CSCI 2410 Systems Programming

Fall 2026

Instructor: Jie Zhou

Department of Computer Science

George Washington University



Slides materials are mostly credited to Gabriel Parmer and Sibin Mohan at GWU.

Process

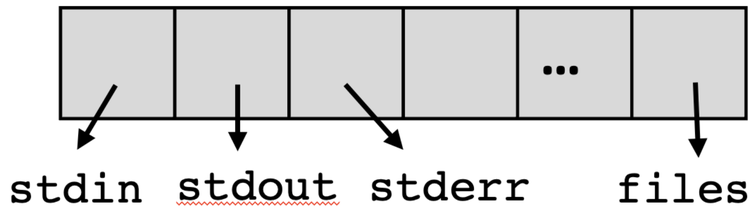
- A running instance of a program, together with the state and resources needed for its execution.
  - has its own virtual address space
  - has execution state such as registers and program counter
  - owns/uses resources such as open files
- Each process has an ID: pid
- Also has an owner
  - Usually is the user who starts the process, but can change.
- Current working directory: `pwd`
- Almost every process has a parent process.
  - PID 1 is the special root of the process tree.

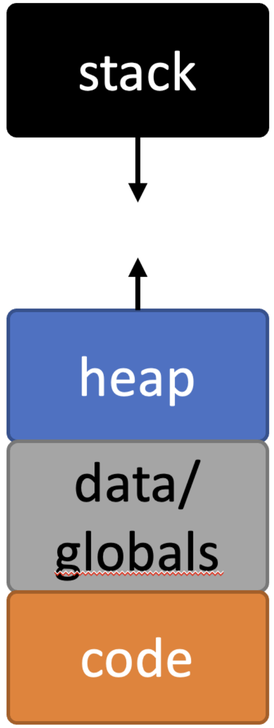
Process

pid
ppid
user: sibin
cwd: /home/sibin

command arguments
environment variables

descriptors

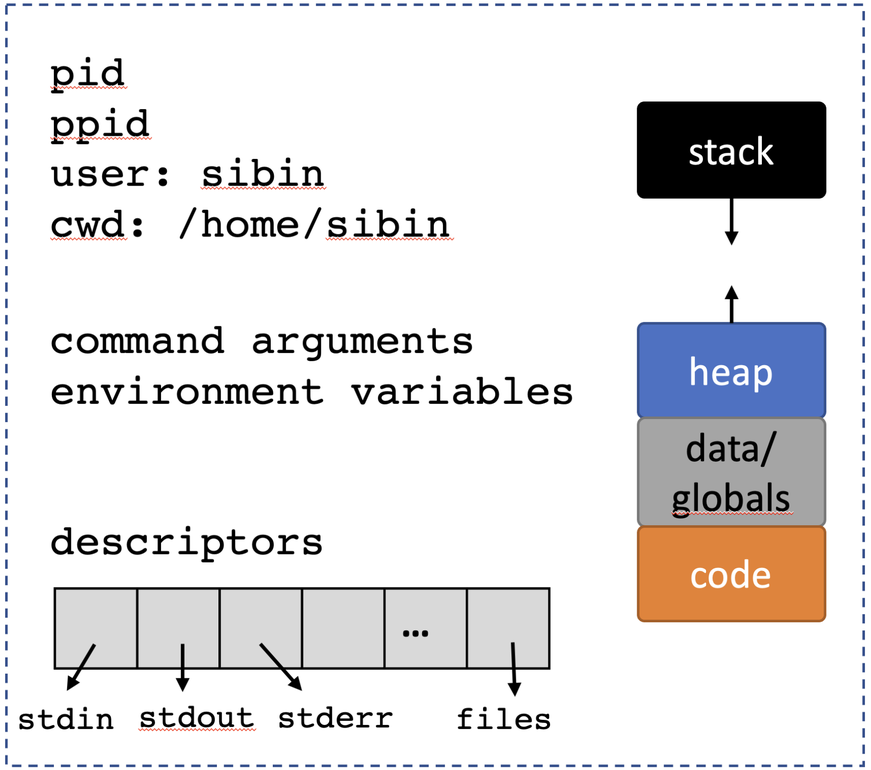


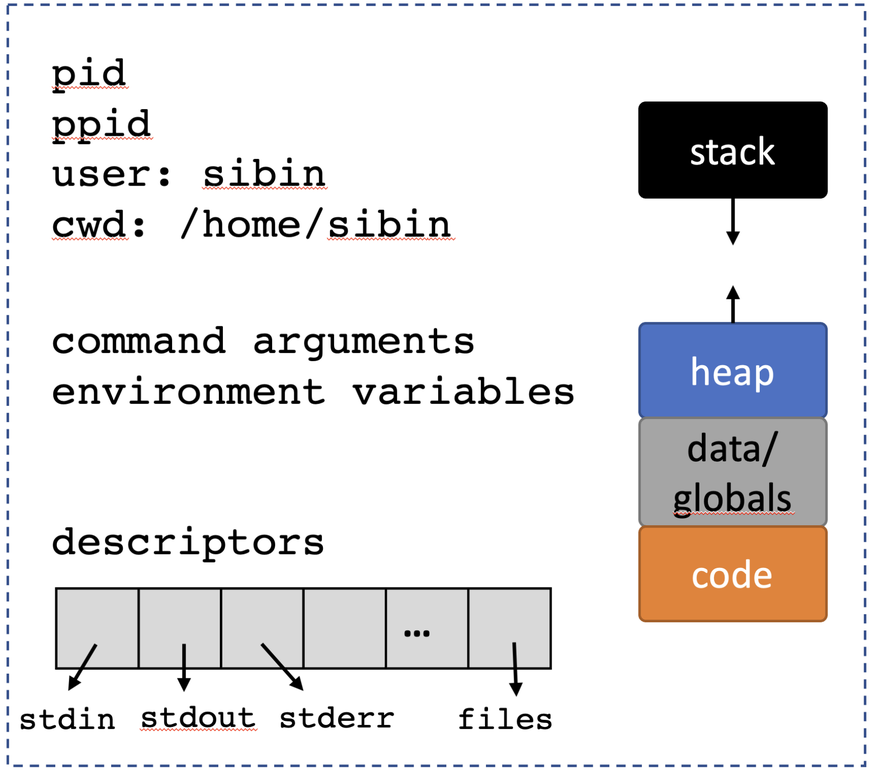


`fork()`

- Create a new, child process from the calling process.
- Defined in `<unistd.h>` (from libc)

