Operating Systems Lab (C+Unix)

Enrico Bini

University of Turin

Outline

Outline

Processes

- A process is an instance of an executing program
- In operating systems, a process is identified by a Process ID (PID)
- The command `ps` is used to view information on the processes

  ```
  man ps
  ```

- the command `top` shows a live update of CPU/mem consumed by processes

  ```
  top
  ```

- the command `kill` can send a "signal" to a process. One special signal is `SIGKILL` (more details on signals, later on)

  ```
  kill -KILL <some-PID>
  ```
  or
  ```
  kill -9 <some-PID>
  ```

- the command `kill` can also be used to stop or continue a process
  1. start a candidate process (a browser)
  2. get its PID
  3. `kill -STOP` PID
  4. try to use that application
  5. `kill -CONT` PID
  6. the application should be back to life

Process ID and Parent Process ID

- Processes are identified by PIDs. The system call

```
pid_t getpid(void);
```

  returns the PID of the calling process (`pid_t` is an integer type)

- each process has a parent: the process that created it. The function

```
pid_t getppid(void);
```

  returns the PID of the parent process (parent's PID = PPID)

- the PPID of each process represents the tree-like relationship of all processes on the system. The parent of each process has its own parent, and so on, going all the way back to process "`init`" (with PID=1), the ancestor of all processes

- `ps -Af | less` (shows all processes, full info)

- `pstree -p | less` (shows the tree of all processes)

- *test-getpid.c*

Process creation: `fork()`

- The `fork()` syscall allows a process (called "parent") to create a "child" process

```
#include <unistd.h>

pid_t fork(void);
```

- The child process is a copy of the parent
  - the OS makes a **copy of all memory segments** of the parent process: stack, BSS, and heap segments, I/O buffers included!!
  - the child executes over the copy: data modified by the child is not seen by the parent!!!
  - (sharing data among processes is possible, shown in last lecture)
- "fork": the parent process is split in two "branches"
- **SUPER IMPORTANT:** `fork()` returns two different values in child and parent processes!!!
  - in parent: the PID of the child on success (or -1 on error)
  - in child: it returns 0

Is the child or parent code?

- A frequent difficulty is in understanding what code we are writing: child? parent? both?

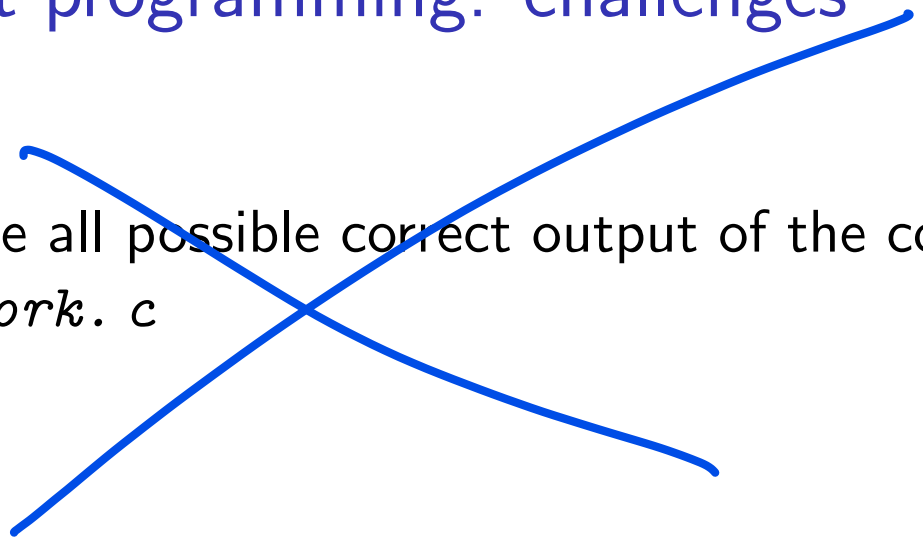
```
/* Executed only once */
if (fork()) {
    /* Executed by parent only */
} else {
    /* Executed by child only */
}
/* Executed twice: by both parent and child */
```

