# Advanced System Programming

Advanced System Programming

# Agenda

- Qemu Raspberry pi Setup
- Basic Debugging
- Process Management
- 

# Qemu RPI

- ❑ Use this link for reference

https://github.com/dhruvvyas90/qemu-rpi-kernel

- ❑ Download img file from

http://debian.rutgers.edu/raspbian_images/raspberrypi/images/raspbian/2019-09-26-raspbian-buster/

# wget http://debian.rutgers.edu/raspbian_images/raspberrypi/images/raspbian/2019-09-26-raspbian-buster/2019-09-26-raspbian-buster-lite.zip

- ❑ Download dtb

# wget versatile-pb-buster.dtb

# Qemu RPI

- Download Kernel

# wget kernel-qemu-4.19.50-buster

- Run Kernel using qemu using command

```
# qemu-system-arm -M versatilepb -cpu arm1176 -m 256 -hda 2019-09-26-raspbian-buster-lite.img -net user,hostfwd=tcp::5022-:22 -dtb versatile-pb-buster.dtb -kernel kernel-qemu-4.19.50-buster -append 'root=/dev/sda2 panic=1' -no-reboot
```

# Debugging – Address Sanitizer

AddressSanitizer (ASan) is an instrumentation tool created by Google security researchers to identify memory access problems in C and C++ programs.

When the source code of a C/C++ application is compiled with AddressSanitizer enabled, the program will be instrumented at runtime to identify and report memory access errors.

# Debugging – Address Sanitizer

**Memory access errors and AddressSanitizer**

C and C++ are very insecure and error-prone languages. And one of the main sources of problems is memory access errors.

Different kind of bugs in the source code could trigger a memory access error, including:

- **Buffer overflow or buffer overrun** occurs when a program overruns a buffer's boundary and overwrites adjacent memory locations.
- **Stack overflow** is when a program crosses the boundary of function's stack.
- **Heap overflow** is when a program overruns a buffer allocated in the heap.
- **Memory leak** is when a program allocates memory but does not deallocate.
- **Use after free (dangling pointer)** is when a program uses memory regions already deallocated.

# Debugging – Address Sanitizer

- **Uninitialized variable** is when a program reads a memory location before it is initialized.

All these errors are due to programming bugs. They could prevent the application from executing, cause invalid results or expose a vulnerability that could be exploited by a malicious actor. They are usually very hard to reproduce, debug and fix.

- **Heap overflow** is when a program overruns a buffer allocated in the heap.
- **Memory leak** is when a program allocates memory but does not deallocate.
- **Use after free (dangling pointer)** is when a program uses memory regions already deallocated.
- **Uninitialized variable** is when a program reads a memory location before it is initialized.

All these errors are due to programming bugs. They could prevent the application from executing, cause invalid results or expose a vulnerability that could be exploited by a malicious actor. They are usually very hard to reproduce, debug and fix.

# Debugging – Address Sanitizer

**-fsanitize=address**

# Debugging – Valgrind

Valgrind is a multipurpose code profiling and memory debugging tool for Linux when on the x86 and, as of version 3, AMD64, architectures. It allows you to run your program in Valgrind's own environment that monitors memory usage such as calls to malloc and free (or new and delete in C++). If you use uninitialized memory, write off the end of an array, or forget to free a pointer, Valgrind can detect it. Since these are particularly common problems, this tutorial will focus mainly on using Valgrind to find these types of simple memory problems, though Valgrind is a tool that can do a lot more.

**# valgrind --tool=memcheck --leak-check=yes ./val1**

# Debugging – Valgrind

**Callgrind**

valgrind Callgrind is a program that can profile your code and report on its resources usage. It is another tool provided by Valgrind, which also helps detect memory issues.

**# valgrind --tool=callgrind program-to-run program-arguments**

**# callgrind_annotate --auto=yes callgrind.out.pid**

# Debugging – Valgrind

**massif**

Massif is a heap profiler. It measures how much heap memory your program uses. This includes both the useful space, and the extra bytes allocated for book-keeping and alignment purposes. It can also measure the size of your program's stack(s), although it does not do so by default.

# Debugging – Valgrind

**massif**

Heap profiling can help you reduce the amount of memory your program uses. On modern machines with virtual memory, this provides the following benefits:

- It can speed up your program -- a smaller program will interact better with your machine's caches and avoid paging.
- If your program uses lots of memory, it will reduce the chance that it exhausts your machine's swap space.