After fork()


pid
ppid
user: sibin
cwd: /home/sibin
command arguments
environment variables
descriptors
stdin
stdout
stderr
files
stack
heap
data/
globals
code


pid
ppid
user: sibin
cwd: /home/sibin
command arguments
environment variables
descriptors
stdin
stdout
stderr
files
stack
heap
data/
globals
code

fork()

```
pid_t fork(void);
```

- Include from <unistd.h>
- `pid_t`: process identifier defined in `<sys/types.h>`.
  - A signed integer, usually by `typedef int`

Return value	Returned in	meaning
child's PID (>0)	parent	child successfully created
0	child	inside child
-1	parent	`fork()` failed; no child created

wait()

```
pid_t wait(int *wstatus);
pid_t waitpid(pid_t pid, int *wstatus, int options);
```

- Called by a parent, wait for state change in a child process
- Parent can wait for
  - any child; e.g., `wait(null)` or `waitpid (-1, ….)`
  - a specific child
  - a subset of child
- A child changes state if
  - child terminates
  - stopped by a signal
  - resumed by a signal

`wait()` return value

Return value	Meaning
pid via pid_t	pid of terminated child
-1	error

wait() Example

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

int main(void) {
    pid_t pid = fork();

    if (pid == 0) {
        // Child process
        // Do whatever it does;
    } else if (pid > 0) {
        // Parent
        int status;
        wait(&status);

        if (WIFEXITED(status)) {
            printf("Child exited with code %d\n",
                   WEXITSTATUS(status));
        }
    }
    return 0;
}
```

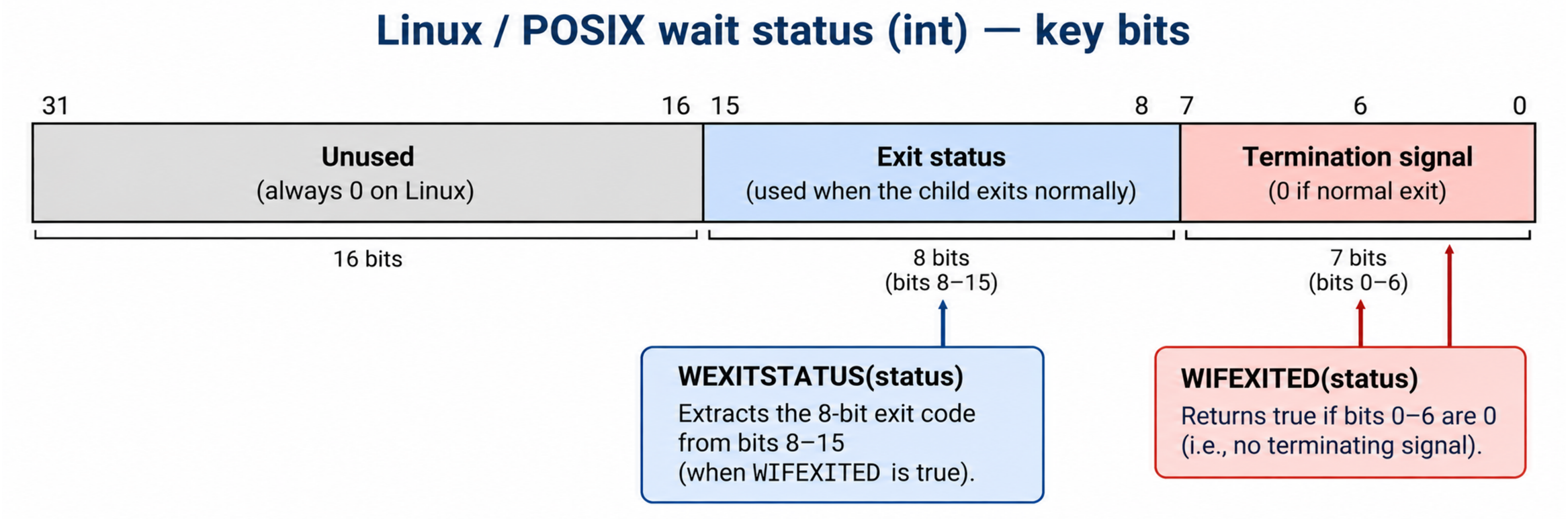
exit()

- Used to does some cleanup work and really terminate current process
- Include from `<stdlib.h>`

```
void exit(int status);
```

- `status`: encode exit status, e.g., success/failure, for its parent process
- Only the lower 8 bits are used, encoding a value in 0–255
- Parent can extract a child's exit status.
  - `WIFEXITED(status)`: returns 0 if child terminated normally
  - `WEXITSTATUS(status)`: extract child's exit status value/code

Status Received by `wait()`



Two Standard C Exit Status

Macro	Meaning	Typical value
EXIT_SUCCESS	Successful termination	0
EXIT_FAILURE	Unsuccessful termination	1

```c
#include <stdlib.h>

int main(void) {
    if (something_wrong)
        exit(EXIT_FAILURE);

    exit(EXIT_SUCCESS);
}
```

wait and exit

Parent

```
int status ;
pid_t pid ;

pid = fork() ;
pid = wait (&status) ;
```

wait and exit

Parent

```
int status ;
pid_t pid ;