- Remember, the returned value of `fork()` is used to determine what process "we are":
  1. if returned 0, then we are in the child code
  2. if returned a positive number, we are in the parent code (and the value is the PID of the just created child)

test-fork.c

test-fork-buf.c

Concurrent programming: challenges



- What are all possible correct output of the code?
  test-fork.c

Concurrent programming: challenges

- What are all possible correct output of the code?
  test-fork. c
- Different executions are possible even if the same input is given. Why?
- The schedule of processes is a hidden input beyond our control.
- In concurrent programming

  sometimes correct = wrong

- A concurrent program must be correct **regardless** the order of execution of processes
- If some schedules generate not desired behaviors, then syncronization mechanisms (semaphores, etc.) must be explicitly added

Debuging processes by gdb

- It is also possible to attach gdb to a running process by
  `gdb -p <PID>`
- If the process is running some system call, you may need `sudo` superpowers by
  `sudo gdb -p <PID>`
- When debugging a program by gdb, if the program forks, do we follow the parent of the child process?
  - `(gdb) set follow-fork-mode parent`
  - `(gdb) set follow-fork-mode child`

Outline

Waiting for child termination by `wait(NULL)`

- The parent can wait for the termination of any child process by invoking the system call (better if the parent process always does wait for child processes)

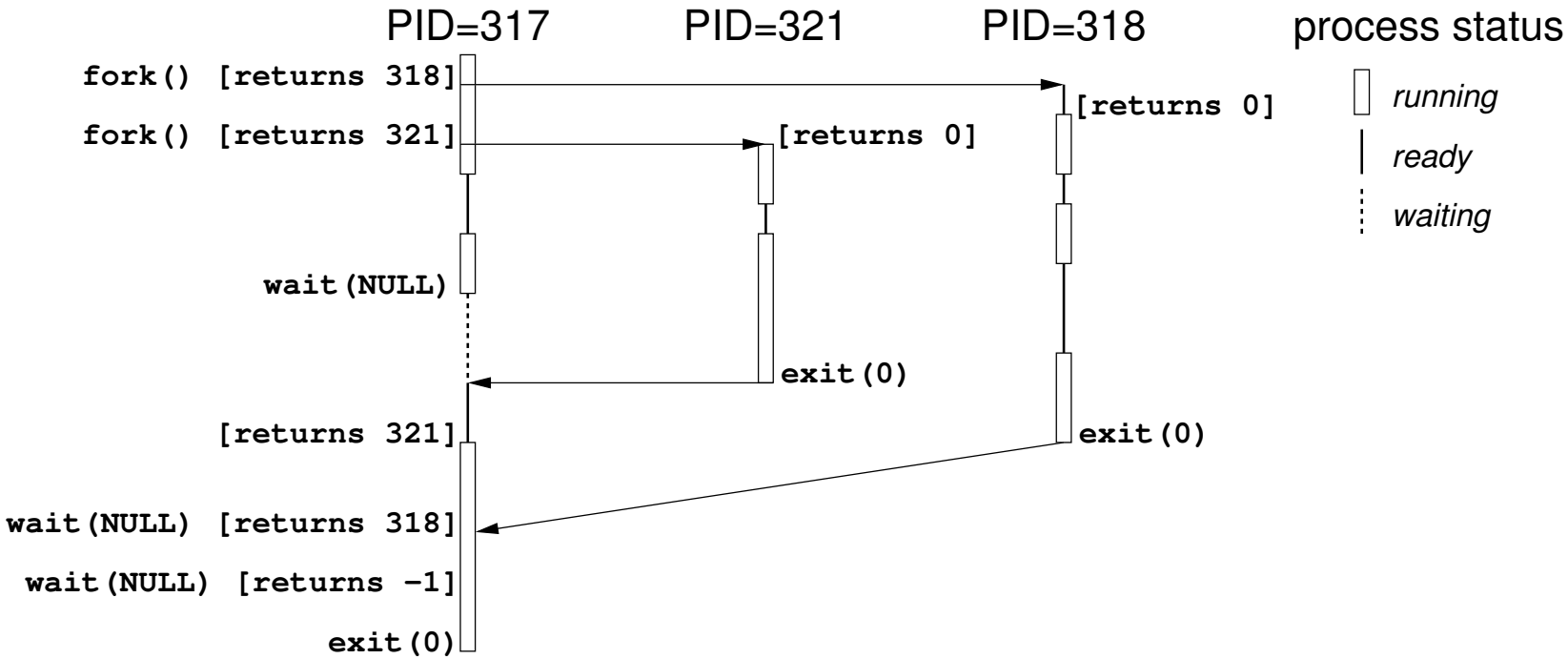
```
pid_t wait(NULL)
```

- `wait(NULL)` returns
  - `-1` if all child processes are terminated (or never had any child process)
    or
  - the PID of any terminated child process
- Standard code to wait for the termination of all child processes is

```
while (wait(NULL) != -1);
```

