at system initialization
  - reacting to different events
  - usually run as background processes (vs. foreground processes)
- **by another process** (common case)
  - there is a **system call** provided for process creation
  - situations
  - leads to a **process hierarchy** based on the **parent-child** relationship

Process Creation

- automatically by the OS (uncommon case)
  - at system initialization
  - reacting to different events
  - usually run as background processes (vs. foreground processes)
- **by another process** (common case)
  - there is a **system call** provided for process creation
  - situations
    - when a process needs help doing some computation
    - when a user action occurs, e.g. interaction with the shell
  - leads to a **process hierarchy** based on the **parent-child** relationship

Process Creation

- automatically by the OS (uncommon case)
  - at system initialization
  - reacting to different events
  - usually run as background processes (vs. foreground processes)
- **by another process** (common case)
  - there is a **system call** provided for process creation
  - situations
    - when a process needs help doing some computation
    - when a user action occurs, e.g. interaction with the shell
  - leads to a **process hierarchy** based on the **parent-child** relationship

Process Creation

- automatically by the OS (uncommon case)
  - at system initialization
  - reacting to different events
  - usually run as background processes (vs. foreground processes)
- **by another process** (common case)
  - there is a **system call** provided for process creation
  - situations
    - when a process needs help doing some computation
    - when a user action occurs, e.g. interaction with the shell
  - leads to a **process hierarchy** based on the **parent-child** relationship

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Process Creation

- automatically by the OS (uncommon case)
  - at system initialization
  - reacting to different events
  - usually run as background processes (vs. foreground processes)
- **by another process** (common case)
  - there is a **system call** provided for process creation
  - situations
    - when a process needs help doing some computation
    - when a user action occurs, e.g. interaction with the shell
  - leads to a **process hierarchy** based on the **parent-child** relationship

Process Creation

- automatically by the OS (uncommon case)
  - at system initialization
  - reacting to different events
  - usually run as background processes (vs. foreground processes)
- **by another process** (common case)
  - there is a **system call** provided for process creation
  - situations
    - when a process needs help doing some computation
    - when a user action occurs, e.g. interaction with the shell
  - leads to a **process hierarchy** based on the **parent-child** relationship

Process Creation

- automatically by the OS (uncommon case)
  - at system initialization
  - reacting to different events
  - usually run as background processes (vs. foreground processes)
- **by another process** (common case)
  - there is a **system call** provided for process creation
  - situations
    - when a process needs help doing some computation
    - when a user action occurs, e.g. interaction with the shell
  - leads to a **process hierarchy** based on the **parent-child** relationship

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Process Creation

- automatically by the OS (uncommon case)
  - at system initialization
  - reacting to different events
  - usually run as background processes (vs. foreground processes)
- **by another process** (common case)
  - there is a **system call** provided for process creation
  - situations
    - when a process needs help doing some computation
    - when a user action occurs, e.g. interaction with the shell
  - leads to a **process hierarchy** based on the **parent-child** relationship

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Process termination

- **voluntarily**, using a special **system call**
  - **normal exit**, i.e. end of program's execution
  - **error detection exit**, like: inexistent files, insufficient or incorrect input etc.
- **involuntarily**, being forcefully terminated
  - initiated by the system due to a "fatal error", like: illegal instructions, division by zero, segmentation fault etc.
  - initiated by another process (i.e. "killed")

Process termination

- **voluntarily**, using a special **system call**
  - **normal exit**, i.e. end of program's execution
  - **error detection exit**, like: inexistent files, insufficient or incorrect input etc.
- **involuntarily**, being forcefully terminated
  - initiated by the system due to a "fatal error", like: illegal instructions, division by zero, segmentation fault etc.
  - initiated by another process (i.e. "killed")

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Process termination

- **voluntarily**, using a special **system call**
  - **normal exit**, i.e. end of program's execution
  - **error detection exit**, like: inexistent files, insufficient or incorrect input etc.
- **involuntarily**, being forcefully terminated
  - initiated by the system due to a "fatal error", like: illegal instructions, division by zero, segmentation fault etc.
  - initiated by another process (i.e. "killed")

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Process termination

- **voluntarily**, using a special **system call**
  - **normal exit**, i.e. end of program's execution
  - **error detection exit**, like: inexistent files, insufficient or incorrect input etc.
- **involuntarily**, being forcefully terminated
  - initiated by the system due to a "fatal error", like: illegal instructions, division by zero, segmentation fault etc.
  - initiated by another process (i.e. "killed")

Process termination

- **voluntarily**, using a special **system call**
  - **normal exit**, i.e. end of program's execution
  - **error detection exit**, like: inexistent files, insufficient or incorrect input etc.
- **involuntarily**, being forcefully terminated
  - initiated by the system due to a "fatal error", like: illegal instructions, division by zero, segmentation fault etc.
  - initiated by another process (i.e. "killed")

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Process termination

- **voluntarily**, using a special **system call**
  - **normal exit**, i.e. end of program's execution
  - **error detection exit**, like: inexistent files, insufficient or incorrect input etc.
- **involuntarily**, being forcefully terminated
  - initiated by the system due to a "fatal error", like: illegal instructions, division by zero, segmentation fault etc.
  - initiated by another process (i.e. "killed")

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Process states

- **running**
  - **executed by the CPU**, i.e. using the CPU, at that moment
  - only one process in that state / CPU, actually as many as the number of system's CPUs
- **ready**
  - ready to be executed, but no CPU available
  - so **wait for a CPU to become available**
  - **transparent to the program**
- **blocked**
  - **wait for an event to occur**, a resource to become available
  - **triggered by the application explicitly** through **blocking system calls**
- just created (optional)
  - waiting for some resources to be allocated
- **terminated** (optional)
  - keeping information about the exit state

