

Phiên 3 — fork, exec, wait: zombie và orphan

Bốn lời gọi hệ thống dựng nên toàn bộ vòng đời một tiến trình. Phiên này làm hỏng vòng đời đó một cách có chủ đích, nhìn cái xác nằm lại trong /proc, rồi dọn.

THỜI LƯỢNG	THUỘC	MỐC	ĐẦU VÀO
2 giờ, không lẩn giờ	Phase 1 — Process, fd, signal	chuẩn bị cho M-L1 (Phiên 8)	Phiên 2 đã xong
ARTIFACT			
ba chương trình C nhỏ + note			

Phase 0 dựng chỗ để phá. Phase 1 bắt đầu phá — và bắt đầu từ tầng nền: cách một tiến trình sinh ra, thay hình đổi dạng, chết, và được dọn xác. Mọi thứ về sau (container thấy PID 1 là chính nó, `docker stop` phải chờ 10 giây, service systemd biết con nó chết) đều là hệ quả của bốn lời gọi trong phiên này.

CÁCH DÙNG TRANG NÀY

Phần cơ chế đọc trước khi gõ — nó không tính vào timebox của block. Ba chương trình C là **sản phẩm của phiên**, nên **không có lời giải ở bất kỳ đâu trên trang này**, kể cả sau spoiler. Thứ nằm trong spoiler chỉ là lệnh vận hành: biên dịch, quan sát, đọc `/proc`.

MỤC 00

Trước khi bấm giờ

TOÀN BỘ PHIÊN NÀY CHẠY TRONG GUEST

Không phải trên máy trạm. Phiên này cố tình tạo tiến trình không được dọn, và phần Stretch còn dựng tới giới hạn số tiến trình. Đó chính là lý do Phase 0 tồn tại. Chụp một snapshot trước khi bắt đầu — nếu chưa quen tay thì đọc lại Phiên 2.

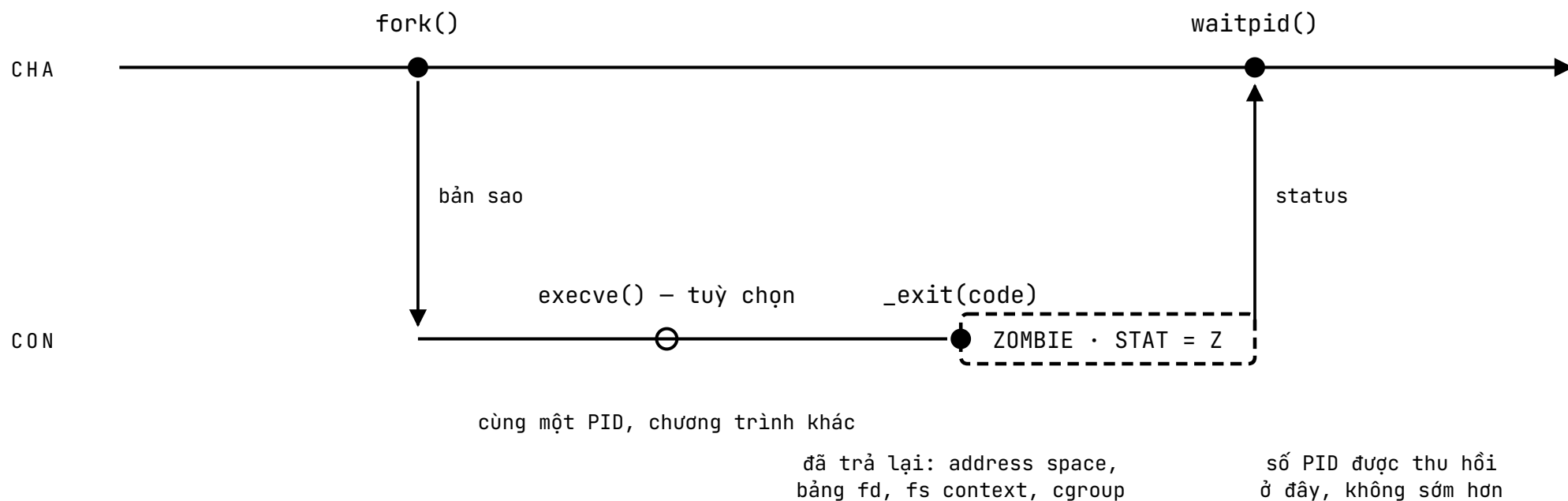
- ☐ VM dựng lại được và ssh vào được. Hỏng chỗ này thì dừng, quay về Phiên 2 — cả phiên hôm nay nằm trong guest.
- ☐ Trong guest: `gcc --version` chạy được. Không có thì `build-essential` chưa vào — sửa ở `lab-reset.sh` chứ đừng `apt install` tay, nếu không lần reset sau lại thiếu.
- ☐ `strace -V` chạy được. Phiên này dùng `strace` để thấy tên syscall *thật*, khác tên hàm C — đó là một trong những thứ đáng nhớ nhất của buổi.
- ☐ `ps-tree -V` chạy được. Cái này **hay thiếu**: nó thuộc gói `psmisc`, không đi kèm `procps`. Thiếu thì thêm vào `lab-reset.sh`.
- ☐ Ghi vào note ba con số của guest, sẽ dùng ở mục Bẫy và Stretch: `cat /proc/sys/kernel/pid_max`, `ulimit -u`, và `nproc`.
- ☐ Ghi luôn `uname -r` và `systemctl --version | head -1`. Hai con số này quyết định câu trả lời của Block B và của phần `pid_max`.
- ☐ **Hai phiên terminal** vào cùng VM, mở sẵn. Một để chạy chương trình, một để quan sát trong lúc nó còn sống. Zombie chỉ tồn tại vài chục giây — không kịp mở terminal thứ hai là mất cảnh.