**# valgrind --tool=massif --time-unit=B prog**

**# ms_print massif.out.pid**

# Creating Processes in Linux

- The simple way: using *system* function.

- The flexible, secure, complex way: using *fork* and *exec*

- By using *system* you can create a subprocess running the standard Bourne shell (/bin/sh) and execute a command in it.

- By using *fork* function you can create a child process which is an exact copy of it's parent.

- By using *exec* family of functions, you can replace the current process image with a new one.

DOS and Windows API use *spawn* family instead of *fork* & *exec*

# Process Management – System()

## Creating Processes in Linux

- The *system* function, uses a shell to invoke the desired program.

- It has the same features, limitations, and security flaws of the system's shell.

```
int system (const char * command )
```

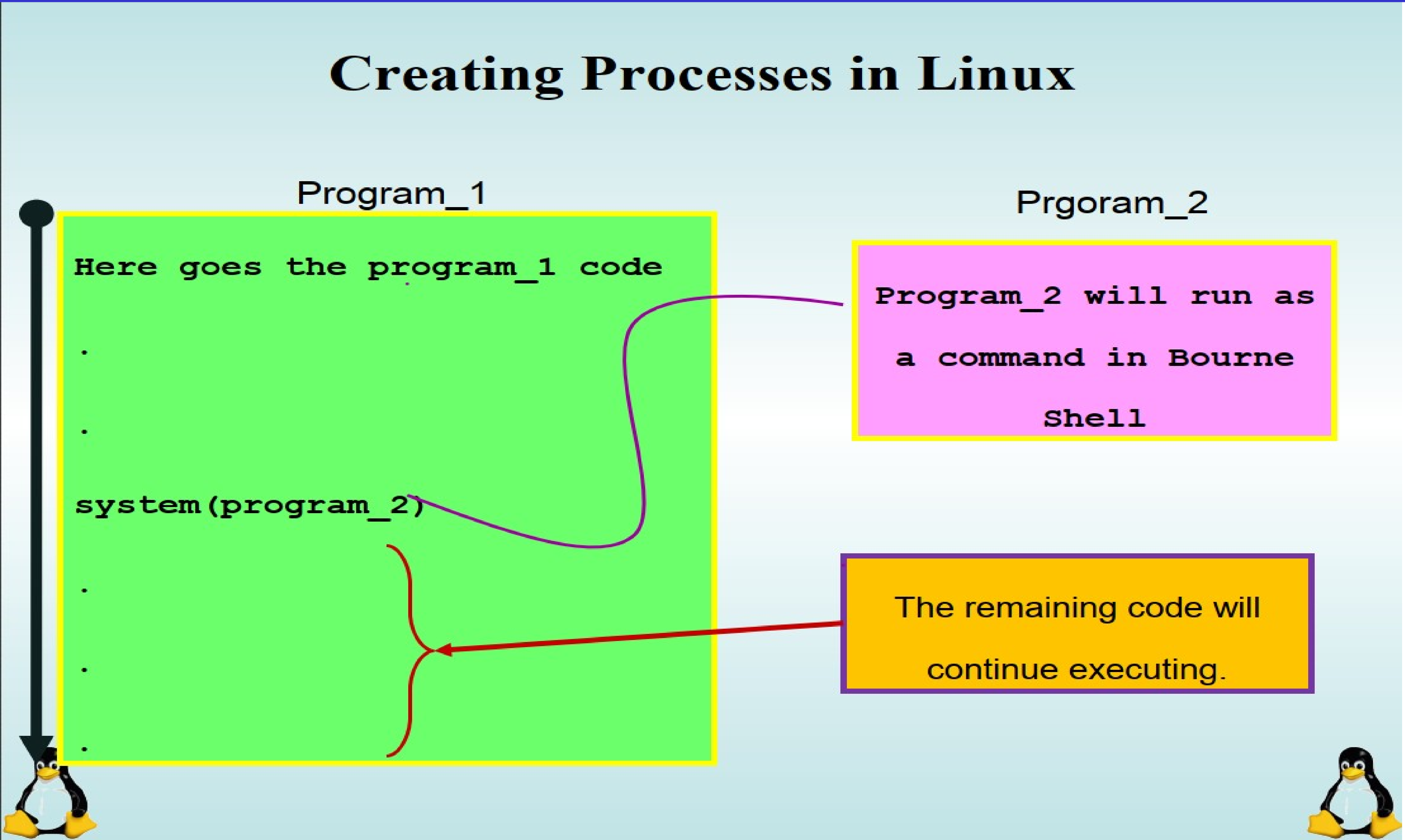
*system* will return the exit status of the command (see *wait*).