Process states

- **running**
  - **executed by the CPU**, i.e. using the CPU, at that moment
  - only one process in that state / CPU, actually as many as the number of system's CPUs
- **ready**
  - ready to be executed, but no CPU available
  - so **wait for a CPU to become available**
  - **transparent to the program**
- **blocked**
  - **wait for an event to occur**, a resource to become available
  - **triggered by the application explicitly** through **blocking system calls**
- just created (optional)
  - waiting for some resources to be allocated
- **terminated** (optional)
  - keeping information about the exit state

Process states

- **running**
  - **executed by the CPU**, i.e. using the CPU, at that moment
  - only one process in that state / CPU, actually as many as the number of system's CPUs
- **ready**
  - ready to be executed, but no CPU available
  - so **wait for a CPU to become available**
  - **transparent to the program**
- **blocked**
  - **wait for an event to occur**, a resource to become available
  - **triggered by the application explicitly** through **blocking system calls**
- just created (optional)
  - waiting for some resources to be allocated
- **terminated** (optional)
  - keeping information about the exit state

Process states

- **running**
  - **executed by the CPU**, i.e. using the CPU, at that moment
  - only one process in that state / CPU, actually as many as the number of system's CPUs
- **ready**
  - ready to be executed, but no CPU available
  - so **wait for a CPU to become available**
  - **transparent to the program**
- **blocked**
  - **wait for an event to occur**, a resource to become available
  - **triggered by the application explicitly** through **blocking system calls**
- just created (optional)
  - waiting for some resources to be allocated
- **terminated** (optional)
  - keeping information about the exit state

Process states

- **running**
  - **executed by the CPU**, i.e. using the CPU, at that moment
  - only one process in that state / CPU, actually as many as the number of system's CPUs
- **ready**
  - ready to be executed, but no CPU available
  - so **wait for a CPU to become available**
  - **transparent to the program**
- **blocked**
  - **wait for an event to occur**, a resource to become available
  - **triggered by the application explicitly** through **blocking system calls**
- just created (optional)
  - waiting for some resources to be allocated
- **terminated** (optional)
  - keeping information about the exit state

Process states

- **running**
  - **executed by the CPU**, i.e. using the CPU, at that moment
  - only one process in that state / CPU, actually as many as the number of system's CPUs
- **ready**
  - ready to be executed, but no CPU available
  - so **wait for a CPU to become available**
  - **transparent to the program**
- **blocked**
  - **wait for an event to occur**, a resource to become available
  - **triggered by the application explicitly** through **blocking system calls**
- just created (optional)
  - waiting for some resources to be allocated
- **terminated** (optional)
  - keeping information about the exit state

Process states

- **running**
  - **executed by the CPU**, i.e. using the CPU, at that moment
  - only one process in that state / CPU, actually as many as the number of system's CPUs
- **ready**
  - ready to be executed, but no CPU available
  - so **wait for a CPU to become available**
  - **transparent to the program**
- **blocked**
  - **wait for an event to occur**, a resource to become available
  - **triggered by the application explicitly** through **blocking system calls**
- just created (optional)
  - waiting for some resources to be allocated
- **terminated** (optional)
  - keeping information about the exit state

Process states

- **running**
  - **executed by the CPU**, i.e. using the CPU, at that moment
  - only one process in that state / CPU, actually as many as the number of system's CPUs
- **ready**
  - ready to be executed, but no CPU available
  - so **wait for a CPU to become available**
  - **transparent to the program**
- **blocked**
  - **wait for an event to occur**, a resource to become available
  - **triggered by the application explicitly** through **blocking system calls**
- just created (optional)
  - waiting for some resources to be allocated
- **terminated** (optional)
  - keeping information about the exit state

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Process states

- **running**
  - **executed by the CPU**, i.e. using the CPU, at that moment
  - only one process in that state / CPU, actually as many as the number of system's CPUs
- **ready**
  - ready to be executed, but no CPU available
  - so **wait for a CPU to become available**
  - **transparent to the program**
- **blocked**
  - **wait for an event to occur**, a resource to become available
  - **triggered by the application explicitly** through **blocking system calls**
- just created (optional)
  - waiting for some resources to be allocated
- **terminated** (optional)
  - keeping information about the exit state

Process states

- **running**
  - **executed by the CPU**, i.e. using the CPU, at that moment
  - only one process in that state / CPU, actually as many as the number of system's CPUs
- **ready**
  - ready to be executed, but no CPU available
  - so **wait for a CPU to become available**
  - **transparent to the program**
- **blocked**
  - **wait for an event to occur**, a resource to become available
  - **triggered by the application explicitly** through **blocking system calls**
- just created (optional)
  - waiting for some resources to be allocated
- **terminated** (optional)
  - keeping information about the exit state

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Process states

- **running**
  - **executed by the CPU**, i.e. using the CPU, at that moment
  - only one process in that state / CPU, actually as many as the number of system's CPUs
- **ready**
  - ready to be executed, but no CPU available
  - so **wait for a CPU to become available**
  - **transparent to the program**
- **blocked**
  - **wait for an event to occur**, a resource to become available
  - **triggered by the application explicitly** through **blocking system calls**
- just created (optional)
  - waiting for some resources to be allocated
- **terminated** (optional)
  - keeping information about the exit state

Process states

- **running**
  - **executed by the CPU**, i.e. using the CPU, at that moment
  - only one process in that state / CPU, actually as many as the number of system's CPUs
- **ready**
  - ready to be executed, but no CPU available
  - so **wait for a CPU to become available**
  - **transparent to the program**
- **blocked**
  - **wait for an event to occur**, a resource to become available
  - **triggered by the application explicitly** through **blocking system calls**
- just created (optional)
  - waiting for some resources to be allocated
- **terminated** (optional)
  - keeping information about the exit state

