



Operations on Processes

Lecture 03 · Section 3.3

Prof. Mostafa Elhosseini

Professor of Smart Systems Engineering

What This Lecture Covers

3.3.1

Process Creation

Process tree, PIDs and the system calls

`fork()` · `exec()` · `wait()` ·
`CreateProcess()`

3.3.2

Process Termination

Normal exit, early termination, cascading

`zombies` · `orphans` · `Android`

The Life of a Process



SENTENCE TO REMEMBER

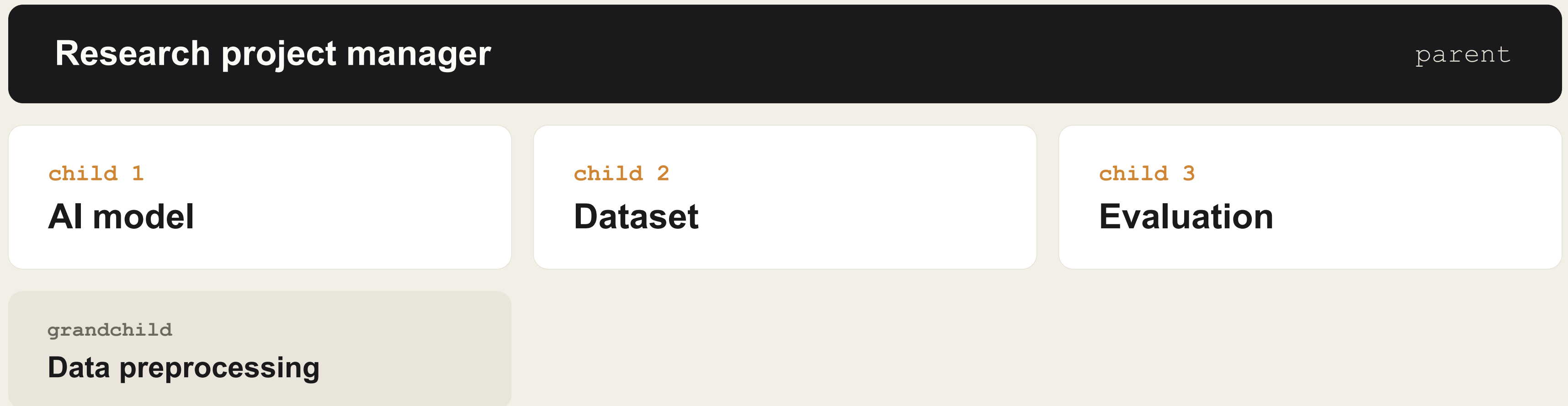
“A process is born, performs work, may create children, and eventually dies.”

3.3.1

Process Creation

A process may create new processes — and each of those may create more.

Parent and Child



The result is a **tree of processes**.

Why Create Child Processes?

Run another program

Serve network clients

Handle user logins

Start background services

Divide work into activities

```
$ python3 analysis.py
```

parent

The shell

child

The Python process

Bash Shell — Parent

Accepts user commands



fork() → exec()



Python — Child

Executes analysis.py

The Process Identifier (PID)

A unique integer assigned by the OS.

Like a student ID — names can repeat, IDs cannot.

PROCESS	PID
Browser	2410
Python	3256
Word processor	4189
Terminal shell	5201

Index . The kernel uses it to reach a process’s attributes: memory in use, parent, open files.

Passed to the parent . When a process is created its identity goes to the parent — which is how a parent can later terminate it.

Using the PID

- A PID is the identity card of a process.

```
$ ps -el
```

List processes and their PIDs

```
$ kill 3256
```

Send **SIGTERM** — a request to stop

SIGTERM: Requests termination

SIGKILL: Forces termination

NOTE

Unique among **active** processes — may be reused later.

The command **ps** stands for Process Status.