*127: shell can not be run*

*-1: any other errors*

*system* will call this command by calling *"/bin/sh -c command"*

# Process Management – System()



# Process Management – fork()

## Creating Processes in Linux

- The *fork* function creates child process which only differ in its PID with his parent.

- Return value in parent process is PID of child and in child is 0.

```
pid_t fork ( void )
```

*fork* will return a PID (PID of child or zero) on success and -1 on failure.

*fork* does not need any additional arguments. It just creates the same process as the parent.

# Process Management – fork()

## fork()

- A process calling `fork()` spawns a child process.
- The child is almost an identical *clone* of the parent:
  - Program Text (segment .text)
  - Stack (ss)
  - PCB (eg. registers)
  - Data (segment .data)

```
#include <sys/types.h>
#include <unistd.h>

pid_t fork(void);
```

# Process Management – fork()

**Creating a Process – fork()**

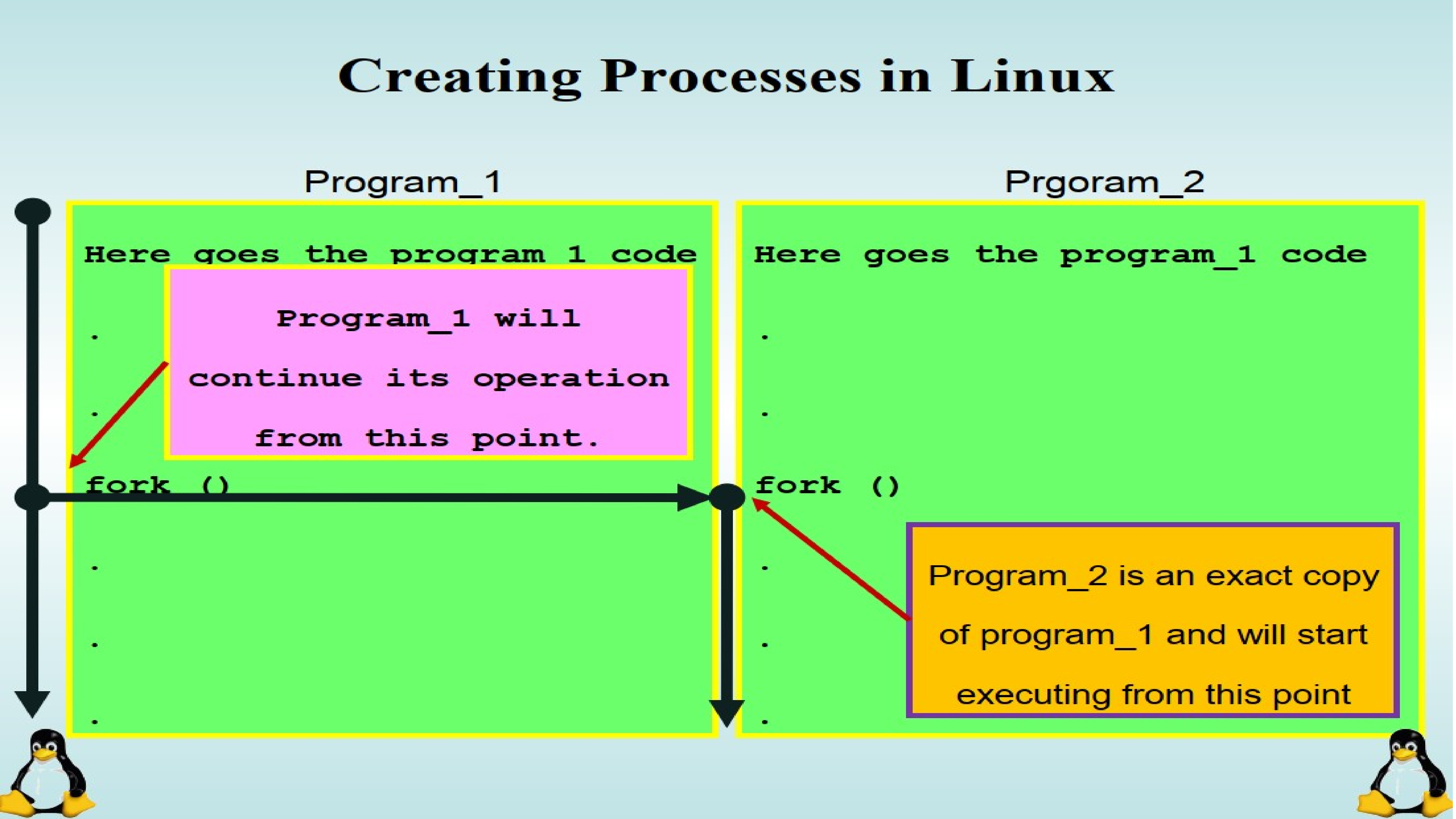
pid_t fork() creates a duplicate of the calling process:

- Both processes continue with *return from fork()*
- Child gets exact copy of code, stack, file descriptors, heap, globals, and program counter

fork() returns

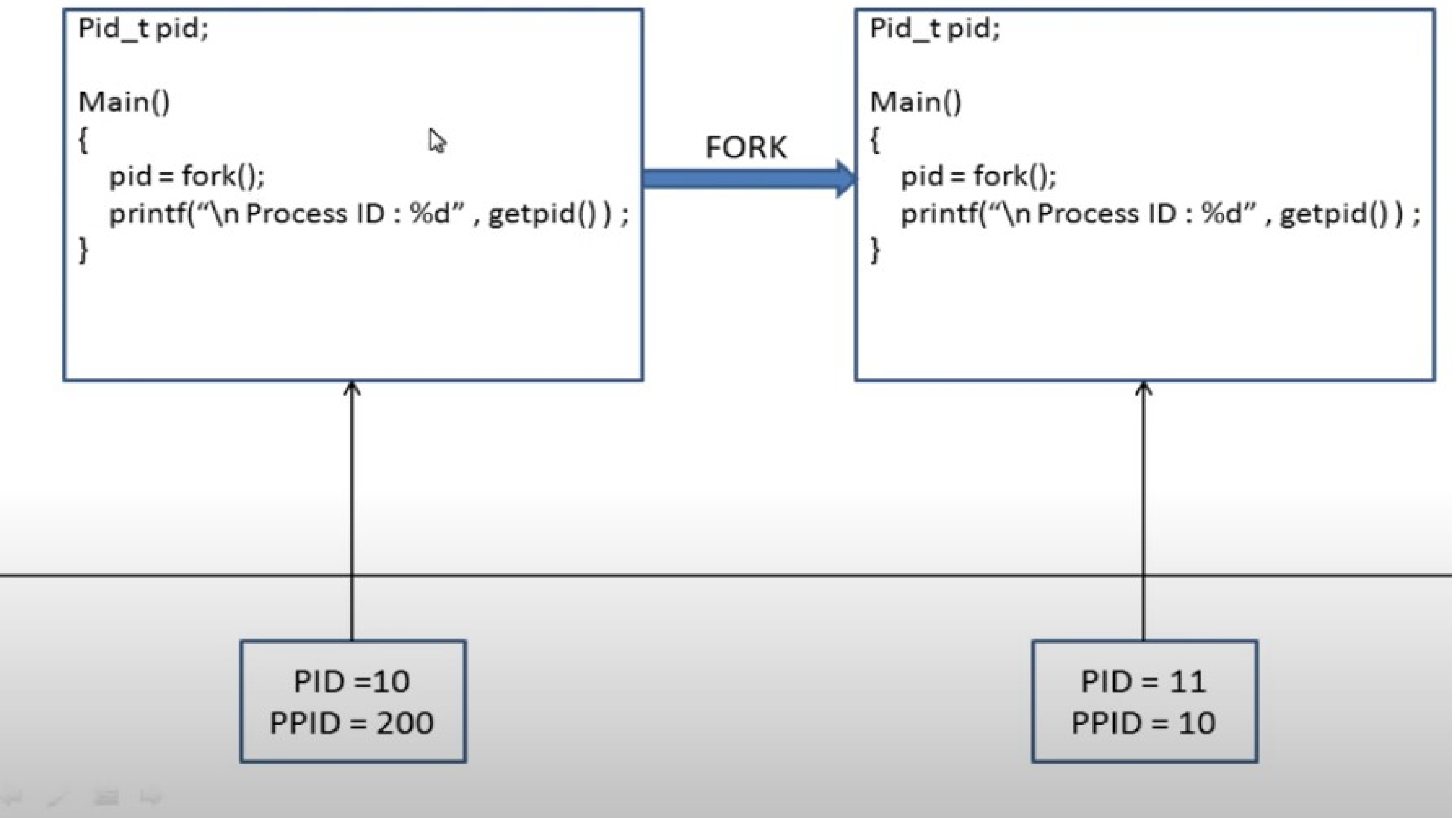
- -1 if fork fails
- 0 in child process
- child's PID in parent process