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Process states

- **running**
  - **executed by the CPU**, i.e. using the CPU, at that moment
  - only one process in that state / CPU, actually as many as the number of system's CPUs
- **ready**
  - ready to be executed, but no CPU available
  - so **wait for a CPU to become available**
  - **transparent to the program**
- **blocked**
  - **wait for an event to occur**, a resource to become available
  - **triggered by the application explicitly** through **blocking system calls**
- just created (optional)
  - waiting for some resources to be allocated
- **terminated** (optional)
  - keeping information about the exit state

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Process states

- **running**
  - **executed by the CPU**, i.e. using the CPU, at that moment
  - only one process in that state / CPU, actually as many as the number of system's CPUs
- **ready**
  - ready to be executed, but no CPU available
  - so **wait for a CPU to become available**
  - **transparent to the program**
- **blocked**
  - **wait for an event to occur**, a resource to become available
  - **triggered by the application explicitly** through **blocking system calls**
- just created (optional)
  - waiting for some resources to be allocated
- **terminated** (optional)
  - keeping information about the exit state

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Process States Transitions

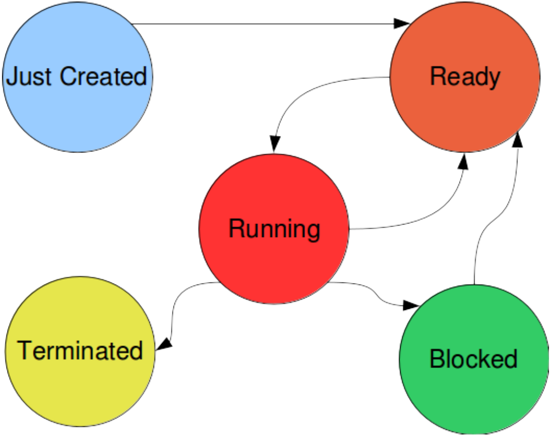


Figure: Process States Transition

Outline

Outline

Process Creation: `fork()`

- system call used to **create a new process**
- child process' contents is identical with that of its parent
- still, two distinct and independent processes
- the two processes are scheduled independently on the CPU
- parent processes continue its execution returning from `fork`
- child starts its execution returning from `fork`
- `fork` returns
  - a positive value (child's PID) in parent
  - zero in child

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Process Creation: `fork()`

- system call used to **create a new process**
- child process' contents is identical with that of its parent
- still, two distinct and independent processes
- the two processes are scheduled independently on the CPU
- parent processes continue its execution returning from `fork`
- child starts its execution returning from `fork`
- `fork` returns
  - a positive value (child's PID) in parent
  - zero in child

Process Creation: `fork()`

- system call used to **create a new process**
- child process' contents is identical with that of its parent
- still, two distinct and independent processes
- the two processes are scheduled independently on the CPU
- parent processes continue its execution returning from `fork`
- child starts its execution returning from `fork`
- `fork` returns
  - a positive value (child's PID) in parent
  - zero in child

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Process Creation: `fork()`

- system call used to **create a new process**
- child process' contents is identical with that of its parent
- still, two distinct and independent processes
- the two processes are scheduled independently on the CPU
- parent processes continue its execution returning from `fork`
- child starts its execution returning from `fork`
- `fork` returns
  - a positive value (child's PID) in parent
  - zero in child

Process Creation: `fork()`

- system call used to **create a new process**
- child process' contents is identical with that of its parent
- still, two distinct and independent processes
- the two processes are scheduled independently on the CPU
- parent processes continue its execution returning from `fork`
- child starts its execution returning from `fork`
- `fork` returns
  - a positive value (child's PID) in parent
  - zero in child

Process Creation: `fork()`

- system call used to **create a new process**
- child process' contents is identical with that of its parent
- still, two distinct and independent processes
- the two processes are scheduled independently on the CPU
- parent processes continue its execution returning from `fork`
- child starts its execution returning from `fork`
- `fork` returns
  - a positive value (child's PID) in parent
  - zero in child

Process Creation: `fork()`

- system call used to **create a new process**
- child process' contents is identical with that of its parent
- still, two distinct and independent processes
- the two processes are scheduled independently on the CPU
- parent processes continue its execution returning from `fork`
- child starts its execution returning from `fork`
- `fork` returns
  - a positive value (child's PID) in parent
  - zero in child

Process Creation: `fork()`

- system call used to **create a new process**
- child process' contents is identical with that of its parent
- still, two distinct and independent processes
- the two processes are scheduled independently on the CPU
- parent processes continue its execution returning from `fork`
- child starts its execution returning from `fork`
- `fork` returns
  - a positive value (child's PID) in parent
  - zero in child

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Process Creation: `fork()`

- system call used to **create a new process**
- child process' contents is identical with that of its parent
- still, two distinct and independent processes
- the two processes are scheduled independently on the CPU
- parent processes continue its execution returning from `fork`
- child starts its execution returning from `fork`
- `fork` returns
  - a positive value (child's PID) in parent
  - zero in child

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

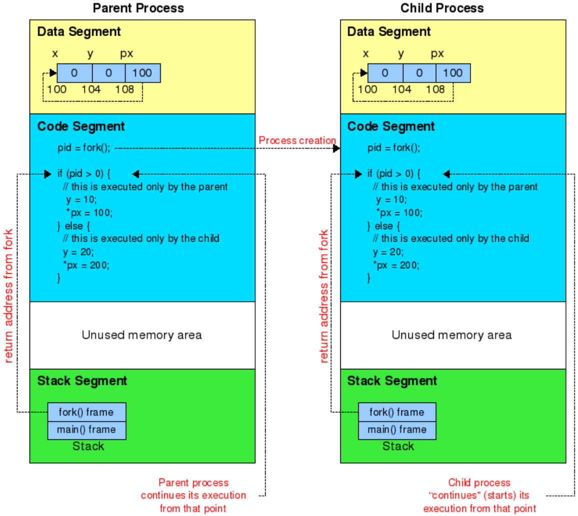
fork Usage Example

```
int x;
static int y;
int *px;

int main(int argc, char **argv)
{
    int pid;

    x = 0;
    px = &x;
    y = 0;

    // up to this point only the parent exists
    // now parent callds fork() to create a new process
    pid = fork();
    if (pid < 0) {
        // error case: no child process created
        perror("Cannot create a new process");
        exit(1);
    }
    // from now on there are two processes: parent and child
```

fork Usage Example (cont.)