Bốn động từ, và không cái nào làm việc của cái kia

Trên Linux **không có** lời gọi "tạo một tiến trình chạy chương trình X". Câu nói đó gộp hai việc rời nhau, và việc gộp chúng trong đầu là nguồn của gần như mọi hiểu nhầm sau này. Bốn động từ, bốn việc khác hẳn nhau:

LỜI GỌI	LÀM GÌ	KHÔNG LÀM GÌ	BỎ QUÊN THÌ THẤY GÌ
fork	nhân bản tiến trình đang gọi thành hai	không nạp chương trình mới, không đổi gì trong bản sao	không kiểm giá trị trả về $\Rightarrow$ cả hai nhánh cùng chạy một đoạn code
execve	thay chương trình <i>bên trong</i> tiến trình hiện tại	không tạo tiến trình mới, không trả về khi thành công	code sau exec vẫn chạy $\Rightarrow$ exec đã fail, không phải "chạy xong rồi về"
_exit	chấm dứt, để lại 8 bit trạng thái cho cha	không xoá tiến trình khỏi hệ thống	không có gì sai — cái xác nằm lại là đúng thiết kế
wait	thu trạng thái và giải phóng phần còn lại của con	không giết được gì, không tác động lên con đang sống	zombie chất đống cho tới khi cha chết

Bốn trang man tương ứng nên mở sẵn trong lúc làm phiên này — phần lớn câu hỏi ở các block dưới trả lời được bằng cách đọc đúng một đoạn trong đó.



Dải ZOMBIE kéo dài đúng bằng khoảng thời gian cha chưa gọi wait. Cha không bao giờ gọi $\Rightarrow$ nó kéo dài tới khi chính cha chết.

Cái chết của một tiến trình có hai thời điểm, không phải một: lúc nó ngừng chạy, và lúc cha thu xác. Khoảng giữa hai thời điểm đó là toàn bộ nội dung của Block A.

fork nhân bản cái gì — và bỏ lại cái gì

Man page mở đầu bằng một câu tuyệt đối: con là *"an exact duplicate of the parent process except for the following points"*. Chữ "except" mới là chỗ đáng đọc, vì mặc định là **kế thừa hết** — danh sách ngoại lệ ngắn và có logic.

Về bộ nhớ, đừng hình dung một phép copy. Cùng man page, mục NOTES: *"Under Linux, fork() is implemented using copy-on-write pages, so the only penalty that it incurs is the time and memory required to duplicate the parent's page tables, and to create a unique task structure for the child."* Chi phí thật là **nhân bản bảng phân trang**, không phải nhân bản dữ liệu — nên fork một tiến trình 8 GiB không tốn 8 GiB, nhưng cũng không hề miễn phí.