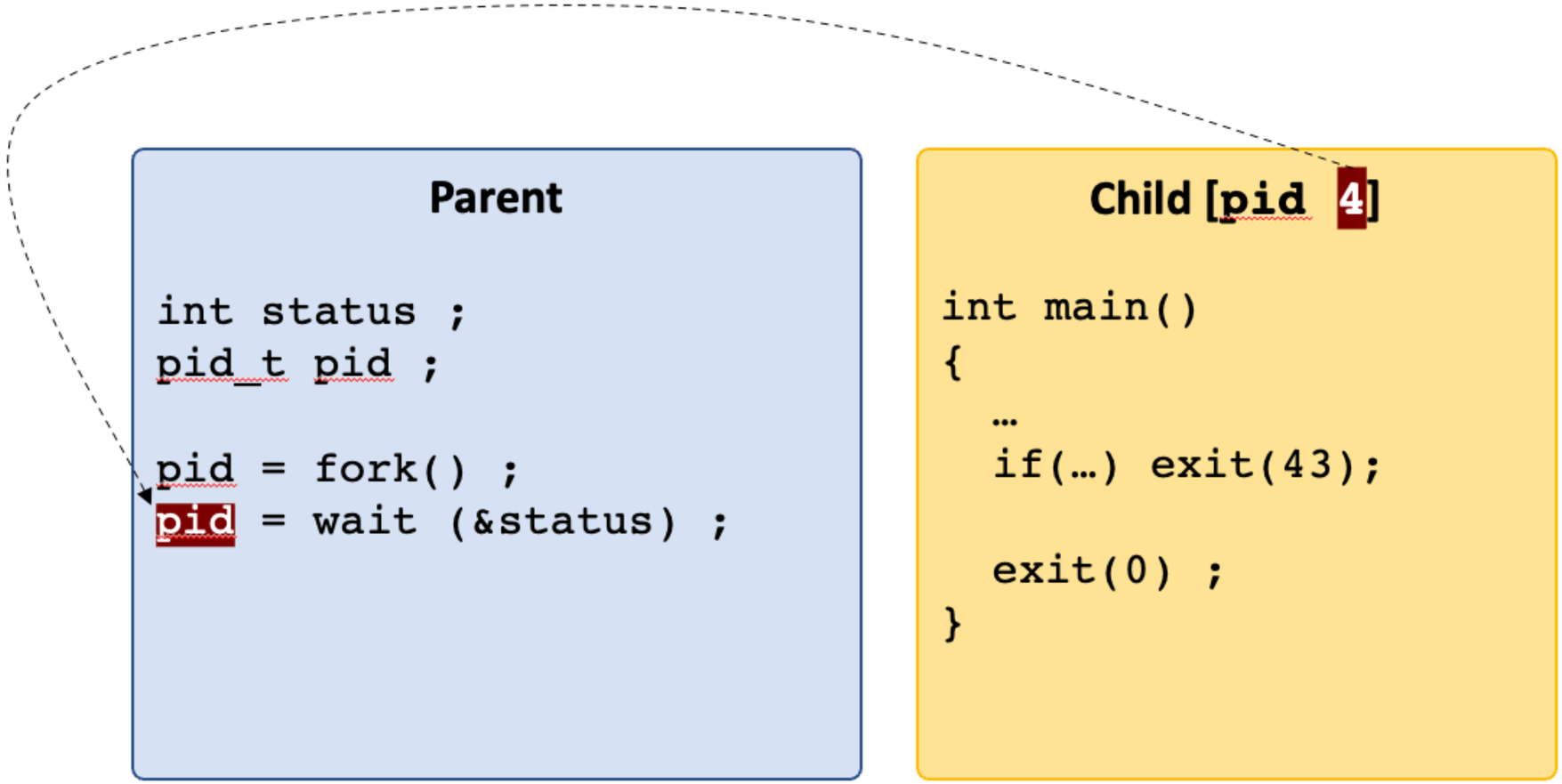
pid = fork() ;
pid = wait (&status) ;
```

Child [pid 4]

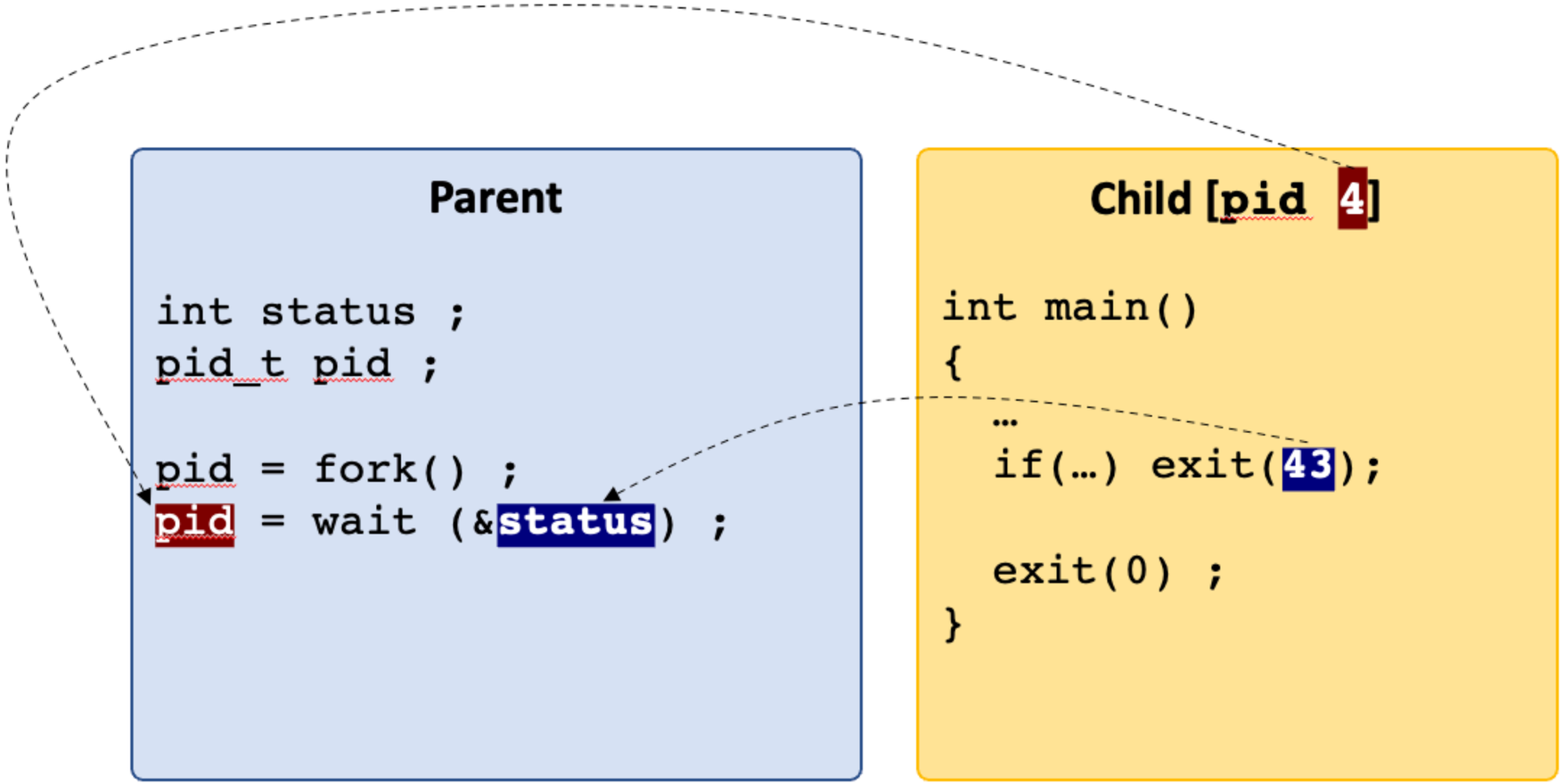
```
int main()
{
  …
  if(…) exit(43);