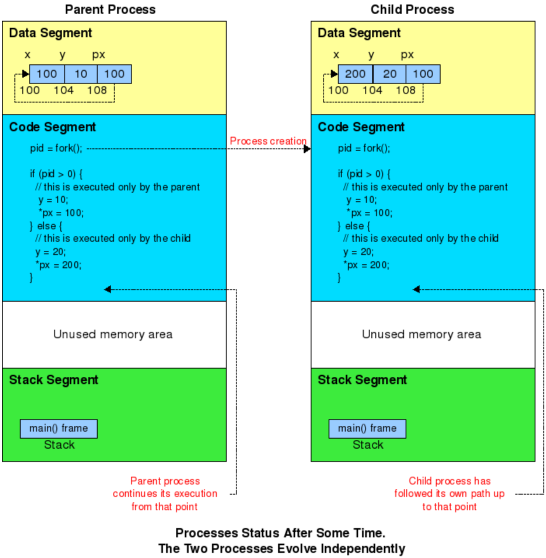
```
    // executed by both processes
    printf("x=%d, px =%p, *px=%d, y=%d\n", x, px, *px, y);
       // parent: x=0, px =0x601050, *px=0, y=0
       // child:  x=0, px =0x601050, *px=0, y=0

    if (pid == 0) { // executed only by the child
        y = 20;
        *px = 200;
    } else {        // executed only by the parent
        y = 10;
        *px = 100;
    }

    // executed by both processes
    printf("x=%d, px=%p, *px=%d, y=%d\n", x, px, *px, y);
       // parent: x=100, px=0x601050, *px=100, y=10
       // child:  x=200, px=0x601050, *px=200, y=20
}
```

fork's Effect Illustration



Processes Status Immediately After Fork.
The Child Contents Is Identical With That of Its Parent

fork's Effect Illustration



fork() syscall creates an *independent* child process, which starts as a *copy of its parent!*

Let's practice!

Have you really understood how fork() works?

If you have, try solving the following problems:

1. You are given the following code:

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

   - How many processes does the following code creates?
   - Draw the resulted process hierarchy.

2. You are given the following code:

```
for(i=1; i<=100; i++)
    fork();
```

   - How many processes does the following code creates?
   - Draw the resulted process hierarchy.

Code Execution: `exec` Family

- system call used to **load a new code into the calling process**
  - replace the calling process' contents, but not its identity
- there are more exec system calls
  - `execl`, `execlp`: with variable number of arguments
  - `execv`, `execvp`: with a fixed number of arguments
- the `exec`'s parameters **similar to a command line**
  - the first argument is **always the path** to the executable file
  - the next argument(s) describe the command line, starting with command name

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Code Execution: exec Family

- system call used to **load a new code into the calling process**
  - replace the calling process' contents, but not its identity
- there are more exec system calls
  - `execl`, `execlp`: with variable number of arguments
  - `execv`, `execvp`: with a fixed number of arguments
- the `exec`'s parameters **similar to a command line**
  - the first argument is **always the path** to the executable file
  - the next argument(s) describe the command line, starting with command name

Code Execution: exec Family

- system call used to **load a new code into the calling process**
  - replace the calling process' contents, but not its identity
- there are more exec system calls
  - `execl`, `execlp`: with variable number of arguments
  - `execv`, `execvp`: with a fixed number of arguments
- the exec's parameters **similar to a command line**
  - the first argument is **always the path** to the executable file
  - the next argument(s) describe the command line, starting with command name

Code Execution: exec Family

- system call used to **load a new code into the calling process**
  - replace the calling process' contents, but not its identity
- there are more exec system calls
  - `execl`, `execlp`: with variable number of arguments
  - `execv`, `execvp`: with a fixed number of arguments
- the exec's parameters **similar to a command line**
  - the first argument is **always the path** to the executable file
  - the next argument(s) describe the command line, starting with command name

Code Execution: `exec` Family

- system call used to **load a new code into the calling process**
  - replace the calling process' contents, but not its identity
- there are more exec system calls
  - `execl`, `execlp`: with variable number of arguments
  - `execv`, `execvp`: with a fixed number of arguments
- the exec's parameters **similar to a command line**
  - the first argument is **always the path** to the executable file
  - the next argument(s) describe the command line, starting with command name

Code Execution: exec Family

- system call used to **load a new code into the calling process**
  - replace the calling process' contents, but not its identity
- there are more exec system calls
  - `execl`, `execlp`: with variable number of arguments
  - `execv`, `execvp`: with a fixed number of arguments
- the `exec`'s parameters **similar to a command line**
  - the first argument is **always the path** to the executable file
  - the next argument(s) describe the command line, starting with command name

Code Execution: exec Family

- system call used to **load a new code into the calling process**
  - replace the calling process' contents, but not its identity
- there are more exec system calls
  - `execl`, `execlp`: with variable number of arguments
  - `execv`, `execvp`: with a fixed number of arguments
- the `exec`'s parameters **similar to a command line**
  - the first argument is **always the path** to the executable file
  - the next argument(s) describe the command line, starting with command name

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Code Execution: `exec` Family