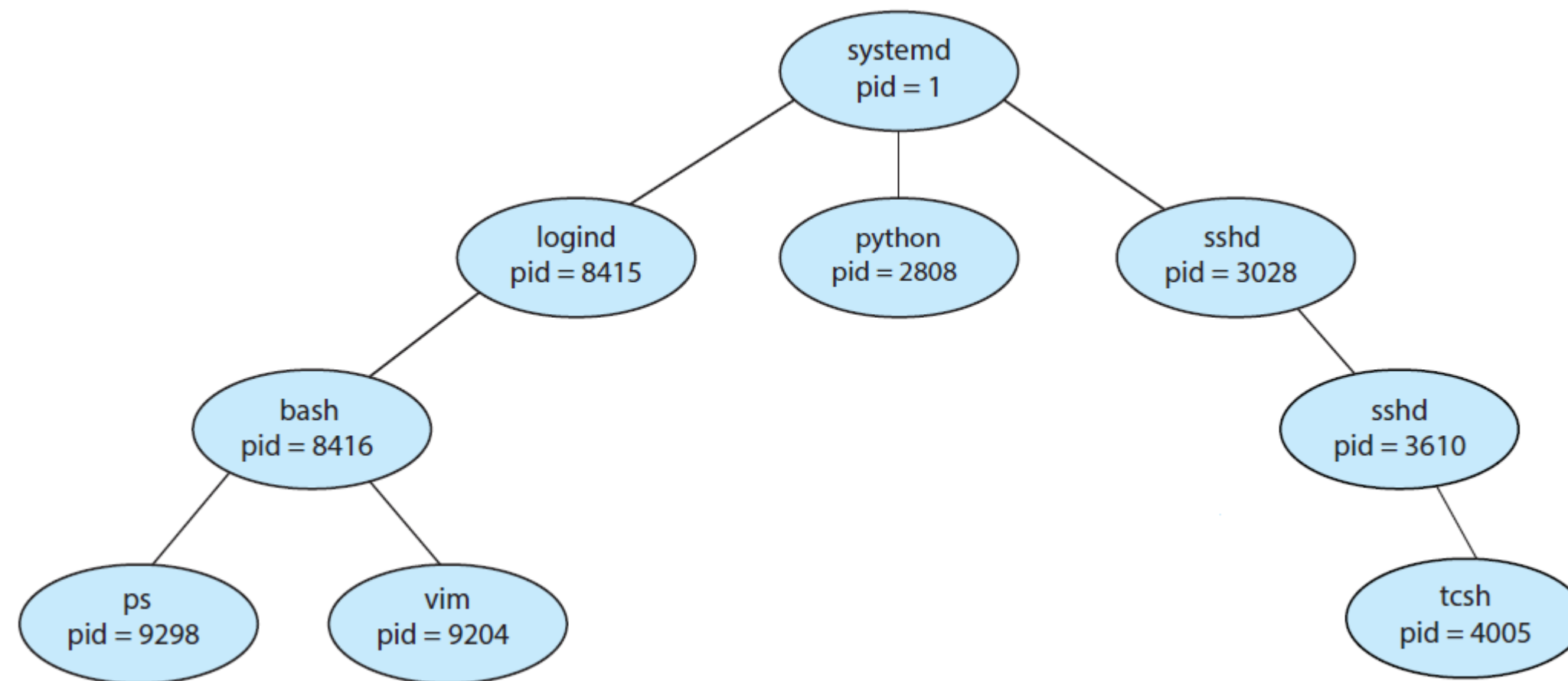
The options mean:

- **-e**: Show all processes.
- **-l**: Use a long listing format containing additional process information.

PID	PPID	S	CMD
1	0	S	systemd
2100	1	S	bash
3256	2100	R	python3
4102	1	S	vlc

FIGURE 3.7

A Tree of Processes on Linux



Reading the Tree

systemd

PID 1

Root of all user processes

- **systemd** is the main service manager in many Linux systems and commonly runs as **PID 1**.

logind

PID 8415

Direct logins → bash

- It represents login-session management.
- Under **logind**, the slide shows a **Bash** shell.

sshd

PID 3028

Remote logins via ssh

- **sshd** stands for Secure Shell Daemon
- The **SSH** daemon accepts remote connections and can create processes to manage user sessions.