- *test-fork-for-wait.c*

`wait(NULL)`: possible interactions



`wait(NULL)` is a blocking system call

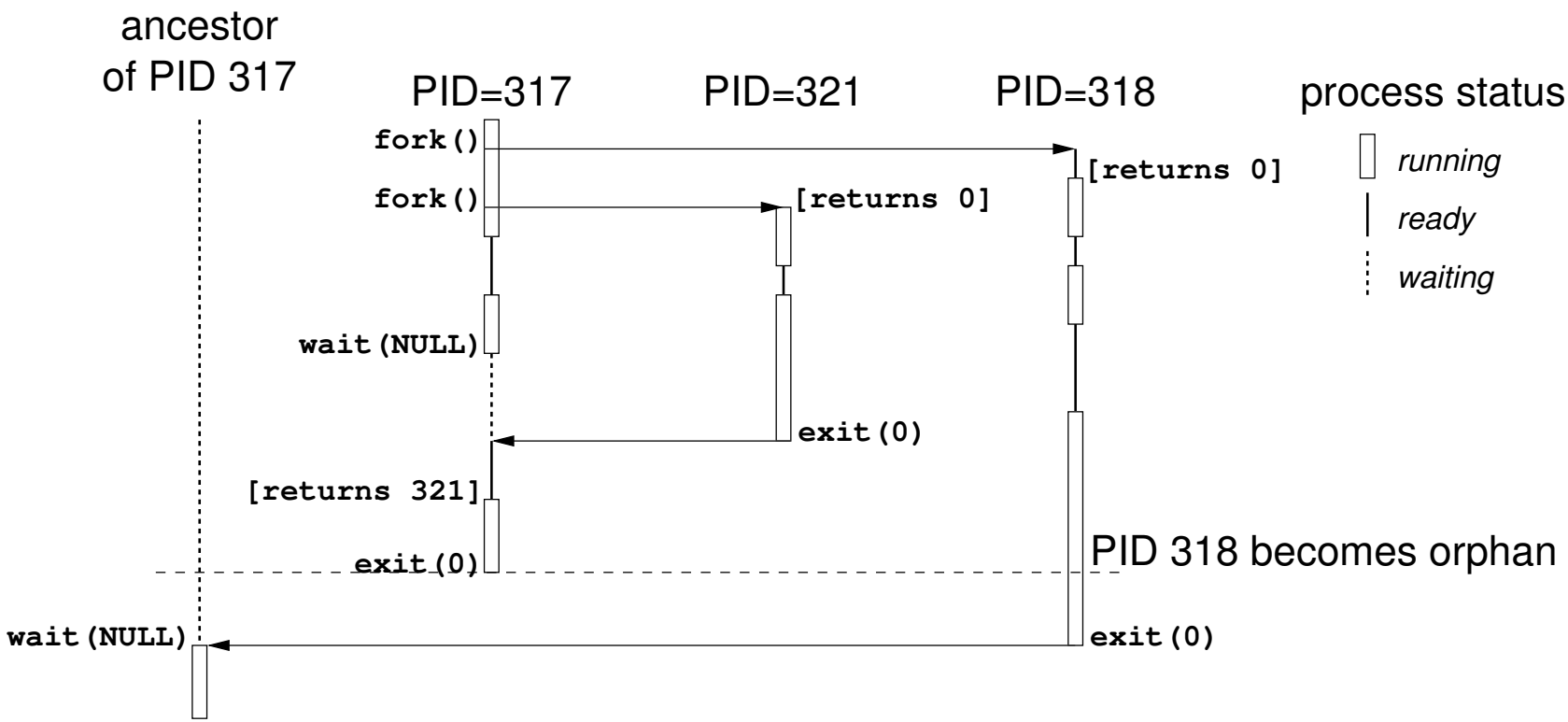
1. the parent process waits for any child process to terminate
2. it returns the PID of any terminated child
3. it returns `-1` if the process has no child