# Process Management – fork()



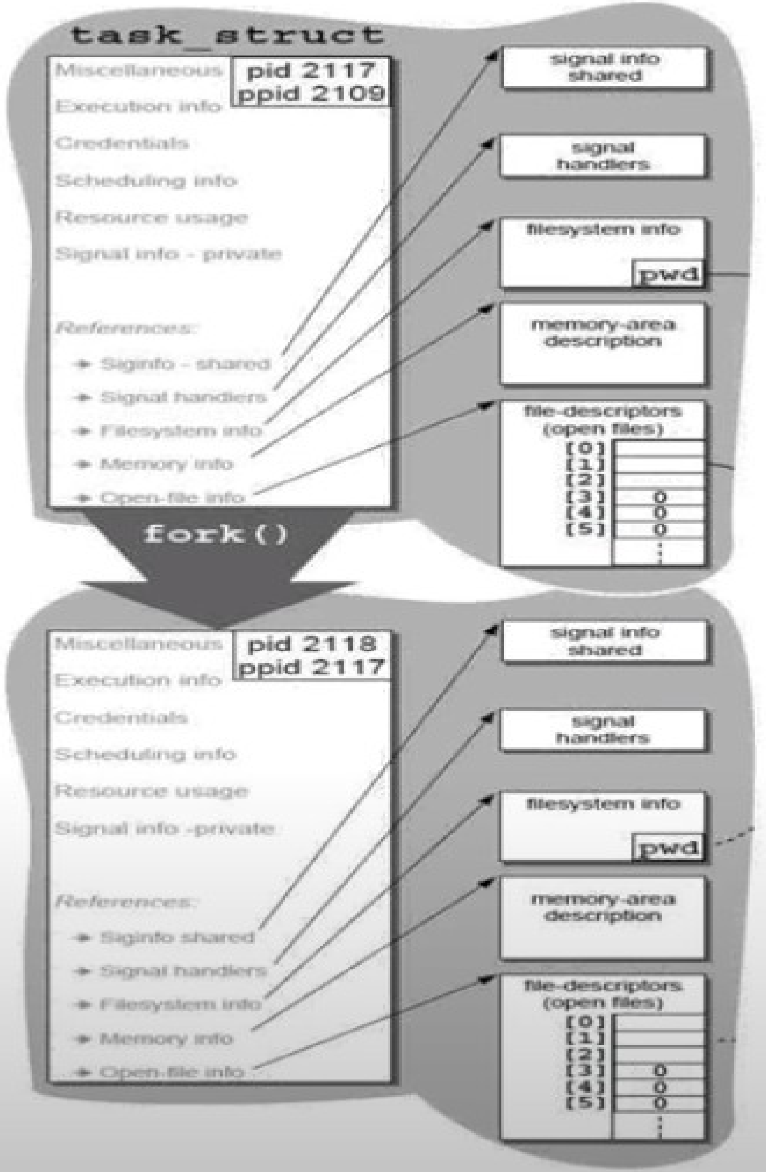
# Process Management – fork()

## Process creation with fork()



# Process Management – fork()

## Process creation with fork()




fork → sys_fork → do_fork ( $SRC_HOME/kernel/fork.c )