SENTENCE TO REMEMBER

“The process tree shows who created whom.”

The *init* and *systemd* Processes

TRADITIONAL UNIX

init

System V *init* — PID 1, first process at boot.

MODERN LINUX

systemd

PID 1 — starts and manages system services.

Analogy: if the kernel is the stage and the physics of the world, *systemd* is the stage manager — lights, sound, actors and props in place before the show, and kept running during it.

The Command Names Behind the Processes

<code>ps</code>	process status	<code>ls</code>	list directory contents	<code>cp</code>	copy files or directories
<code>mv</code>	move or rename	<code>rm</code>	remove	<code>cd</code>	change directory
<code>pwd</code>	print working directory	<code>cat</code>	concatenate file contents	<code>man</code>	manual pages
<code>grep</code>	global regular expression print	<code>chmod</code>	change mode — permissions	<code>chown</code>	change owner
<code>kill</code>	send a signal to a process	<code>ssh</code>	secure shell	<code>scp</code>	secure copy
<code>bash</code>	Bourne Again SHell	<code>tcsh</code>	TENEX C SHell	<code>vim</code>	Vi IMproved — text editor

`bash` and `tcsh` are where you talk to Linux — the same conversation in two accents. `vim` is the Swiss Army knife of editing inside the terminal.

Useful Linux Commands

<code>ps -e</code>	List processes
--------------------	----------------

<code>ps -e</code>	List processes
<code>pstree</code>	Show parent–child links

<code>pstree</code>	Show parent–child links
<code>ls</code>	List directory contents

<code>ls</code>	List directory contents
<code>cd</code>	Change directory

<code>cd</code>	Change directory
<code>bash</code>	Run the Bash shell

<code>ps -el</code>	Long process listing
---------------------	----------------------

<code>ps -el</code>	Long process listing
<code>kill</code>	Signal a process

<code>kill</code>	Signal a process
<code>pwd</code>	Show working directory

<code>pwd</code>	Show working directory
<code>grep</code>	Search for text

<code>grep</code>	Search for text
<code>vim</code>	Open the Vim editor

What does the child receive?

With `fork()`, the child **inherits** much of the parent's execution environment — yet stays a **separate process** with its **own identity**.

TYPICALLY INHERITED WITH `FORK()`

1 **A copy of the address space**
Code, data, heap, stack

2 **Environment variables**
PATH, HOME, ...

3 **Open file descriptors**
Files and sockets already open

4 **Execution attributes**
User ID, working directory, limits

PARENT CAN PASS ARGUMENTS

```
$ python analysis.py patients.csv
```

WHY LIMIT IT?

One faulty child could **overload the system**.

ANALOGY

A manager creates a **new department** : it receives information, facilities, and permissions — but is **responsible for its own activities**.

What the Child Gets

Resources

CPU time, memory, files, I/O devices — from the OS, or a **subset of the parent's**.

Partitioned or shared

Split among children, or shared — memory, files.

Why restrict

Stops a process **overloading the system** with too many children.

Initialization data

Input from the parent — e.g. the file name `hw1.c` and output device.

Also inherited: **privileges, scheduling attributes**, open files and environment variables.

Two Choices at Creation Time

EXECUTION

Continue or wait?

- 1 Parent runs concurrently
- 2 Parent waits for children

ADDRESS SPACE

Same or new program?

- 1 Duplicate of the parent
- 2 A new program loaded

Two choices at creation time

Two **independent** decisions when a parent creates a child

EXECUTION

Continue or wait?

1 Parent runs concurrently

A web server keeps accepting connections

2 Parent waits for children

It pauses when the next task **depends on the child's result**

A manager assigns a special meal — then **keeps managing**, or **waits** for the chef.