- system call used to **load a new code into the calling process**
  - replace the calling process' contents, but not its identity
- there are more exec system calls
  - `execl`, `execlp`: with variable number of arguments
  - `execv`, `execvp`: with a fixed number of arguments
- the `exec`'s parameters **similar to a command line**
  - the first argument is **always the path** to the executable file
  - the next argument(s) describe the command line, starting with command name

execl and execlp Usage Example

```
// first parameter is the EXPLICIT path to the executable file
execl("/bin/ls", "ls", "-l", NULL);
execl("./myprg.exe", "myprg.exe", "param1", "param2", NULL);

// first parameter is the IMPLICIT path to the executable file
// the path is searched in the directories stored
// in the PATH environment variable
execlp("ls", "ls", "-l", 0);
```

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

execv and execvp Usage Example

```
char cmdline[10][100]; // equiv. to char *cmdline[];
                        // equiv. to char **cmdline;

// build the command line
strncpy(cmdline[0], "ls", 99);
strncpy(cmdline[1], "-l", 99);
cmdline[2] = NULL;

// call the exec

// first parameter is the EXPLICIT path to the executable file
execv("/bin/ls", cmdline);

// first parameter is the IMPLICIT path to the executable file
// the path is searched in the directories stored
// in the PATH environment variable
execvp("ls", cmdline);
```

exec() syscalls loads a new code in the calling process!
There is no return from exec() if successfully executed!

Relationship Between `fork` and `exec`

- used to
  - create a child process
  - executing something else than its parent
- **Why there are two separated steps instead of just one?**
  - between them the parent "has control" over its child (see standard input and output redirection below)
  - the **parent is released** by the burden of (i.e. time spent) **loading a new code in child**
  - ⇒ **better performance** for the parent

Relationship Between `fork` and `exec`

- used to
  - create a child process
  - executing something else than its parent
- **Why there are two separated steps instead of just one?**
  - between them the parent "has control" over its child (see standard input and output redirection below)
  - the **parent is released** by the burden of (i.e. time spent) **loading a new code in child**
  - ⇒ **better performance** for the parent

Relationship Between `fork` and `exec`

- used to
  - create a child process
  - executing something else than its parent
- **Why there are two separated steps instead of just one?**
  - between them the parent "has control" over its child (see standard input and output redirection below)
  - the **parent is released** by the burden of (i.e. time spent) **loading a new code in child**
  - $\Rightarrow$ **better performance** for the parent

Relationship Between `fork` and `exec`

- used to
  - create a child process
  - executing something else than its parent
- **Why there are two separated steps instead of just one?**
  - between them the parent "has control" over its child (see standard input and output redirection below)
  - the **parent is released** by the burden of (i.e. time spent) **loading a new code in child**
  - $\Rightarrow$ **better performance** for the parent

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Relationship Between `fork` and `exec`

- used to
  - create a child process
  - executing something else than its parent
- **Why there are two separated steps instead of just one?**
  - between them the parent "has control" over its child (see standard input and output redirection below)
  - the **parent is released** by the burden of (i.e. time spent) **loading a new code in child**
  - $\Rightarrow$ **better performance** for the parent

Relationship Between `fork` and `exec`

- used to
  - create a child process
  - executing something else than its parent
- **Why there are two separated steps instead of just one?**
  - between them the parent "has control" over its child (see standard input and output redirection below)
  - the **parent is released** by the burden of (i.e. time spent) **loading a new code in child**
  - $\Rightarrow$ **better performance** for the parent

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Relationship Between `fork` and `exec`

- used to
  - create a child process
  - executing something else than its parent
- **Why there are two separated steps instead of just one?**
  - between them the parent "has control" over its child (see standard input and output redirection below)
  - the **parent is released** by the burden of (i.e. time spent) **loading a new code in child**
  - $\Rightarrow$ **better performance** for the parent

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

fork and exec Usage Example

- parent code

```
int main()
{
    pid = fork();

    if (pid > 0) {
        // parent doing something
    } else {
        // child loading and executing a new code
        execl("./child.exe", "child.exe", "p1", "10", 0);
        perror("execl has not succeded");
    }
}
```

- child code

```
int main(int argc, char **argv)
{
    int p;
    for (p=0; p<argc; p++)
        printf("argv[%d]=%s\n", argv[p]);
}
```

Process Termination: exit

- system call used to **terminate voluntarily a process**
- terminate the calling process
- specify an exit code
  - 0 (zero) exit code considered successfully termination
  - anything else considered erroneously termination
- exit code is kept until the parent process asks for it
- example
  - `exit(0);`
  - `exit(1);`

Process Termination: exit

- system call used to **terminate voluntarily a process**
- terminate the calling process
- specify an exit code
  - 0 (zero) exit code considered successfully termination
  - anything else considered erroneously termination
- exit code is kept until the parent process asks for it
- example
  - `exit(0);`
  - `exit(1);`

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Process Termination: `exit`

- system call used to **terminate voluntarily a process**
- terminate the calling process
- specify an exit code
  - 0 (zero) exit code considered successfully termination
  - anything else considered erroneously termination
- exit code is kept until the parent process asks for it
- example
  - `exit(0);`
  - `exit(1);`

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Process Termination: `exit`

- system call used to **terminate voluntarily a process**
- terminate the calling process
- specify an exit code
  - 0 (zero) exit code considered successfully termination
  - anything else considered erroneously termination
- exit code is kept until the parent process asks for it
- example
  - `exit(0);`
  - `exit(1);`

Process Termination: `exit`

- system call used to **terminate voluntarily a process**
- terminate the calling process
- specify an exit code
  - 0 (zero) exit code considered successfully termination
  - anything else considered erroneously termination
- exit code is kept until the parent process asks for it
- example
  - `exit(0);`
  - `exit(1);`

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Process Termination: `exit`

- system call used to **terminate voluntarily a process**
- terminate the calling process
- specify an exit code
  - 0 (zero) exit code considered successfully termination
  - anything else considered erroneously termination
- exit code is kept until the parent process asks for it
- example
  - `exit(0);`
  - `exit(1);`