- ✓ fail if #processes for this user exceeded
- ✓ increment #processes for this user
- ✓ fail if #processes in system exceeds maximum
  ***nr_threads >= max_threads***
- ✓ Allocate new ***task_struct*** instance
- ✓ determine pid for child
- ✓ clear pending signals for child
- ✓ Copy the satellite structures for the child process.
  - copy ***files_struct***
  - copy ***fs_struct***
  - copy ***signal_struct***
  - copy ***sighand_struct***
  - copy ***mm_struct***
- ✓ copy ***cpu-registers*** from parent to child; fill **eip** of child with other continuation address
- ✓ set child's state to '***running***', add to runqueue and force process-switch to child
- ✓ Return PID of child to parent

# Process Management – exec()

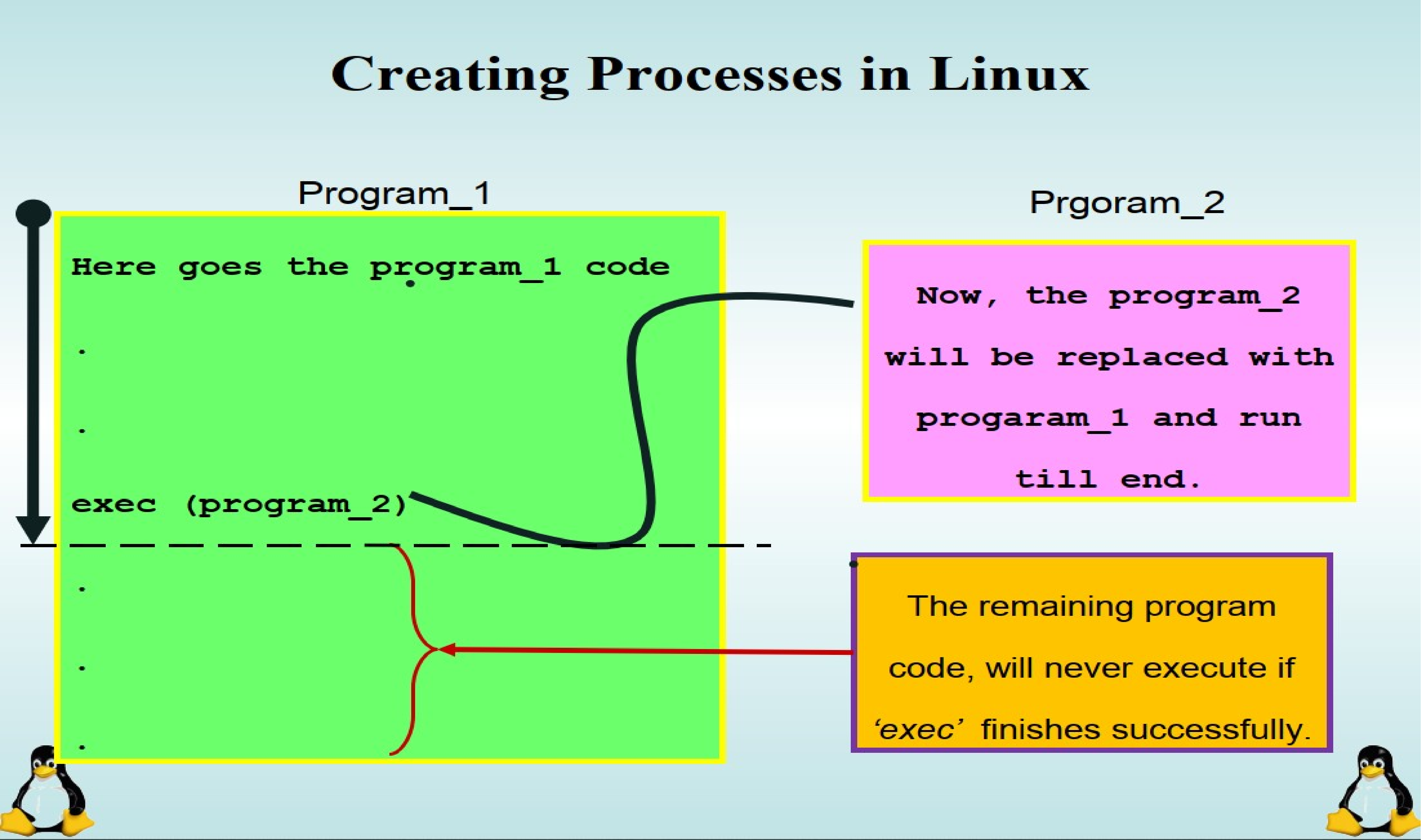
## Creating Processes in Linux

- The *exec* family, vary slightly in their capabilities and the way of calling:

  - Functions containing the letter ‘*p*’ in their name, accept a program name and search it in current execution PATH.
  - Those who contain the letter ‘*v*’ or ‘*l*’ in their name, accept the argument list as an array or list for the new program.
  - Those who contain the letter ‘*e*’ in their name, accept an array of environment variables.