ADDRESS SPACE

Same or new program?

1 Duplicate of the parent

`fork()` gives a logical copy — both continue after `fork()`

2 A new program loaded

The child calls `exec()` — it **replaces** the program, it does not create a process

Two students with **identical worksheets** — same start, solved **independently**.

Creating a Process with `fork()`

The child starts as a **copy** of the parent.

Both continue from the instruction after `fork()`.

Implemented efficiently with **copy-on-write**.

ANALOGY

A photocopied worksheet

Same questions — different answers written on each copy.

How Copy-on-Write works

- Instead of copying every writable memory page immediately, the operating system initially allows parent and child to reference many of the same physical memory pages.
- These pages are protected so that writes can be detected.
- When one process tries to modify a shared COW page, the operating system creates a private copy as needed.

One Call, Two Return Values

RETURN VALUE

MEANING

0

Running in the **child**

> 0

In the **parent** — value is the child's PID

-1

Failed — no child created

Output order is **not guaranteed** — the scheduler decides.

Predict the Result

```
printf("Start\n"); fork();  
printf("Hello\n");
```

How many times does **Hello** print?

1

2

3

4

Assume fork() succeeds.

Replacing the Program with `exec()`

Same process, **new program**.

No new process is created.

```
execlp("ls", "ls", "-l", NULL);
```

- This command replaces the currently running program with the Linux `ls -l` command, which displays a detailed list of files and directories.

On success, it never returns.

ANALOGY

An actor changing roles

Same actor — new costume, new script.

Before `exec()`

PID = 2500
Program A running



`exec("Program B")`



After `exec()`

PID = 2500
Program B running

fork () vs. exec ()

	fork ()	exec ()
Does	Creates a new process	Replaces the program
After	Both processes continue	New program starts
Image	Child resembles parent	Program image changes
Returns	In both processes	Not on success

fork creates a worker; exec gives it a new job.

Collecting the Child with `wait()`

- `wait()` allows a parent process to wait for one of its child processes to finish, collect its termination status, and then continue execution.
- `wait()` → Waits for a child to terminate and collects its status.
- `wait()` returns the child's PID and places termination information in status

The parent blocks until a child ends.

Then it collects the child's **exit status**.

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

ANALOGY

Waiting for a student's report

The professor continues once it is submitted.

How the Three Calls Work Together



- A parent process creates a child using `fork()`. The child executes another program using `exec()`, while the parent uses `wait()` to wait for the child to finish and collect its termination status.

SENTENCE TO REMEMBER

“Fork to create, exec to replace, wait to collect.”

FIGURE 3.8

Creating a Separate Process with `fork()`

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

int main()
{
    pid_t pid;

    /* fork a child process */
    pid = fork();

    if (pid < 0) { /* error occurred */
        fprintf(stderr, "Fork Failed");
        return 1;
    }
    else if (pid == 0) { /* child process */
        execlp("/bin/ls", "ls", NULL);
    }
    else { /* parent process */
        /* parent will wait for the child to complete */
        wait(NULL);
        printf("Child Complete");
    }