Process termination

- A process terminates and "returns a status" when
  1. it is returned from the `int main(...)` function by `return status`
  2. the system call `exit(status)` is invoked
  3. it catches a terminating signal, e.g. pressing `Ctrl+C` (details later)
- the returned value `status` gives information about the outcome of the program. It must be between 0 and 255
- two macros (defined in `stdlib.h`) may be used:
  - `EXIT_SUCCESS` (usually 0)
  - `EXIT_FAILURE` (usually 1)
- When a process terminates
  1. all streams are flushed, and all open file descriptors are closed
  2. a `SIGCHILD` signal is sent to the parent (more info about signals later)
  3. any child of the terminated process is assigned to a new parent (the granparent or `init` PID$=1$, depending on the OS)
  4. the resources (memory, open file descriptors) are released
  5. the `exit` status truncated to 8 bits (`& 0xFF`) is stored

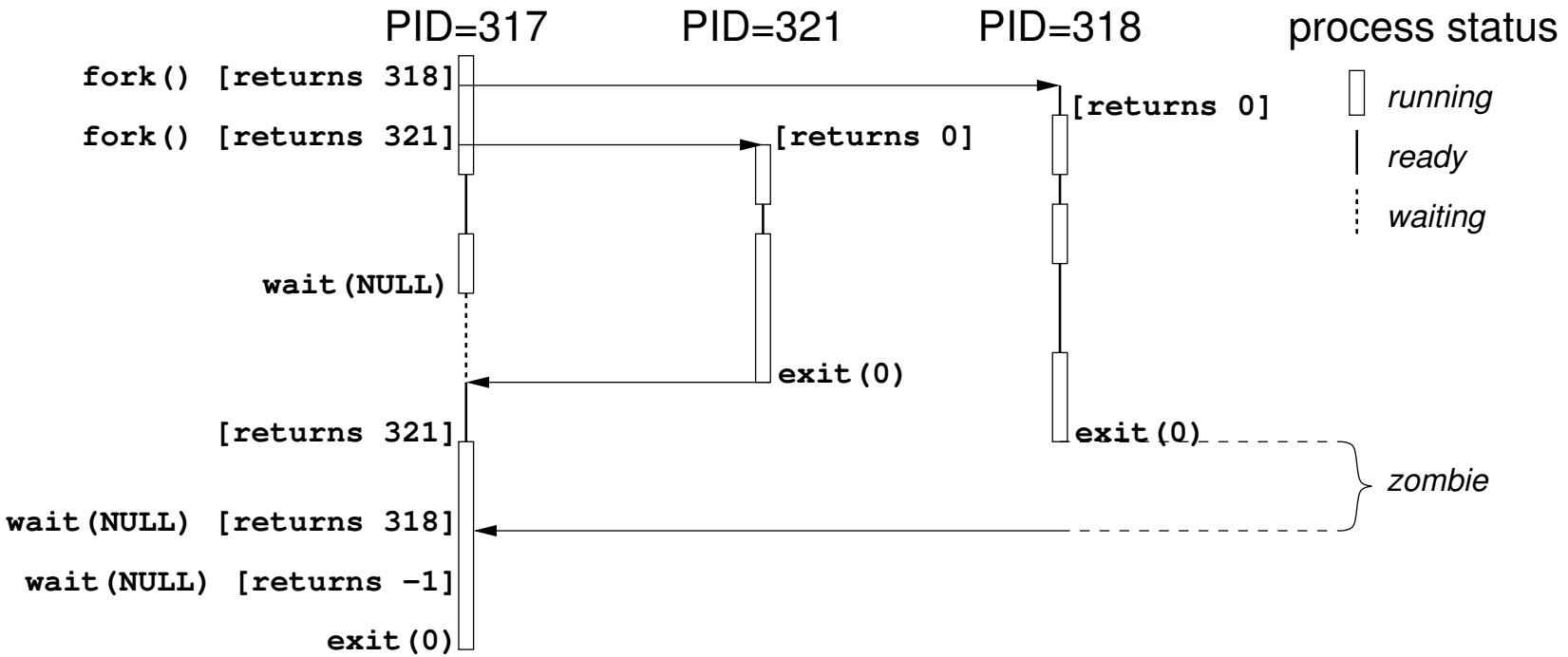
Orphans: parent process terminating before children

- The parent and the child processes are different and will terminate at different instants
- If a parent does not invoke `wait()` on all the child processes, it may terminate before the child (**not** recommended). All child processes become *orphans*.
- Orphan processes are adopted (by some ancestor process)



Zombies: process terminates before parent's `wait(...)`

- From termination to the parent's `wait()`, the process is a "zombie".