*exec* replaces the calling process with another one, so it will never return a value on success but on failure, returns -1.

# Process Management – exec()



# Process Management – exec()

## Creating Processes in Linux

- All of the *exec* family of functions, use just one system call: *execve()*

- execl() functions are variadic functions.

- When calling exec, remember that almost all Linux applications, use argv[0] as their binary image name.

- When using *exec* family, the new process does not have the previous' signal handlers and other stuff.

- The new process has the same values for its PID, PPID, priority and permissions.

# Process Management – vfork()

## vfork

- Creates a new processes with the express purpose of exec-ing a new program
- New child process runs in parent's address space until exec or exit is called
- vfork guarantees that the child will run before the parent until exec or exit call is reached

# Process Management – vfork()

## Differences between fork() and vfork()

| | fork() | vfork() |
|---|---|---|
| **Address space** | Both the child and parent process will have different address space | Both child and parent process share the same address space |
| **Modification in address space** | Any modification done by the child in its address space is not visible to parent process as both will have separate copies | Any modification by child process is visible to both parent and child as both will have same copies |
| **CoW(copy on write)** | This uses copy-on-write. | Vfork doesn't use CoW |
| **Execution summary** | Both parent and child executes simultaneously | Parent process will be suspended until child execution is completed. |
| **Outcome of usage** | Behaviour is predictable | Behaviour is not predictable |

# Process Management – clone()

- **Clone process** is created using primitive “clone,” by duplicating its parent process
  - Allows both processes to share same segment of code and data
  - Modification of one is visible to other, which is unlike classical processes
- Ability to clone processes brings possibility of implementing servers in which several threads may be executing

# Process Management – getpid() and getppid()

**Process Identification**

UNIX identifies processes via a unique Process ID

Each process also knows its parent process ID since each process is created from a parent process.

Root process is the ‘init’ process

‘*getpid*’ and ‘*getppid*’ – functions to return process ID (PID) and parent process ID (PPID)

**Example 1**

```
#include <stdio.h>
#include <unistd.h>
int main (void) {
    printf("I am  process %ld\n", (long)getpid());
    printf("My parent id is %ld\n", (long)getppid());
    return 0;
}
```

# Process Management – exit()

**Process Termination**

Normal exit (voluntary)

- Returning zero from main()
- exit(0)

Error exit (voluntary)

- exit(1)

Fatal error (involuntary)

- Divide by 0, seg fault, exceeded resources

Killed by another process (involuntary)

- Signal: kill(procID)

# wait(), waitpid() System Calls

pid_t wait(int *status);

*wait()* causes parent process to wait (block) until *some* child finishes

*wait()* returns child's pid and exit status to parent

*waitpid()* waits for a specific child

| *errno* | Cause |
|---|---|
| ECHILD | Caller has no unwaited-for children |
| EINTR | Function was interrupted by signal |
| EINVAL | Options parameter of waitpid was invalid |

# Process Management – wait()

**pid_t wait(int *status);**

In the case of a terminated child, performing a wait allows the system to release the resources associated with the child; if a wait is not performed, then the terminated child remains in a "zombie“ state.

If a child has already terminated, then the call returns immediately. Otherwise it blocks until either a child terminates or a signal handler interrupts the call. A child that has terminated and which has not yet been waited upon by this system call (or waitpid) is termed waitable.

# Process Management – wait()

If status is not NULL, wait() stores status information in the int to which it points. This integer can be inspected with specific macros (see man pages):

**WIFEXITED(status)**

returns true if the child terminated normally, that is, by calling exit, or by returning from main().

**WEXITSTATUS(status)**

returns the exit status of the child. This consists of the least significant 8 bits of the status argument that the child specified in a call to exit or as the argument for a return statement in main(). This macro should only be employed if WIFEXITED returned true.