  exit(0) ;
}
```

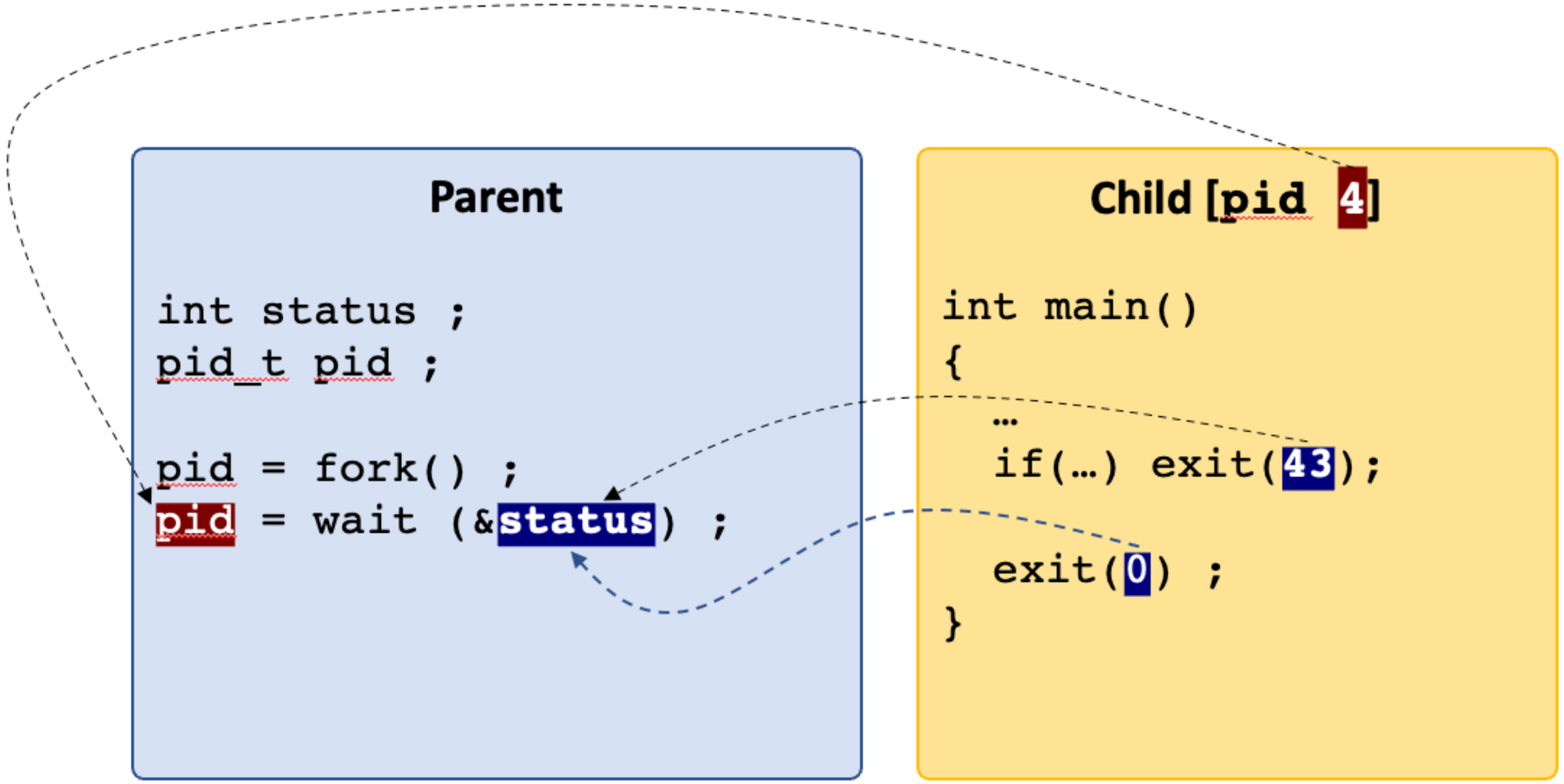
wait and exit



wait and exit



wait and exit



How is the exit code *transferred* from child to parent?

Cleanup during `exit()`

- We can register callback functions to perform cleanup before a process terminates.
- We can register more than one callbacks and they execute in reverse registration order
- Register callbacks using
  - `atexit()`, from `<stdlib.h>`
  - `on_exit()`, from `<stdlib.h>`

`atexit()`: Register a Cleanup Callback

```
typedef void (*function)(void);
int on_exit(function my_func);
```

- register a *user-defined* function pointer to call during program termination
- Takes no arguments and returns nothing
- Can register callbacks multiple times

```
#include <stdio.h>
#include <stdlib.h>

void cleanup(void) {
    printf("Cleaning up!\n");
}

int main(void) {
    atexit(cleanup);
    printf("Hello, world!\n");
    return 0;
}
```

`on_exit()`: Register a Cleanup Callback

```
typedef void (*function)(int, void *);
int on_exit(function my_func, void *arg);
```

- register a *user-defined* function pointer to call during program termination
- callback receives two arguments
  - `int status`: Exit status passed to `exit()`
  - `void *`: User-defined argument passed to `on_exit()`

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3
 4 void cleanup(int status, void *arg) {
 5     printf("Cleaning up! Exit status: %d\n", status);
 6     printf("Message: %s\n", (char *)arg);
 7 }
 8
 9 int main(void) {
10     on_exit(cleanup, "Goodbye!");
11     printf("Hello, world!\n");
12     return 0;
13 }
```

