# *Software Lab*

## *Processes and threads*

Roberto Farina
roberto.farina@cefriel.it

# Summary

- Processes: fundamental concepts
  - Commands
  - Creation and termination
  - Wait
- Threads: fundamental concepts
  - Creation and communication
  - Termination and deleting
  - Critical sections
  - News about implementation

# Basic concepts

- Process hierarchy
  - In Linux hierarchy starts with `sched`, the scheduler process
    - In recent versions is `init`

- `PID` = Process ID
  - Unique process identifier
- `PPID` = Parent Process ID
  - Unique parent process identifier

 

# Linux process states

- `R` Running or ready for execution
- `S` Interruptible wait (waiting for an event)
- `D` Uninterruptible wait (e.g.: I/O operations)
- `T` Stopped
- `X` Dead (it should never be seen)
- `Z` Defunct (zombie process)

- `<` High priority (niceness)
- `N` Low priority (niceness)
- `+` In the foreground processes group

# Commands

- `ps`
  - Lists processes in the system and their properties
  - It is possible to select information to show
- `pstree`
  - Shows the system process hierarchy
  - It is possible to change the root of the tree
- `top`
  - Not a static snapshot but a run-time view of processes
  - It is possible to work on the single process
- `jobs`
  - Shows the hierarchy starting from the current shell

# First steps

- ALWAYS read man pages!
- Most of the primitives to handle processes are defined in `unistd.h`
- PID uniquely identifies processes
  - Standard POSIX: up to 32000
  - From 2.6.x: up to $10^9$
  - ALWAYS use the data type `pid_t`
    - `#include <sys/types.h>`
- `getpid()` returns the PID of the invoking process
- `getppid()` to get the parent process PID

 

# Creation

- Two different techniques
  - `system`
    - Allows the execution of an external command
    - Very simple but dangerous for security
  - `fork`
    - Allows to create child processes (more than one)
    - Harder to use
    - More flexible, quick and secure
- When you create a child process, resources are copied (file descriptors, …)
  - A modification in the child process doesn't affect other related processes

# `system`

- `system` creates a process that executes the standard shell (`/bin/sh`) and handles the command passed as a parameter
  - `system("ls -l /");`
- Returns the exit code of the executed program
  - 127 if the shell cannot be executed
  - -1 for other errors
- Inherits shell limitations
- It is preferred to use `fork`

# `fork`

- Creates a child process as an EXACT COPY of the current process
- Returns different values to different processes
  - Child process' PID or –1 (`errno`) in the parent
  - 0 to the child process
- Execution of both processes continues from the point `fork` was called
- In Linux it is implemented using COPY-ON-WRITE for pages
  - Only the cost for page table duplication and creation of data structure

# exec

- Substitutes the executing program in a process with another one
  - Stops the execution of previous program
  - The new execution starts from the beginning
- `exec` doesn’t return if no errors
- `fork + exec = spawn`

- `system` internally use
  - `fork`
  - `exec(“/bin/sh” …)`

# `exec` families

- With the `p` letter, it accepts a program name which is searched in the current PATH
  - `execvp`, `execlp`
- With the `v` letter, it accepts an argument list for the new program as a NULL-terminated array of string pointers
  - `execcv`, `execvp`
- With the `l` letter, it accepts an argument list with the standard C mechanism (`varargs`)
- With the `e` letter, it accepts as an additional argument an array of environment variables as a NULL-terminated array of string pointers
  - `execve`, `execle`

# Niceness

- It is possible to specify the importance of a process through a numeric value that modifies its priority
  - The bigger the nice value, the smaller the priority
  - Varies from -20 to 19

- Commands:
  - `nice -n <valore> <comando>`
  - `renice <valore> <pid>`
- Call:
  - `nice(<valore>);`

# Signals: main concepts

- Messages for communication and process handling
- Asynchronous: the current operation is stopped when a signal is received and the message is handled
- The corresponding action depends on the signal disposition
  - Default disposition: standard behavior when the process doesn’t handle the signal
  - Signal handler: process function invoked when the signal is received
    - It is possible to mask signals

 

# Important signals

- 6 (SIGABRT) Causes the process termination and the generation of a core file
- 9 **(SIGKILL)** Forced process termination
- 10 *(SIGUSR1) The behavior can be defined*
- 11 (SIGSEGV) Access to a non valid memory segment
- 12 *(SIGUSR2) The behavior can be defined*
- 15 (SIGTERM) Notification of termination
- 17 (SIGCHLD) Child termination
- 18 (SIGCONT) Continues the execution if stopped
- 19 **(SIGSTOP)** Stops the execution

# Using signals

- The system sends signals as answers to specific actions
  - SIGSEGV, SIGBUS, SIGFPE, ...