CON KHÔNG KẾ THỪA	VÌ SAO HỢP LÝ
tập signal đang chờ (pending)	signal gửi cho cha không có lý do gì phải nhân đôi
alarm, setitimer, timer_create	hai tiến trình cùng nổ chuông một lần đặt là vô nghĩa
memory lock (mlock), semaphore adjustment, async I/O đang dở	đều là tài nguyên gắn với một tiến trình cụ thể
record lock kiểufcntl truyền thống	nhưng OFD lock và flock thì có kế thừa — ba loại khoá, hai hành vi khác nhau
bộ đếm thời gian CPU (times) và getrusage	đặt lại về 0, nếu không thì mọi phép đo đều sai
PR_SET_PDEATHSIG	bị xoá ⇒ muốn "con chết theo cha" thì phải đặt lại <i>trong con</i> , sau fork

CHỖ TÀI LIỆU PHỔ BIẾN NÓI SAI • IN HAI LẦN KHÔNG PHẢI LỖI CỦA FORK

Buffer của `stdio` là bộ nhớ user-space, nên nó **được nhân bản nguyên trạng, kể cả phần chưa flush**. Cả hai tiến trình về sau đều flush bản của mình ⇒ dòng chữ ra hai lần.

Lời khuyên phổ biến "nhớ thêm `\n`" **chỉ đúng khi stdout là terminal**. Luật buffering là: stream nối vào terminal thì line-buffered, nối vào file hay pipe thì *fully buffered*, còn stderr thì không buffer. Nghĩa là chương trình có `\n` đầy đủ vẫn in hai lần ngay khi bạn `> out.txt` hoặc `| tee` — đây chính là dạng bug "chạy tay thì đúng, chạy trong CI thì sai" kinh điển.

Hai cách sửa, và chúng khác nhau: `fflush(NULL)` *trước* khi fork (dọn sạch buffer trước khi nhân đôi), và dùng `_exit` thay `exit` trong con (không chạy atexit, không flush bản sao). Block A sẽ bắt bạn nhìn thấy hiện tượng này trước khi sửa.

BẦY • STRACE IN "CLONE", KHÔNG PHẢI "FORK"

Từ glibc 2.3.3, hàm `fork()` không gọi syscall cùng tên nữa mà gọi `clone`. Man page phát biểu tương đương ngữ nghĩa là *"a call to fork() is equivalent to a call to clone(2) specifying flags as just SIGCHLD"*; `strace` thực tế sẽ hiện thêm `CLONE_CHILD_SETTID|CLONE_CHILD_CLEARTID` — đó là sổ sách nội bộ của thư viện thread, không đổi ngữ nghĩa nào mà chương trình quan sát được.

Đi xa hơn một bước: trên arm64 và riscv **không hề có syscall `fork`**, chỉ có `clone`. Nên "hàm C tên gì" và "syscall tên gì" là hai câu hỏi khác nhau — thói quen tách bạch hai tầng đó sẽ trả công suất các phase sau.

Bảng đối chiếu: qua fork thì sao, qua exec thì sao

Đây là bảng đáng chép tay vào note. Hai phép biến đổi này **không đối xứng**, và mấy dòng lệch nhau là nơi sinh ra những con bug khó nhất.

THUỘC TÍNH	QUA FORK	QUA EXECVE
PID	mới	giữ nguyên
file descriptor đang mở	kế thừa	giữ, trừ cái có cờ FD_CLOEXEC
signal mask	kế thừa	giữ – không hề bị reset
signal đang chờ (pending)	xoá sạch	giữ
signal handler đã cài	kế thừa	reset về SIG_DFL; signal đang <i>ignore</i> thì vẫn ignore
alarm / setitimer	không kế thừa	giữ
POSIX timer (timer_create)	không kế thừa	không giữ
vùng nhớ (mmap, heap, stack)	nhân bản kiểu COW	vứt hết, dựng lại từ file thực thi
handler atexit	kế thừa	mất
cwd, umask, nice, rlimit	kế thừa	giữ

ĐỌC KỸ HAI DÒNG NGƯỢC NHAU

`alarm`/`setitimer` **chết ở fork nhưng sống qua exec**; POSIX timer thì chết ở cả hai. Rất nhiều tài liệu gộp chúng thành một dòng "timer" – sai ở cả hai chiều. Tương tự, tập signal đang chờ bị xoá ở fork nhưng được *giữ* qua exec: đúng ngược nhau.