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Process Termination: `exit`

- system call used to **terminate voluntarily a process**
- terminate the calling process
- specify an exit code
  - 0 (zero) exit code considered successfully termination
  - anything else considered erroneously termination
- exit code is kept until the parent process asks for it
- example
  - `exit(0);`
  - `exit(1);`

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Process Termination: `exit`

- system call used to **terminate voluntarily a process**
- terminate the calling process
- specify an exit code
  - 0 (zero) exit code considered successfully termination
  - anything else considered erroneously termination
- exit code is kept until the parent process asks for it
- example
  - `exit(0);`
  - `exit(1);`

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Process Termination: `exit`

- system call used to **terminate voluntarily a process**
- terminate the calling process
- specify an exit code
  - 0 (zero) exit code considered successfully termination
  - anything else considered erroneously termination
- exit code is kept until the parent process asks for it
- example
  - `exit(0);`
  - `exit(1);`

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Wait For Termination: wait and waitpid

- system calls used by a process to **wait for the termination of its children**
- return the exit code of the terminated child
- example

```
int child_status;

wait(&child_status);

printf("Child process terminated with code %d\n",
        WEXITSTATUS(status));
```

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Relationship Between `wait` and `exit`

- a way to **synchronize two processes**' execution (parent and child)
- a simple way to **communicate between processes** (parent and child)
- when a (parent) process terminates
  - all its children get as their new parent a system process
- when a (child) process terminates before its parent
  - its state is said to be **zombie** and
  - its exit state is maintained by OS until its parent process asks for it or terminates

Relationship Between `wait` and `exit`

- a way to **synchronize two processes**' execution (parent and child)
- a simple way to **communicate between processes** (parent and child)
- when a (parent) process terminates
  - all its children get as their new parent a system process
- when a (child) process terminates before its parent
  - its state is said to be **zombie** and
  - its exit state is maintained by OS until its parent process asks for it or terminates

Relationship Between `wait` and `exit`

- a way to **synchronize two processes**' execution (parent and child)
- a simple way to **communicate between processes** (parent and child)
- when a (parent) process terminates
  - all its children get as their new parent a system process
    - on older systems: the *init* process, having pid = 1
    - on newer systems: a per user *init* process
- when a (child) process terminates before its parent
  - its state is said to be **zombie** and
  - its exit state is maintained by OS until its parent process asks for it or terminates

Relationship Between `wait` and `exit`

- a way to **synchronize two processes**' execution (parent and child)
- a simple way to **communicate between processes** (parent and child)
- when a (parent) process terminates
  - all its children get as their new parent a system process
    - on older systems: the *init* process, having pid = 1
    - on newer systems: a per user *init* process
- when a (child) process terminates before its parent
  - its state is said to be **zombie** and
  - its exit state is maintained by OS until its parent process asks for it or terminates

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Relationship Between `wait` and `exit`

- a way to **synchronize two processes**' execution (parent and child)
- a simple way to **communicate between processes** (parent and child)
- when a (parent) process terminates
  - all its children get as their new parent a system process
    - on older systems: the *init* process, having pid $= 1$
    - on newer systems: a per user *init* process
- when a (child) process terminates before its parent
  - its state is said to be **zombie** and
  - its exit state is maintained by OS until its parent process asks for it or terminates

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Relationship Between `wait` and `exit`

- a way to **synchronize two processes**' execution (parent and child)
- a simple way to **communicate between processes** (parent and child)
- when a (parent) process terminates
  - all its children get as their new parent a system process
    - on older systems: the *init* process, having pid $= 1$
    - on newer systems: a per user *init* process
- when a (child) process terminates before its parent
  - its state is said to be **zombie** and
  - its exit state is maintained by OS until its parent process asks for it or terminates

Relationship Between `wait` and `exit`

- a way to **synchronize two processes**' execution (parent and child)
- a simple way to **communicate between processes** (parent and child)
- when a (parent) process terminates
  - all its children get as their new parent a system process
    - on older systems: the *init* process, having pid $= 1$
    - on newer systems: a per user *init* process
- when a (child) process terminates before its parent
  - its state is said to be **zombie** and
  - its exit state is maintained by OS until its parent process asks for it or terminates

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Relationship Between `wait` and `exit`

- a way to **synchronize two processes**' execution (parent and child)
- a simple way to **communicate between processes** (parent and child)
- when a (parent) process terminates
  - all its children get as their new parent a system process
    - on older systems: the *init* process, having pid $= 1$
    - on newer systems: a per user *init* process
- when a (child) process terminates before its parent
  - its state is said to be **zombie** and
  - its exit state is maintained by OS until its parent process asks for it or terminates

Relationship Between `wait` and `exit`

- a way to **synchronize two processes**' execution (parent and child)
- a simple way to **communicate between processes** (parent and child)
- when a (parent) process terminates
  - all its children get as their new parent a system process
    - on older systems: the *init* process, having pid $= 1$
    - on newer systems: a per user *init* process
- when a (child) process terminates before its parent
  - its state is said to be **zombie** and
  - its exit state is maintained by OS until its parent process asks for it or terminates

Let’s practice!