- If a child terminates, all resources are released, but an entry in the process table is kept with
  - its PID
  - its exit status, and
  - the statistics of the used resources
- This entry is held by the OS until the parent executes a `wait()`.

Zombies are not healthy

- A zombie cannot be killed by any signal (not even `SIGKILL`). They exist to makes sure that the parent can access the exit status by a `wait()`
- If the parent terminates without executing a `wait()` all its terminated child processes (zombies) also becomes orphans:
  - when the ancestor adopts zombie processes, it immediately executes as many `wait()` as needed to make the zombies R.I.P.
- An excessive number of zombies may fill the process table up by holding a PID, and prevents the creation of new processes
- Since zombies cannot be killed, the only way to remove them from the system is to kill the parent, which will trigger their adoption and the consequent `wait()`, which finally erases the zombies from the process list
- Why doesn't the OS free the child processes as soon as they terminate?

Zombies are not healthy

- A zombie cannot be killed by any signal (not even `SIGKILL`). They exist to makes sure that the parent can access the exit status by a `wait()`
- If the parent terminates without executing a `wait()` all its terminated child processes (zombies) also becomes orphans:
  - when the ancestor adopts zombie processes, it immediately executes as many `wait()` as needed to make the zombies R.I.P.
- An excessive number of zombies may fill the process table up by holding a PID, and prevents the creation of new processes
- Since zombies cannot be killed, the only way to remove them from the system is to kill the parent, which will trigger their adoption and the consequent `wait()`, which finally erases the zombies from the process list
- Why doesn't the OS free the child processes as soon as they terminate?
  `wait()` is a basic synchronization mechanism: `wait(NULL)` allows the parent to be certain that the returned PID terminated

Retrieving more information about the child termination

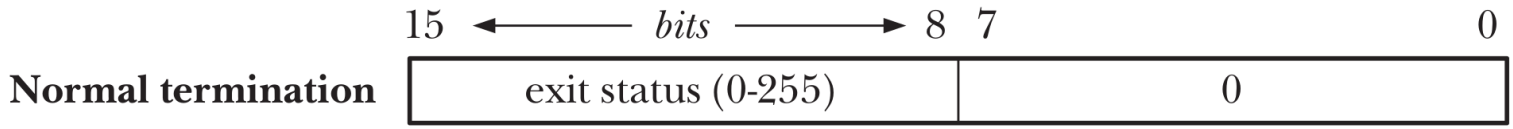
- The parent process can get information about the terminated child process by:

```
pid_t wait(int *status)
```

- A process invoking
  `child_pid = wait(&child_status);`
  checks if any child has terminated
  - if the process has no child, `wait()` returns -1 and `errno` is set to `ECHILD`
  - if the process has some terminated child, `wait()` immediately returns the PID of any terminated child and eliminates this child process from the list of children
  - If child processes exist, but none of them has terminated yet, the parent process moves to the *waiting* state, waiting for the first child to terminate

Format of returned child `status`

- Once the parent correctly returns from `wait(&status)` (meaning that a child has terminated), the variable `status` is filled with information about the child process
- the format of status is as follows



- the macro `WEXITSTATUS(status)` extracts the status from the value `status` written by `wait(&status)`
  - the macro `WEXITSTATUS(status)` is just