Man page của `execve` còn dành hẳn một đoạn để dập một cách nói phổ biến: *"One sometimes sees execve() ... described as 'executing a new process'... This is a highly misleading description: there is no new process; many attributes of the calling process remain unchanged (in particular, its PID)."*

Một zombie thật sự còn giữ những gì

Man page định nghĩa gọn: *"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"*.

Chữ *minimal* đó cụ thể tới mức nào thì phải xem source. Trong `do_exit()`, thứ tự các bước là bằng chứng: address space được trả lại (`exit_mm`), bảng file descriptor được trả lại (`exit_files`), fs context, IPC, namespace, cgroup – **tất cả đều xong trước**, rồi mới tới `exit_notify()` là chỗ đặt `EXIT_ZOMBIE`.

Không. Đến lúc thành zombie thì RSS đã về 0, file đã đóng, kernel stack đã trả. Còn lại là `task_struct` + cấu trúc signal + một `struct pid` — cỡ vài KB bộ nhớ *kernel*, và quan trọng hơn cả: **một số PID**.

Điều này **kiểm được tại chỗ**, và Block A bắt bạn kiểm: `/proc/<zombie>/status` không có bất kỳ dòng `Vm*` hay `Rss*` nào — không phải "bằng 0", mà là **không tồn tại dòng đó**, vì trình sinh nội dung bỏ qua cả khối khi tiến trình không còn `mm`. Và `/proc/<zombie>/fd/` rỗng.

Cái giá thật khi zombie tích tụ là **cạn PID**, không phải cạn RAM. Nguồn chính chủ của `tini` nói đúng chữ đó: zombie có thể *"starve your entire system for PIDs"*.

`kill -9` lên một zombie **không làm gì cả**, và tệ hơn: nó **trả về 0**, nên trông như đã thành công. Cơ chế nằm ở `prepare_signal()` trong kernel: mọi lần `exit_group()` đều bật cờ `SIGNAL_GROUP_EXIT`, và hàm này vứt signal ngay khi thấy cờ đó, kèm đúng một dòng comment *"The process is in the middle of dying, drop the signal."* Không có gì để giết vì không còn gì chạy.

Cách duy nhất làm zombie biến mất là **cha gọi `wait`**, hoặc **cha chết** để zombie được chuyển sang một reaper biết dọn. Mẹo dân gian "giết cha nó đi" vì thế *đúng*, nhưng đúng vì lý do khác với lý do người ta hay giải thích.

Ghi chú về nguồn: không man page nào phát biểu thẳng điều này — bằng chứng ở trên lấy từ kernel source. Trong phiên, hãy tự kiểm bằng thực nghiệm ở Block A chứ đừng tin vì trang này nói.

Tạo một zombie, rồi dọn nó

BLOCK NÀY KHÔNG CÓ LỜI GIẢI

Chương trình C là thứ đang được luyện. Dưới đây có mô tả *việc*, các *câu hỏi phải tự trả lời trước khi gõ*, và *lệnh vận hành* để quan sát. Không có mã nguồn, ở đây hay ở bất kỳ spoiler nào.

Làm ra zombie và nhìn tận mắt

Chương trình 1 — cha fork ra một con; con thoát ngay với một mã thoát *bạn tự chọn*; cha ngủ đủ lâu để bạn kịp quan sát ở terminal thứ hai.

Trả lời trước khi gõ dòng đầu tiên:

- Chỉ bằng giá trị `fork()` trả về, làm sao biết mình đang ở nhánh nào? Và vì sao **bắt buộc** phải xử lý riêng trường hợp trả về âm — bỏ qua nó thì chuyện gì xảy ra khi hệ thống cạn PID?
- Cha muốn in PID của con, con muốn in PID của chính nó. Hai giá trị đó lấy từ hai chỗ khác nhau — chỗ nào?
- Con nên kết thúc bằng `exit` hay `_exit`? Đọc lại hộp cảnh báo về `stdio` ở mục 01 trước khi quyết định, và **ghi lý do vào note**.

- Chọn mã thoát nào? Đừng chọn 0 hay 1 – chọn một số dễ nhận ra giữa đồng output, ví dụ một số hai chữ số không tròn.