`_exit()`

- immediate termination of process!
- no return value
- output buffers are not flushed
- does not call any user-defined cleanup functions

wait() Example

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

int main(void) {
    pid_t pid = fork();

    if (pid == 0) {
        // Child process
        // Do whatever it does;
    } else if (pid > 0) {
        // Parent
        int status;
        wait(&status);

        if (WIFEXITED(status)) {
            printf("Child exited with code %d\n",
                   WEXITSTATUS(status));
        }
    }
    return 0;
}
```

Orphaned Children?

- What if parent exits *before* child?
- Who “waits” for child and gets its status?
- Grandparent?
- That would be too complicated.
- The `init` process takes over.
  - First process started during Unix/Linux boot, with PID = 1
  - Ancestor of all processes
  - In modern Linux, it is named `systemd`

Various States of A Process

Process States Summary

Running Process
- Executing code
- Using CPU and memory

Blocked Process
- Waiting for event
- e.g., wait() for child

Zombie Process
- Terminated but not reaped
- Exit status in process table

Orphan Process
- Parent terminated first
- Adopted by init (PID 1)

Properly Terminated
- Process ended cleanly
- Parent called wait()

Key: Always call wait() in parent process to prevent zombie accumulation!

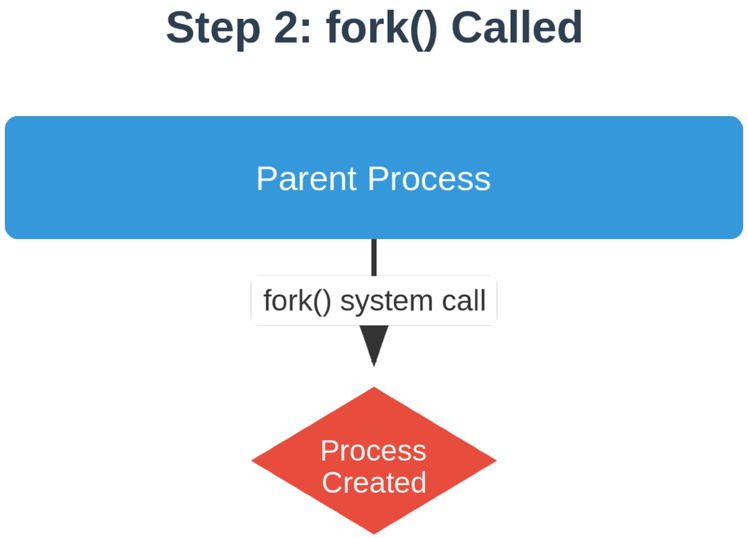
Various States of A Process

Step 1: Initial State

Parent Process Running

The parent process is executing normally before calling fork()

Various States of A Process

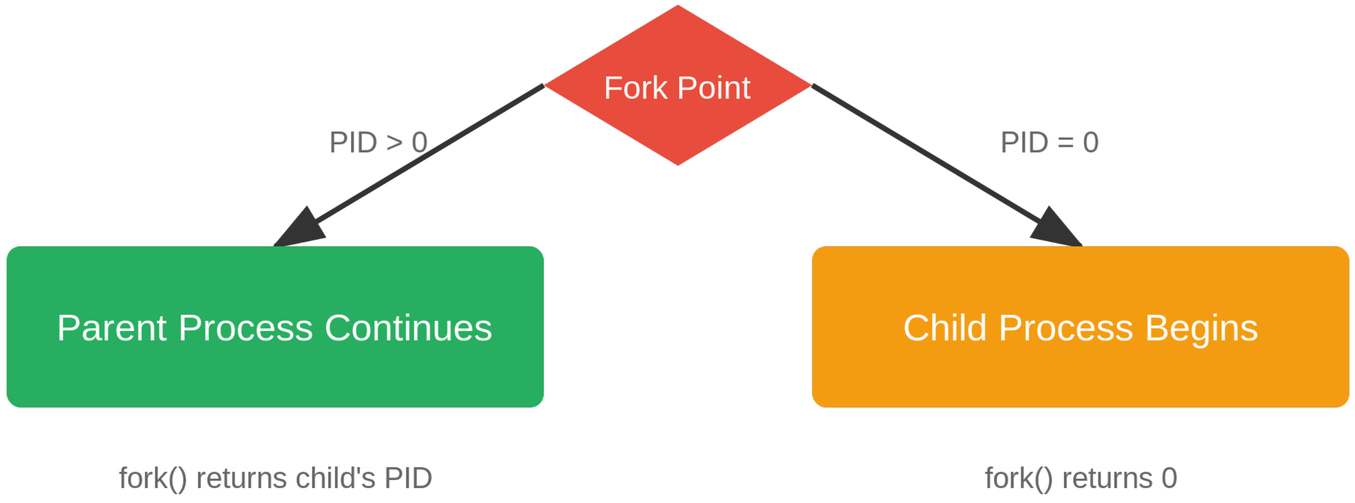


fork() creates an exact copy of the parent process

Both processes continue from this point

Various States of A Process

Step 3: Process Split

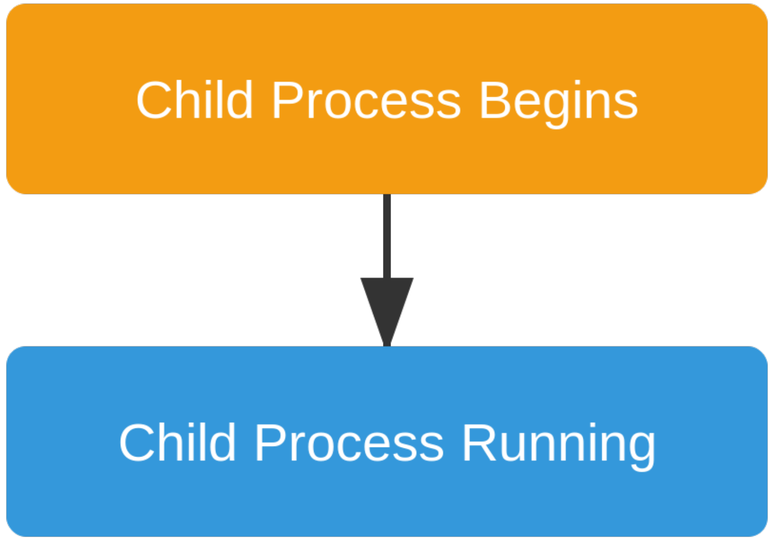