You are given the following C code and are required to:

1. Draw the process hierarchy corresponding to the processes created by the code below.
2. Specify the number of times each *printf* is executed, supposing every instruction is executed successfully.

```c
printf ("[1] Hello world!\n");

pid = fork();

printf ("[2] Hello world!\n");

pid = fork();

printf ("[3] Hello world!\n");

if (pid == 0) {
    execlp("ps", "ps", 0);
    printf ("[4] Hello world!\n");
}

fork();

printf ("[5] Hello world!\n");
```

Outline

Shell Basic Code (Functionality)

```
char **cmdline; // it must be build like argv param of main
while (TRUE) {
    display_prompt_on_screen();
    cmdline = read_cmd_line();

    pid = fork();       // creates a new process
    if (pid < 0) {
        perror("canot creat a new process");
        continue;
    }

    if (pid == 0)
        execvp(cmdline[0], cmdline);
    else
        waitpid(pid, NULL, 0);
}
```

Standard Input Redirection

- command line

```
cat < file.txt
```

- STDIN redirection in C program

```
pid = fork();

if (pid > 0) {
    // parent
} else {
    // child
    fd = open("file.txt", O_RDONLY);
    dup2(fd, 0);
    close(fd);

    execlp("cat", "cat", 0);
    perror("execl has not succeded");
}
```

Standard Output Redirection

- command line

```
ls > file.txt
```

- STDIN redirection in C program

```
pid = fork();

if (pid > 0) {
    // parent
} else {
    // child
    fd = creat("file.txt", 0600);
    dup2(fd, 1);
    close(fd);

    execlp("ls", "ls", 0);
    perror("execl has not succeded");
}
```

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Outline

ps Command

- displays a snapshot of the active processes in the system
- `ps -l -u acolesa --forest`

```
mc - linux:~/school/os
File  Edit  View  Terminal  Tabs  Help
acolesa@linux:~/school/os$
acolesa@linux:~/school/os$ ps -l -u acolesa --forest
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
4 S  1000 17351  3772  0  75   0 -  5293 -      ?        00:00:00 x-session-manag
1 S  1000 17394 17351  0  75   0 -  1034 429496 ?        00:00:00  \_ ssh-agent
0 S  1000 26952     1  0  75   0 -  8821 -      ?        00:00:00 gimp-2.2
0 S  1000 26953 26952  0  78   0 -  3418 429496 ?        00:00:00  \_ script-fu
0 S  1000 24740     1  0  78   0 -  1040 wait   ?        00:00:00 soffice
0 S  1000 24754 24740  0  75   0 - 50102 stext  ?        00:00:06  \_ soffice.bin
1 S  1000 24621     1  0  82   0 -  7115 429496 ?        00:00:00 kio_uiserver
1 S  1000 24469     1  0  75   0 -  6674 -      ?        00:00:00 kded
1 S  1000 24463     1  0  77   0 -  6187 429496 ?        00:00:00 dcopserver
1 S  1000 24459     1  0  76   0 -  6251 429496 ?        00:00:00 kdeinit
1 S  1000 24467 24459  0  75   0 -  6382 -      ?        00:00:00  \_ klauncher
```

ps Command

- displays a snapshot of the active processes in the system
- `ps -l -u acolesa --forest | grep -v '?'`

```
acolesa@linux:~/school/os$
acolesa@linux:~/school/os$ ps -l -u acolesa --forest | grep -v '?'
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S  1000 24478 24457  0  75   0 -  1744 -      pts/3    00:00:00  \_ bash
0 S  1000 17821 17819  0  75   0 -  1747 wait   pts/0    00:00:00  \_ bash
0 S  1000 24111 17821  0  75   0 -  1733 429496 pts/0    00:00:00      \_ mc
0 S  1000 24113 24111  0  75   0 -  1747 wait   pts/1    00:00:00          \_ bash
0 R  1000 27214 24113  0  77   0 -   851 -      pts/1    00:00:00              \_ ps
0 S  1000 27215 24113  0  75   0 -   712 pipe_w pts/1    00:00:00              \_ grep
acolesa@linux:~/school/os$
```

ps Command

- displays a snapshot of the active processes in the system
- `ps -l -e --forest | head -n 50`

```
mc - linux:~/school/os
File  Edit  View  Terminal  Tabs  Help
acolesa@linux:~/school/os$
acolesa@linux:~/school/os$ ps -l -e --forest | head -n 50
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
4 S     0     1     0  0  75   0 -   486 -      ?        00:00:01 init
1 S     0     2     1  0 -40   - -     0 migrat ?        00:00:00 migration/0
1 S     0     3     1  0  94  19 -     0 ksofti ?        00:00:00 ksoftirqd/0
1 S     0     4     1  0 -40   - -     0 migrat ?        00:00:00 migration/1
1 S     0     5     1  0  94  19 -     0 ksofti ?        00:00:00 ksoftirqd/1
5 S     0     6     1  0  70  -5 -     0 worker ?        00:00:00 events/0
1 S     0     7     1  0  70  -5 -     0 worker ?        00:00:00 events/1
1 S     0     8     1  0  70  -5 -     0 worker ?        00:00:00 khelper
1 S     0     9     1  0  72  -5 -     0 worker ?        00:00:00 kthread
1 S     0    13     9  0  70  -5 -     0 worker ?        00:00:00  \_ kblockd/0
1 S     0    14     9  0  70  -5 -     0 worker ?        00:00:00  \_ kblockd/1
```

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

`top` and `htop` Commands

- display a continuously updated list of processes and their on-line scheduling

proc File System

- It is a pseudo file system
- It is mounted in `/proc`
- It is used by the OS to display information about processes
  - each process has a directory named with the process id
  - reading this information is similar to reading any other files and dirs

Outline

What we talked about

- process definition
- process states and state transitions
  - running, ready, blocked, terminated
- system calls to create and terminate a process
- Linux system calls related to processes
  - fork
  - exec
  - exit
  - wait

What we talked about

- process definition
- process states and state transitions
  - running, ready, blocked, terminated
- system calls to create and terminate a process
- Linux system calls related to processes
  - fork
  - exec
  - exit
  - wait

What we talked about

- process definition
- process states and state transitions
  - running, ready, blocked, terminated
- system calls to create and terminate a process
- Linux system calls related to processes
  - fork
  - exec
  - exit
  - wait

What we talked about

- process definition
- process states and state transitions
  - running, ready, blocked, terminated
- system calls to create and terminate a process
- Linux system calls related to processes
  - fork
  - exec
  - exit
  - wait