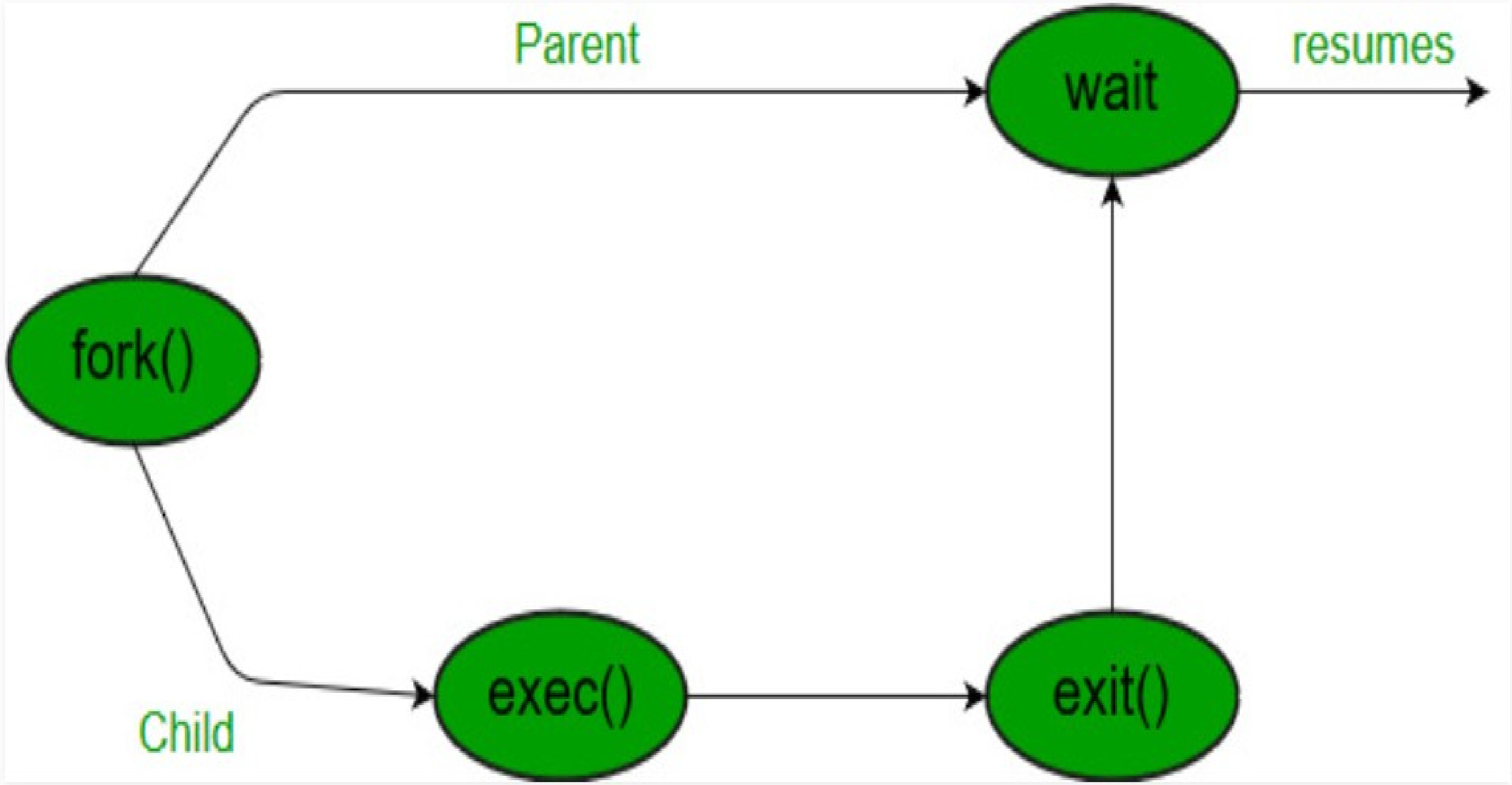
# Process Management – wait()

A child that terminates, but has not been waited for becomes a "zombie". The kernel maintains a minimal set of information about the zombie process (PID, termination status, resource usage information) in order to allow the parent to later perform a wait to obtain information about the child.

As long as a zombie is not removed from the system via a wait, it will consume a slot in the kernel process table, and if this table fills, it will not be possible to create further processes.

If a parent process terminates, then its "zombie" children (if any) are adopted by init(8), which automatically performs a wait to remove the zombies.

# Process Management – wait()



# Thank you