- **`kill`** command
  - Sends a signal (not only KILL signal)
    - Without parameters, it sends SIGTERM
  - **`kill -<signal> <pid>|<%jobid>`**
    - E.g.: **`kill -9 812,kill -KILL 812,kill -9 %1`**

- **`kill`** function
  - **`kill(<pid>, <signal>);`**
    - E.g.: **`kill(child_pid, SIGTERM);`**

 

# Changing disposition

- `#include <signal.h>`
  - The signal-number association is defined in `/usr/include/bits/signum.h`
- `sigaction` structure
  - `sa_handler` field to indicate the signal handling routine
- `sigaction` function
  - Signal to modify disposition for
  - Data structure that indicates the new handler
  - Data structure to save the old disposition
    - NULL if it is not needed to save the old one

 

# sigaction structure

- `sa_handler` field
  - `SIG_DFL`: use the default disposition
  - `SIG_IGN`: ignore the signal (not with every signal)
  - Function pointer: accepts a numeric parameter (signal number) and returns `void`
- Asynchronous mechanism
  - Can take the process to a non-stable status
    - Never call I/O primitives or library functions within a signal handler
  - Interruptible by another signal
    - `sig_atomic_t` to grant assignment operations are executed in a single instruction

# wait

- `wait` system call family to wait for the termination of a child process and to obtain information about how the child has exited
- They are needed in order to force a sequence in the execution of operations
  - Not against parallelism hypothesis
  - Processes are scheduled independently
- Four different calls

# wait

- BLOCKING call
- Integer pointer for information coming from child process
- **WEXITSTATUS** macro to extract information about process exit status
- **WIFSIGNALED** macro returns TRUE if child process termination was caused by a not handled signal
- **WIFEXITED** macro to determine if the process has exited normally (**exit** or **return**) or the termination was caused by a not handled signal
  - **WTERMSIG** to extract the signal number

 

# Zombie processes

- **Definition**: a terminated process but still not cleaned (still in the system)
  - E.g.: a terminated child process on which the parent never invoked `wait`
  - The child process is not removed from the system
    - Termination info would be lost
  - It is not necessary to intercept termination
    - `wait` call automatically cleans the process
- A parent terminates without invoking `wait`: zombie processes are inherited by `init` which goes for the clean

# wait calls

- `waitpid`: waiting a specific process
- `wait3`: returns statistics about CPU usage by the child process
- `wait4`: it is possible to specify additional information about the process to wait for and to tune its behavior with options
  - `WNOHANG` to make the call NON BLOCKING if no process has terminated yet

# Threads

- Finer grained execution unit than processes
  - Threads exist within a process
  - Every thread executes the SAME program (a different section) in the SAME process
- Every thread shares same resources
  - File descriptors, memory space, ...
  - An action in a thread can modify the behavior of other threads
- If a thread calls an `exec` function, every thread terminates and the new program is executed

# Thread details

- GNU/Linux implements POSIX standard API for threads (`pthread`)
  - From 2.4.20 it implements NPTL
  - Every data type and function for threads are defined in `<pthread.h>`
  - Not included in the standard library, but in `libpthread.so`
    - It is necessary to specify this library when linking

      `-lpthread`
- Every thread is identified by a thread ID
  - ALWAYS use `pthread_t`

 

# Creation and termination

- Every thread executes a specific function: the thread function
- A thread function accepts a `void*` parameter and returns a `void*` value
- `pthread_create`
  - A pointer to a `pthread_t` variable
  - A pointer to a thread attribute object
  - A pointer to a thread function
  - A thread argument
- A thread terminates if:
  - the thread function terminates
  - the thread invokes `pthread_exit`

# Thread scheduling

- Invocation of `pthread_create` terminates immediately and the execution continues from the next instruction
- Scheduling is asynchronous
  - NEVER base on the execution order
- Problem: the main thread terminates before others
  - Data structures to which other threads refer could be removed from memory
- Solution: `pthread_join`
  - Similar to `wait` for processes

# Communication

- It is possible to give parameters to the thread through the `void*` argument
  - It is possible to use the same function for more than one thread
  - Usually the argument is a pointer to a data structure or an array
  - It is necessary to use casting inside the thread
- It is possible to catch the thread return value using a non NULL argument as second parameter of `pthread_join`
  - It is necessary to use casting

 

# Useful functions

- `pthread_equal(<pthread_t>, <pthread_t>)` to test two thread IDs
- `pthread_self()` returns the thread ID of the thread it is invoked in
  - Useful to avoid a thread to invoke a join on itself
    - `EDEADLK`

```
if (!pthread_equal(pthread_self(), ptid))
  pthread_join(ptid, NULL);
```