    return 0;
}
```

`pid == 0`

Child runs `/bin/ls`

`pid > 0`

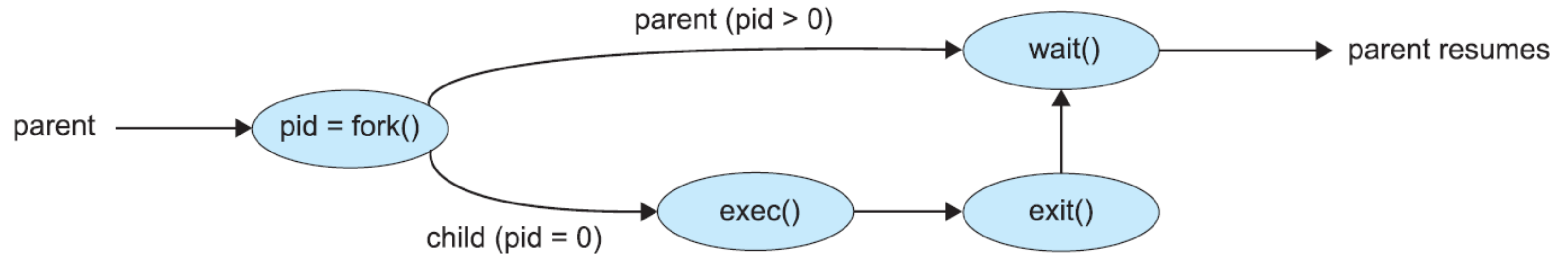
Parent waits, then prints

`pid < 0`

Fork failed

FIGURE 3.9

Process Creation Using `fork()`



The child's `exit()` releases the parent from `wait()`.

FIGURE 3.10

Creating a Process in Windows

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

int main(VOID)
{
    STARTUPINFO si;
    PROCESS_INFORMATION pi;

    /* allocate memory */
    ZeroMemory(&si, sizeof(si));
    si.cb = sizeof(si);
    ZeroMemory(&pi, sizeof(pi));

    /* create child process */
    if (!CreateProcess(NULL, /* use command line */
        "C:\\WINDOWS\\system32\\mspaint.exe", /* command */
        NULL, /* don't inherit process handle */
        NULL, /* don't inherit thread handle */
        FALSE, /* disable handle inheritance */
        0, /* no creation flags */
        NULL, /* use parent's environment block */
        NULL, /* use parent's existing directory */
        &si,
        &pi))
    {
        fprintf(stderr, "Create Process Failed");
        return -1;
    }
    /* parent will wait for the child to complete */
    WaitForSingleObject(pi.hProcess, INFINITE);
    printf("Child Complete");

    /* close handles */
    CloseHandle(pi.hProcess);
    CloseHandle(pi.hThread);
}
```

CreateProcess()

Creates and loads in one call

STARTUPINFO

Window and I/O settings

PROCESS_INFORMATION

New process and thread IDs

WaitForSingleObject()

Parent waits for the child

UNIX and Windows, Side by Side

	UNIX / Linux	Windows
Create	<code>fork()</code>	<code>CreateProcess()</code>
New program	<code>exec()</code> family	Same call
Wait	<code>wait()</code> / <code>waitpid()</code>	<code>WaitForSingleObject()</code>
Terminate	<code>exit()</code> · signals	<code>TerminateProcess()</code>

UNIX creates **then** replaces; Windows does **both at once**.

3.3.2

Process Termination

Every process ends — normally or early.

Normal Termination: `exit()` and `wait()`

- `exit()` → A process terminates itself.
- `kill()` → A process sends a signal to another process, often requesting termination.
- `exit()`: A guest finishes their stay and voluntarily checks out.
- `kill()`: The hotel manager asks a guest to leave or arranges their removal.

الإنهاء الطبيعي: استدعاء `exit()` ← تحرير الموارد ← إرسال حالة النهاية للأب عبر `wait()`

The process asks to be deleted

It finishes its final statement and calls `exit()`. In C this happens even without writing it — the run-time library calls `exit()` by default.

A status value goes to the parent

Typically an integer, collected by the waiting parent through `wait()`, which also returns *which* child terminated.

Resources are reclaimed

Physical and virtual memory, open files and I/O buffers are all deallocated by the operating system.

```
exit(1);
```

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

Normal Termination: `exit()`

- Here, `0` is an argument passed to the `exit()` function.
- It tells the operating system: **I am terminating successfully.**

```
exit(0);
```

`0` Success

`≠ 0` Error or other outcome

OS frees resources; the parent collects the status.

ANALOGY

A hotel guest checking out

The room is freed; a record stays briefly.

Termination by Another Process

- A process can be terminated by another authorized process, not only by itself.

e.g. `TerminateProcess()` in Windows — needs access rights.