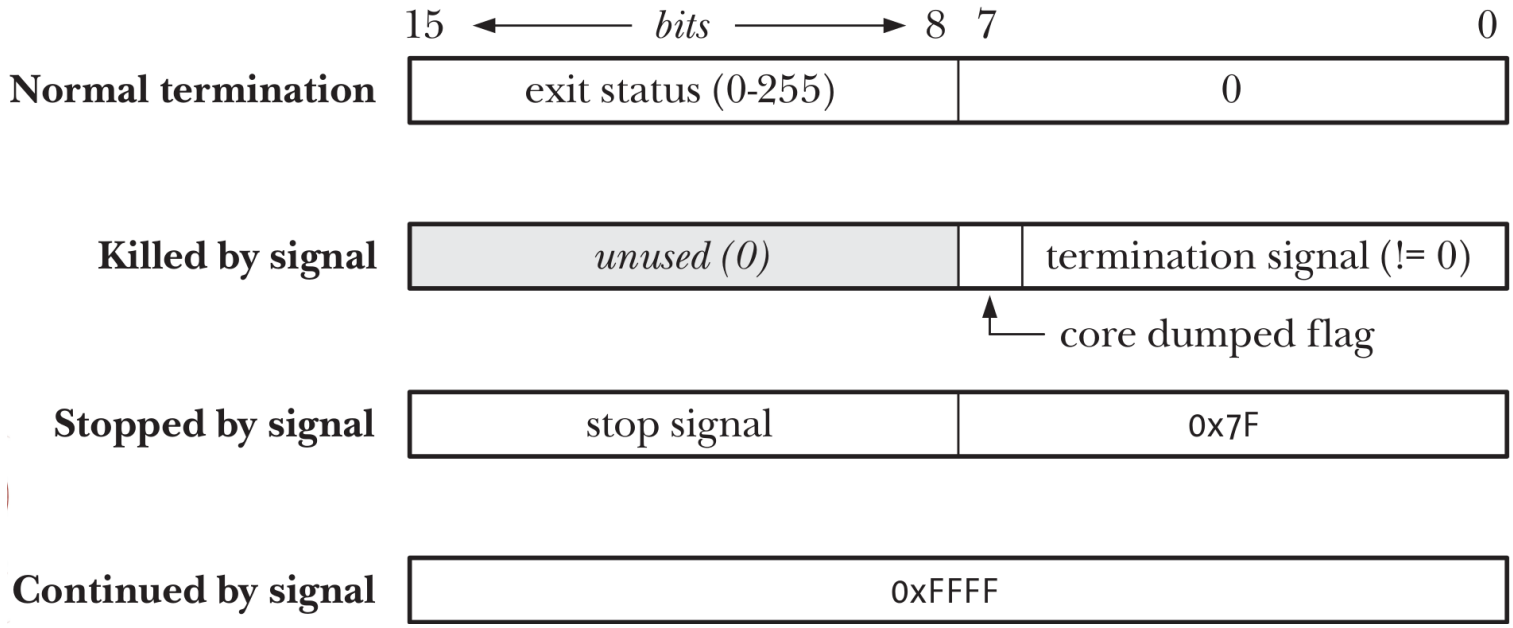
```
#define WEXITSTATUS(x) ((x) >> 8)
```

- *test-fork-wait.c*
  - We may confine the execution over a subset of cores by

```
taskset <affinity-mask> <command>
```

Extracting information from the returned status

- Once it is returned from `wait(&status)`, the value of `status` is filled and gives information about the status of the child



- macro exists (declared in `sys/wait.h`) to extract this information

`man 2 wait`

Waiting for a specific child process

- We showed that by calling `wait()` a parent waits for the completion of any child process
- To wait for a specific child process, the next system call can be used

```
pid_t waitpid(pid_t pid, int *status_child, int options)
```

- After the call to `waitpid(...)`, the parent process waits for the termination the child process `pid`. If `pid == -1`, it waits for any child process.
  `waitpid(-1,&status_child, 0)` is equivalent to `wait(&status_child)`
- The returned value is:
  - the PID of the process whose status is reported;
  - -1 if an error occured. If so, `errno` has the following values:
    - `ECHILD` no child with PID `pid` to wait for,
    - `EINVAL` invalid `option` argument

Options of `waitpid()`

- The following options can be specified as bitwise OR (`|`) of the following flags
  - `WNOHANG` "Wait NO HANGing": if no child process has terminated, `waitpid()` will **not wait** for the termination of the specified `pid`. Rather it continues, and it returns `0` to indicate this condition
    - `waitpid(ch_pid, &ch_stat, WNOHANG)` just checks termination, the parent process does `not` wait if child process `ch_pid` isn't terminated
    - the actual termination of a child process can then be handled by catching the signal `SIGCHILD`, sent to the parent any time a child process terminates
  - `WUNTRACED`: `waitpid()` returns also if the selected child processes have stopped (by some signal)
  - `WCONTINUED`: `waitpid()` returns also if the selected child processes have continued (by some signal) after they were stopped
- *`test-fork-waitpid.c`*