After fork(), two identical processes exist

They can be distinguished by fork()'s return value

Various States of A Process

Step 4: Child Process Running

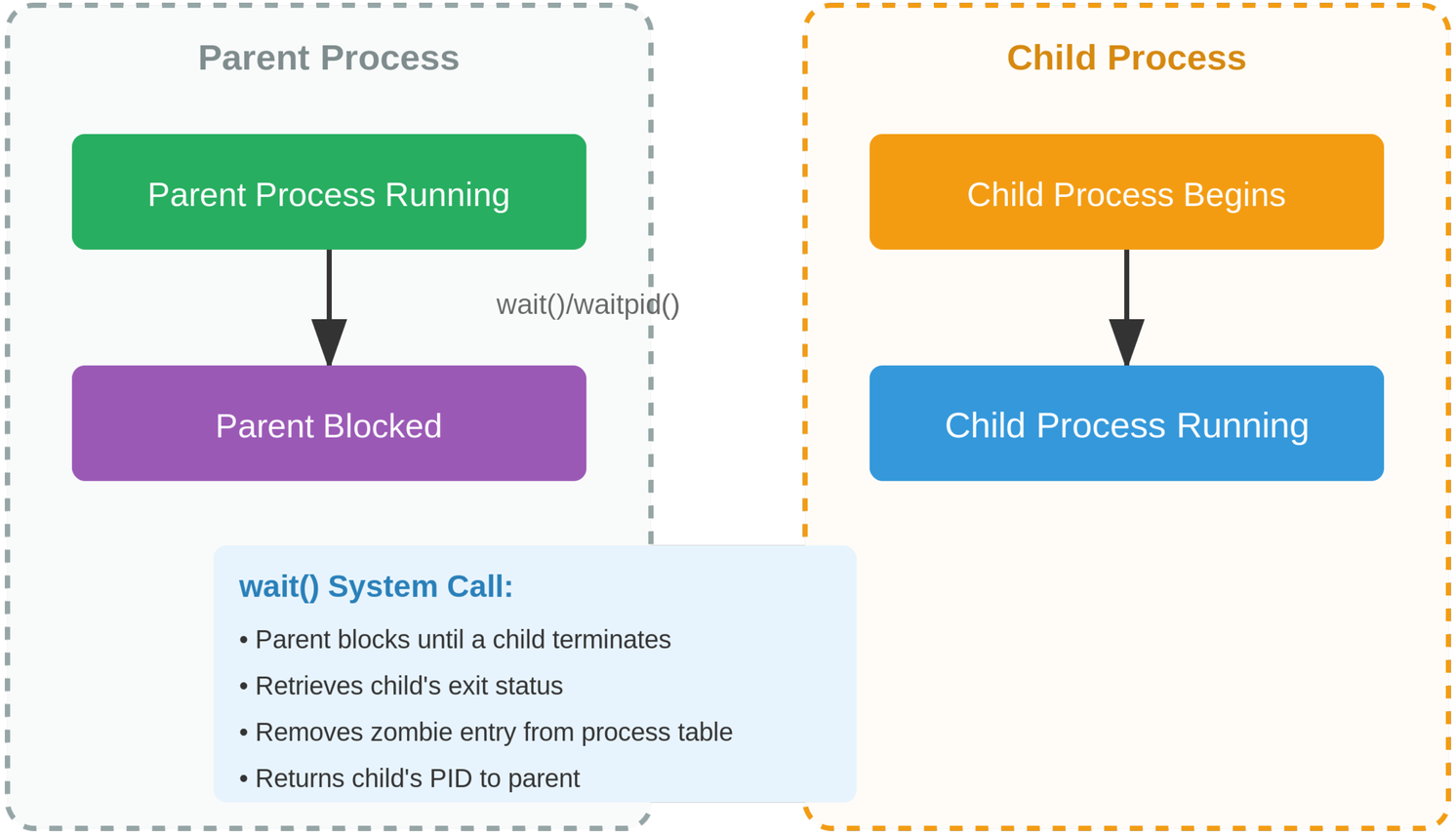


Child executes its code independently:

- Performs assigned tasks
- Has its own PID and memory space

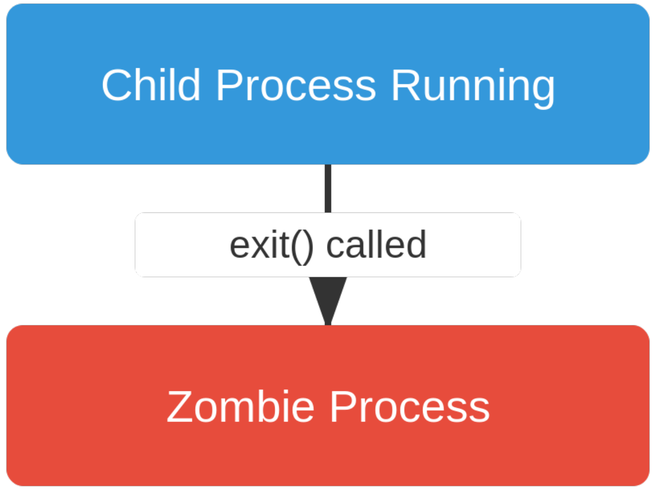
Various States of A Process

Step 4a: Parent Calls wait()



Various States of A Process

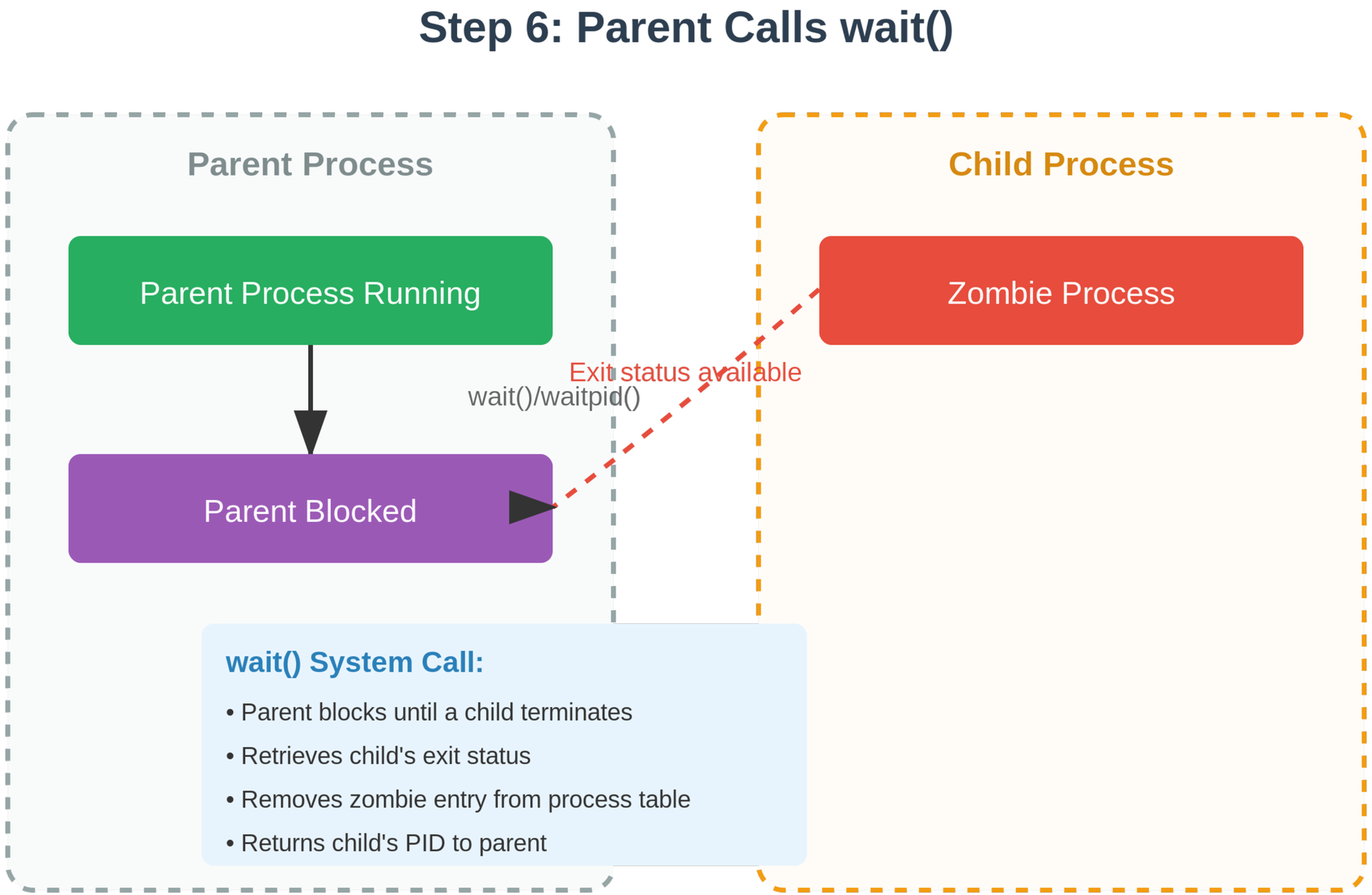
Step 5: Child Becomes Zombie



Zombie State Characteristics:

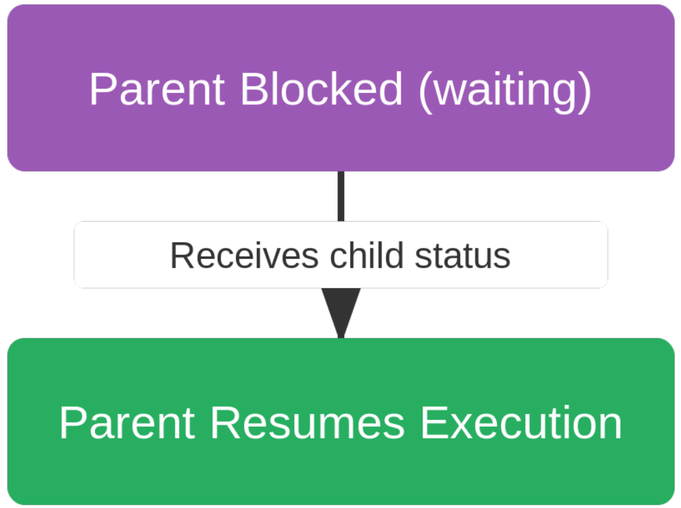
- Process has terminated
- Exit status stored in process table
- Waiting for parent to collect status
- Consumes minimal resources (only PID entry)

Various States of A Process



Various States of A Process

Step 7: Parent Resumes



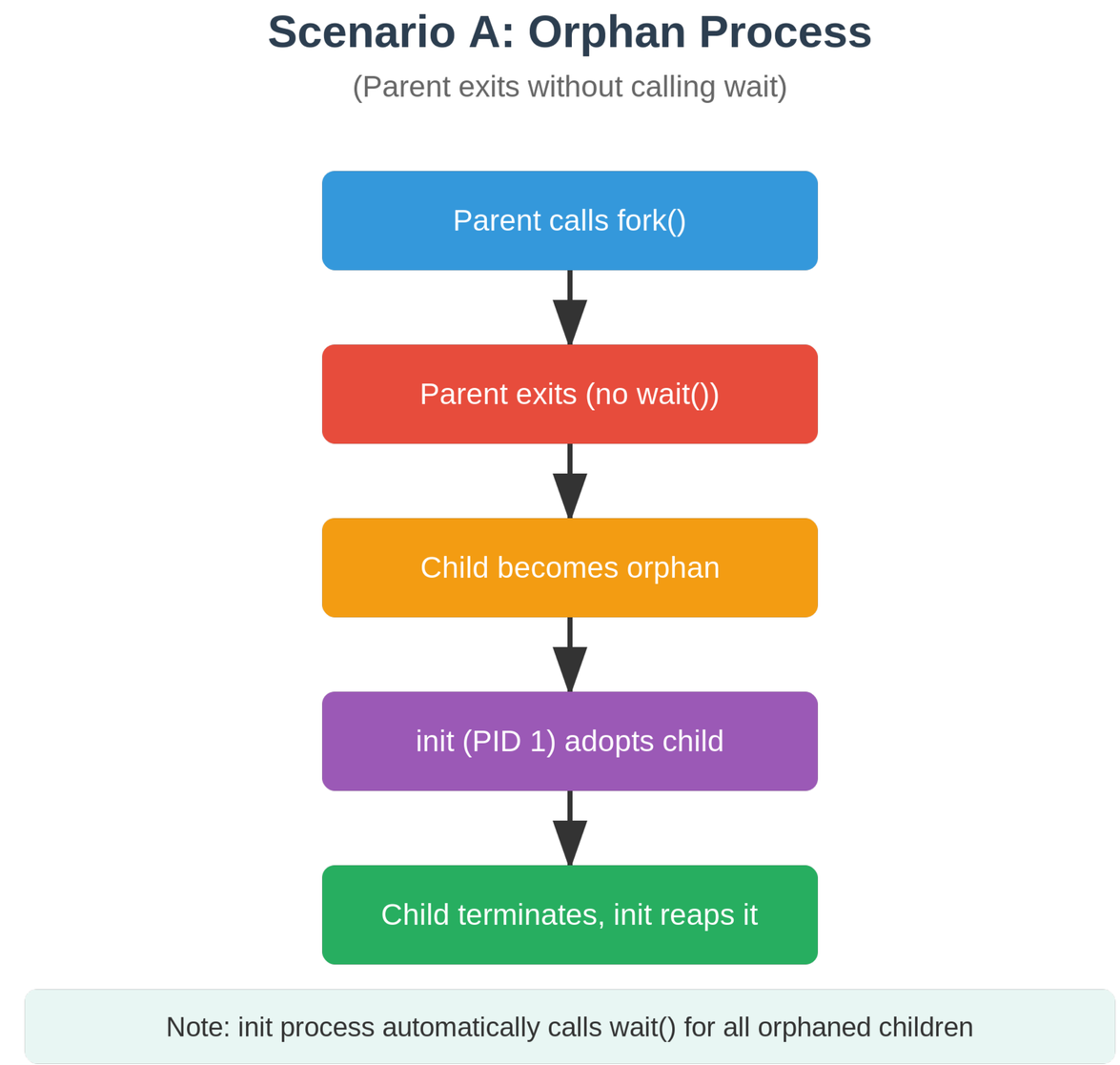
After wait() Returns:

- Parent has child's exit status
- Zombie process fully removed
- Parent can continue execution
- Child's resources fully released

Zombie Process

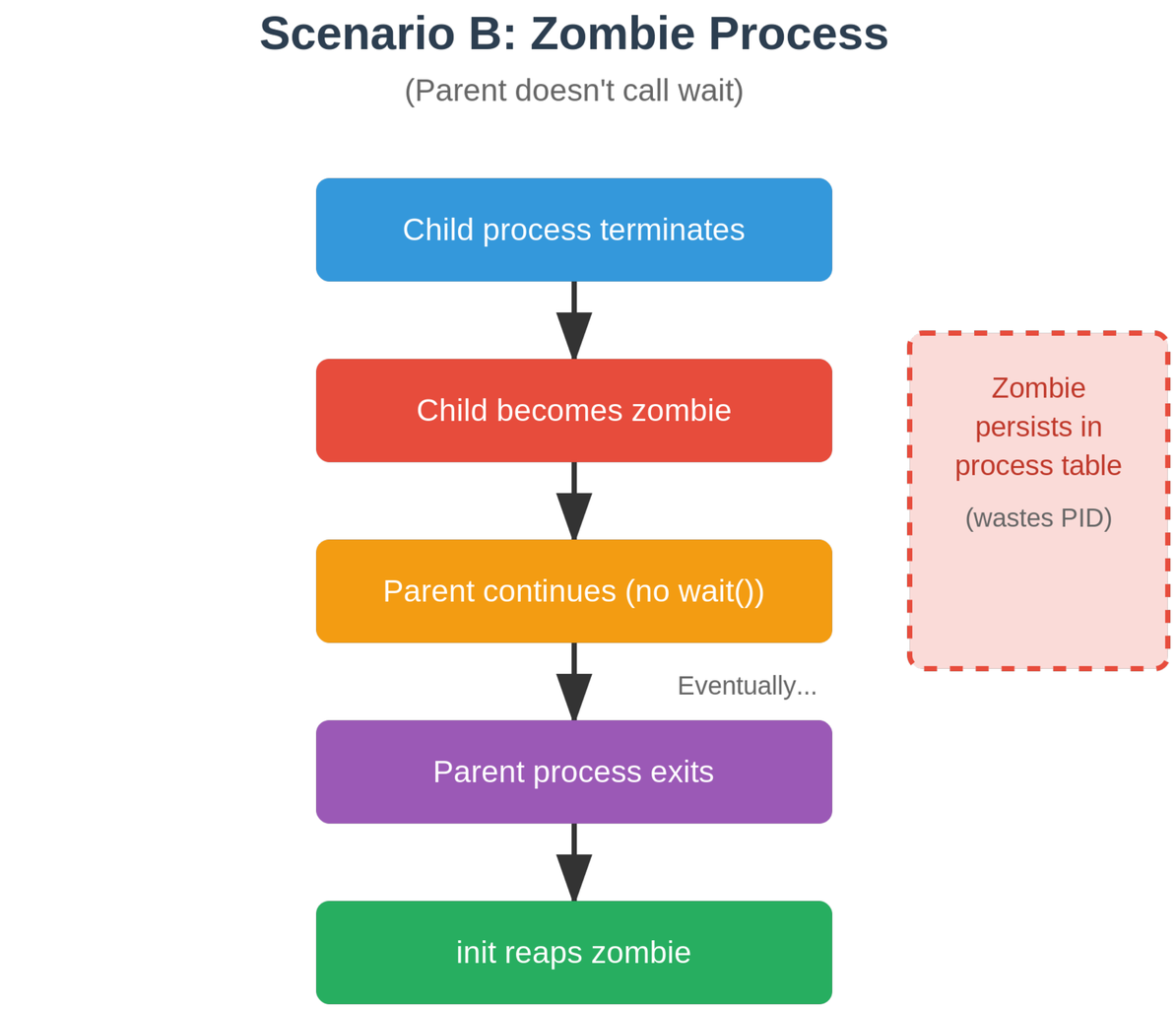


Scenario A: Orphan Process



Note: init process automatically calls wait() for all orphaned children

Scenario B: Zombie Process



Warning: Zombies consume PIDs and can exhaust system resources if not reaped!

Orphan vs Zombie

	Orphan	Zombie
Definition	Parent terminated before child.	Child terminated but parent hasn't collected its exit status.
Still executing?	Yes, potentially	No
Parent live?	Original parent is gone.	Usually yes
What happens next?	Adopted by init (PID 1) or a subreaper	Remains a zombie until cleaned up.
Can become a zombie?	Yes, when it terminates.	Already is
Can become an orphan?	Already is	Yes, when its parent terminates.

Fork(), Wait() and Process Lifecycle in Unix/Linux