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

What we talked about

- process definition
- process states and state transitions
  - running, ready, blocked, terminated
- system calls to create and terminate a process
- Linux system calls related to processes
  - fork
  - exec
  - exit
  - wait

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

What we talked about

- process definition
- process states and state transitions
  - running, ready, blocked, terminated
- system calls to create and terminate a process
- Linux system calls related to processes
  - fork
  - exec
  - exit
  - wait

What we talked about

- process definition
- process states and state transitions
  - running, ready, blocked, terminated
- system calls to create and terminate a process
- Linux system calls related to processes
  - fork
  - exec
  - exit
  - wait

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

What we talked about

- process definition
- process states and state transitions
  - running, ready, blocked, terminated
- system calls to create and terminate a process
- Linux system calls related to processes
  - fork
  - exec
  - exit
  - wait

What we talked about

- process definition
- process states and state transitions
  - running, ready, blocked, terminated
- system calls to create and terminate a process
- Linux system calls related to processes
  - fork
  - exec
  - exit
  - wait

Lessons Learned

- process is a virtualization and isolation concept
  - virtualize an entire compute for a program's execution
  - isolate one execution by another
- process states
  - running: the desired one
  - ready: exists due to limited no of CPUs; is transparent to processes
  - blocked: triggered explicitly by a process due to a blocking syscall
- create a new process
  - `fork()` called by the parent to create an identical child process
  - `exec()` called by the child to load new code
- every process terminates with `exit()`

Lessons Learned

- process is a virtualization and isolation concept
  - virtualize an entire compute for a program's execution
  - isolate one execution by another
- process states
  - running: the desired one
  - ready: exists due to limited no of CPUs; is transparent to processes
  - blocked: triggered explicitly by a process due to a blocking syscall
- create a new process
  - `fork()` called by the parent to create an identical child process
  - `exec()` called by the child to load new code
- every process terminates with `exit()`

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Lessons Learned

- process is a virtualization and isolation concept
  - virtualize an entire compute for a program's execution
  - isolate one execution by another
- process states
  - running: the desired one
  - ready: exists due to limited no of CPUs; is transparent to processes
  - blocked: triggered explicitly by a process due to a blocking syscall
- create a new process
  - `fork()` called by the parent to create an identical child process
  - `exec()` called by the child to load new code
- every process terminates with `exit()`

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Lessons Learned

- process is a virtualization and isolation concept
  - virtualize an entire compute for a program's execution
  - isolate one execution by another
- process states
  - running: the desired one
  - ready: exists due to limited no of CPUs; is transparent to processes
  - blocked: triggered explicitly by a process due to a blocking syscall
- create a new process
  - `fork()` called by the parent to create an identical child process
  - `exec()` called by the child to load new code
- every process terminates with `exit()`

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Lessons Learned

- process is a virtualization and isolation concept
  - virtualize an entire compute for a program's execution
  - isolate one execution by another
- process states
  - running: the desired one
  - ready: exists due to limited no of CPUs; is transparent to processes
  - blocked: triggered explicitly by a process due to a blocking syscall
- create a new process
  - `fork()` called by the parent to create an identical child process
  - `exec()` called by the child to load new code
- every process terminates with `exit()`

Lessons Learned

- process is a virtualization and isolation concept
  - virtualize an entire compute for a program's execution
  - isolate one execution by another
- process states
  - running: the desired one
  - ready: exists due to limited no of CPUs; is transparent to processes
  - blocked: triggered explicitly by a process due to a blocking syscall
- create a new process
  - `fork()` called by the parent to create an identical child process
  - `exec()` called by the child to load new code
- every process terminates with `exit()`

Lessons Learned

- process is a virtualization and isolation concept
  - virtualize an entire compute for a program's execution
  - isolate one execution by another
- process states
  - running: the desired one
  - ready: exists due to limited no of CPUs; is transparent to processes
  - blocked: triggered explicitly by a process due to a blocking syscall
- create a new process
  - `fork()` called by the parent to create an identical child process
  - `exec()` called by the child to load new code
- every process terminates with `exit()`

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Lessons Learned

- process is a virtualization and isolation concept
  - virtualize an entire compute for a program's execution
  - isolate one execution by another
- process states
  - running: the desired one
  - ready: exists due to limited no of CPUs; is transparent to processes
  - blocked: triggered explicitly by a process due to a blocking syscall
- create a new process
  - `fork()` called by the parent to create an identical child process
  - `exec()` called by the child to load new code
- every process terminates with `exit()`

Lessons Learned

- process is a virtualization and isolation concept
  - virtualize an entire compute for a program's execution
  - isolate one execution by another
- process states
  - running: the desired one
  - ready: exists due to limited no of CPUs; is transparent to processes
  - blocked: triggered explicitly by a process due to a blocking syscall
- create a new process
  - `fork()` called by the parent to create an identical child process
  - `exec()` called by the child to load new code
- every process terminates with `exit()`

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Lessons Learned

- process is a virtualization and isolation concept
  - virtualize an entire compute for a program's execution
  - isolate one execution by another
- process states
  - running: the desired one
  - ready: exists due to limited no of CPUs; is transparent to processes
  - blocked: triggered explicitly by a process due to a blocking syscall
- create a new process
  - `fork()` called by the parent to create an identical child process
  - `exec()` called by the child to load new code
- every process terminates with `exit()`

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Lessons Learned

- process is a virtualization and isolation concept
  - virtualize an entire compute for a program's execution
  - isolate one execution by another
- process states
  - running: the desired one
  - ready: exists due to limited no of CPUs; is transparent to processes
  - blocked: triggered explicitly by a process due to a blocking syscall
- create a new process
  - `fork()` called by the parent to create an identical child process
  - `exec()` called by the child to load new code
- every process terminates with `exit()`