# Attributes

- Mechanism to specialize the behavior of a thread
  - If the second parameter of `pthread_create` is NULL use the default mechanism
- In most cases a single attribute is interesting: the detach state
  - Other attributes are typically used in real-time systems

 

# Joinable vs detach

- Joinable thread: when the function terminates the thread is not automatically cleaned (same as zombie processes)
- Detach thread: the thread is automatically cleaned when the function terminates
  - It is impossible to synchronize with it and to read its return value
  - **`pthread_attr_setdetach_state`**
    - **`PTHREAD_CREATE_DETACHED`** as second parameter
- It is possible to transform a joinable thread in a detached invoking **`pthread_detach`**
  - The contrary is not possible

# Attributes modification

1. Create a `pthread_attr_t` object
2. Invoke `pthread_attr_init` passing a pointer to the object previously created
3. Modify created object
4. Pass this object to `pthread_create`
5. Invoke `pthread_attr_destroy` to destroy the object
   - The variable is not removed, can be initialized once again

# Cancellation

- **Definition:** termination request by another thread
- Invoke `pthread_cancel` passing the thread ID of which termination is requested
- It is possible to invoke `pthread_join` on a cancelled thread in order to free resources
- The return value of a cancelled thread is `PTHREAD_CANCELED`

# Cancellation states

- A thread can contain code that needs to be executed in an “all-or-nothing” manner
  - Resources allocation: a cancellation could not let you free allocated resources
- It is possible to control cancellation
  - Asynchronously cancelable: can be cancelled in every moment
  - Synchronously cancelable: cancellation requests are queued and processed when specific points in the code are reached (cancellation points)
  - Uncancelable: cancellation requests are ignored

# Sync vs Async

- `pthread_setcanceltype()`
  - Works on the thread that invokes it
  - `PTHREAD_CANCEL_ASYNCHRONOUS` to set asynchronous mode
  - `PTHREAD_CANCEL_DEFERRED` to set synchronous mode (restore defaults)
- Cancellation points:
  - `pthread_testcancel` to process a pending cancellation request
    - To be invoked periodically in long computations
  - `man pthread_cancel` lists other cancellation points

# Uncancelable

- `pthread_setcanceltype(...);`
  - `PTHREAD_CANCEL_DISABLE`
  - `PTHREAD_CANCEL_ENABLE`
- It is possible to implement critical sections disabling cancellation
- It is important to restore the original state, which can be different from `PTHREAD_CANCEL_ENABLE` (do not use it unconditionally)

# Thread-specific data

- It is possible to define an independent memory space
  - E.g.: to create copies of variables in order to modify them without affecting the behavior of other threads
- It is possible to create an arbitrary number of data item
  - Everyone is a `void*`
  - They are referenced by a key
    - Every thread use its key to access the specific copy

 

# Creation and use of the key

- `pthread_key_create` to create the key
  - Pointer to a variable `pthread_key_t` to be used to access one's own copy
  - Pointer to a cleanup function
    - Automatically invoked when the thread terminates
    - Invoked upon cancellation requests too
    - Not invoked if thread-specific data is NULL
    - The local copy of the variable is passed
- `pthread_setspecific()` to set one's own value in the local variable
- `pthread_getspecific()` to read the value

# Cleanup handlers

- Function invoked when a thread terminates
  - Not specific for every thread data item
- It accepts a single `void*` parameter
  - Specified upon handler registration
  - Useful to deallocate multiple instances of a resource
- A way to deallocate resources when a thread terminates or it is cancelled rather than terminating the execution of a particular code region
  - In normal circumstances the resource must be explicitly deallocated and the handler must be removed

 

# Registration and cancellation

- `pthread_cleanup_push` to register a cleanup handler
  - Pointer to the cleanup function
  - `void*` argument for the function
- `pthread_cleanup_pop` to cancel the registration of a cleanup handler
  - Balances the invocation of `pthread_cleanup_push`
  - Integer flag: if not zero, the function is executed and only subsequently the registration is cancelled

# News: NPTL

- Why?
  - The old solution will become less and less scalable
  - Doesn’t take into account modern processors characteristics
- Linux implements threads as processes
  - Not similar to processes created by fork: they don’t have a own address space but they share the same space of the main thread
  - Modifications on the scheduler

# Thread manager

- Created by the first invocation of `pthread_create`
- If a process receives a signal which thread manages it?
  - In Linux, threads are implemented as processes…
    - Usually signals are sent to the process of the main thread
- E.g.: a process executes a fork; the child executes a multithread program with an `exec` call
- The parent keeps the PID of the process that implements the main thread as the child ID
- `pthread_kill(<pthread_t>, <signal>)` to send signals to a specific thread