Resource overuse

Exceeds its allowed limits

- The process exceeds allowed limits (CPU, memory, etc.).

Task not needed

e.g. a cancelled video job

- The user cancels a task, such as video rendering.

Parent exiting

Dependent children must stop

- child processes may be terminated when their parent exits, depending on OS or application policy.

Cascading Termination

Stopping the parent stops its
dependent children.

Depends on the OS and environment.

ON LINUX

Children usually survive — as orphans.

ANALOGY

A cancelled construction project

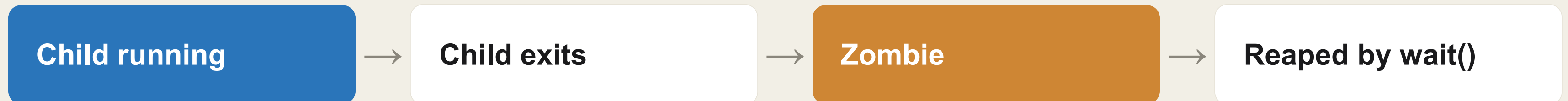
Every subcontractor's work stops too.

- الإنهاء المتسلسل يعني لما العملية الأب تنتهي، العمليات الأبناء التابعة لها تنتهي أيضاً، بشرط أن النظام يطبق هذه السياسة. أما في Linux، فالأبناء غالباً يستمرون في العمل ويصبحون Orphan Processes.
- إذا ترك الأستاذ الجامعة قبل انتهاء المشروع، فإن الطالب يواصل عمله، ولكن تحت إشراف أستاذ آخر.
- وبالمثل، في Linux، عندما تنتهي العملية الأب بينما الابن لا يزال يعمل، يصبح الابن Orphan Process، ويتولى نظام التشغيل إعادة إسناده إلى عملية أخرى، مثل PID 1 أو عملية Subreaper. (حاصد العمليات الفرعية)

Zombie Processes

Terminated — but the parent has not yet called `wait()`.

عملية منتهية لكن الأب لم يستدع `wait()`



Only the exit-status record remains in the process table.

- A Zombie is a dead process whose parent has not yet collected its death certificate.

■ عملية الابن انتهت بالفعل، لكن الأب لم يستدع `wait()` لاستلام حالة انتهائها. لذلك يظل سجل صغير لها داخل نظام التشغيل حتى يتم جمع حالتها وحذف السجل.

Orphan Processes

The parent ends while the child **keeps running**.

يتيم — إذا مات الأب قبل الابن يتبناه systemd

ANALOGY

A student whose supervisor retires

A new supervisor is appointed.

Linux reparents it to **PID 1** or a subreaper.

Zombie vs. Orphan

	Zombie	Orphan
Child	Has terminated	May still be running
Parent	Hasn't reaped it	Has terminated
Remains	Exit-status record	Gets a new parent
Needs	Reaping	Nothing — continues

Zombie: **child** died first. Orphan: **parent** died first.

Knowledge Check

1	Which call creates a child on UNIX?	<code>fork()</code>
2	<code>fork()</code> value in the child?	0
3	What happens when <code>exec()</code> succeeds?	Image replaced
4	Dead child, status uncollected?	Zombie
5	Parent dies, child keeps running?	Orphan
6	Which call collects a child's status?	<code>wait()</code>

ONE SENTENCE TO REMEMBER

fork

creates

exec

replaces

wait

collects

exit

terminates

— and the OS manages the entire process family.

RECAP

Key Terms from Section 3.3

parent / child

tree of processes

process identifier (pid)

systemd

init

ps -el

pstree

fork()

exec() / execlp()

wait()

exit()

CreateProcess()

STARTUPINFO

PROCESS_INFORMATION

WaitForSingleObject()

TerminateProcess()

cascading termination

zombie

orphan

importance hierarchy

COMING NEXT

3.4

Interprocess Communication

Shared memory and message passing.