Việc cần làm:

1. Biên dịch với cảnh báo bật hết. Trình biên dịch sẽ bắt giúp bạn ít nhất một lỗi ở bài này nếu bạn để nó nói.
2. Chạy. Trong lúc cha còn ngủ, ở terminal thứ hai: tìm dòng có `STAT` bắt đầu bằng `Z`. Ghi lại PID và PPID của nó.
3. **Phép đo quan trọng nhất của block:** đếm số dòng của `/proc/<zombie>/status` và của `/proc/<cha>/status`. Hai con số phải khác nhau rõ rệt. Liệt kê ít nhất hai dòng có ở cha mà không có ở zombie – đây là bằng chứng cho mục 01, và là thứ đi vào Definition of Done.
4. `ls /proc/<zombie>/fd/` và `cat /proc/<zombie>/cmdline`. Kết quả giải thích luôn vì sao `ps` hiển thị tên trong ngoặc vuông kèm `<defunct>` thay vì dòng lệnh đầy đủ.
5. Thử `kill -9` lên zombie. Ghi lại **cả hai**: exit code của chính lệnh `kill`, và trạng thái của zombie sau đó. Một trong hai sẽ làm bạn ngạc nhiên.
6. Chạy lại chương trình nhưng chuyển hướng ra file, rồi `cat` file đó. So số dòng với lần chạy thẳng ra terminal. Khác nhau thì **đừng sửa vội** – hiểu vì sao trước đã.
7. `strace` chương trình, chỉ lọc bốn syscall liên quan. Ba trong bốn tên đó không trùng với tên hàm C bạn đã gọi. Ghi vào note ba cặp tên đó.

■ HIỆN LỆNH QUAN SÁT • BLOCK A

Đây là lệnh vận hành, không phải khung chương trình. Thay `<pid>` bằng số thật.

BIÊN DỊCH VÀ CHẠY

TRONG GUEST

```
$ gcc -Wall -Wextra -O0 -g -o zombie zombie.c
$ ./zombie # terminal 1
$ ./zombie > out.txt ; wc -l out.txt # lần chạy thứ hai, so số dòng
```

QUAN SÁT TRONG LÚC CHA CÒN NGỦ

TERMINAL 2, CÙNG GUEST

```
# tìm mọi tiến trình có trạng thái Z; neo ^Z để chỉ khớp ký tự đầu
$ ps -eo pid,ppid,stat,comm --no-headers | awk '$3 ~ /^Z/'

# nhìn cả cây để thấy AI đang không dọn
$ ps -eo pid,ppid,stat,comm --forest | grep -A2 -B2 defunct
$ pstree -p <pid cha>

# phép đo chính: hai con số này phải khác nhau
$ wc -l /proc/<zombie>/status /proc/<cha>/status
$ diff <(cut -d: -f1 /proc/<cha>/status) <(cut -d: -f1 /proc/<zombie>/status)

$ grep -E '^(Name|State|Pid|PPid|Threads):' /proc/<zombie>/status
$ ls -l /proc/<zombie>/fd/ ; cat /proc/<zombie>/cmdline | wc -c

# thử giết cái đã chết – chú ý exit code
$ kill -9 <zombie> ; echo "kill tra ve $?"
$ ps -o pid,stat= -p <zombie>
```

```
$ strace -f -e trace=clone,clone3,fork,execve,wait4,waitid,exit,exit_group ./zombie
# ghi vào note: fork() hiện ra là gì, exit() hiện ra là gì, waitpid() hiện ra là gì
```

Dọn bằng waitpid

Chương trình 1b – bản sửa: cha gọi `waitpid` và in ra mã thoát mà nó thu được, đối chiếu với con số bạn đã chọn.

Trả lời trước khi gõ:

- Tham số `pid` của `waitpid` có bốn nghĩa tùy dấu và giá trị. Bài này cần nghĩa nào, và nếu sau này có mười con thì đổi sang nghĩa nào?
- Mặc định `waitpid` chọn. Nếu muốn cha vừa chờ vừa làm việc khác thì thêm cờ nào – và khi đó giá trị trả về `0` nghĩa là gì, khác gì với `-1`?
- Biến `status` nhận về **không phải** mã thoát. Muốn lấy mã thoát ra thì phải qua macro nào, và macro đó chỉ được dùng sau khi macro nào trả về đúng?

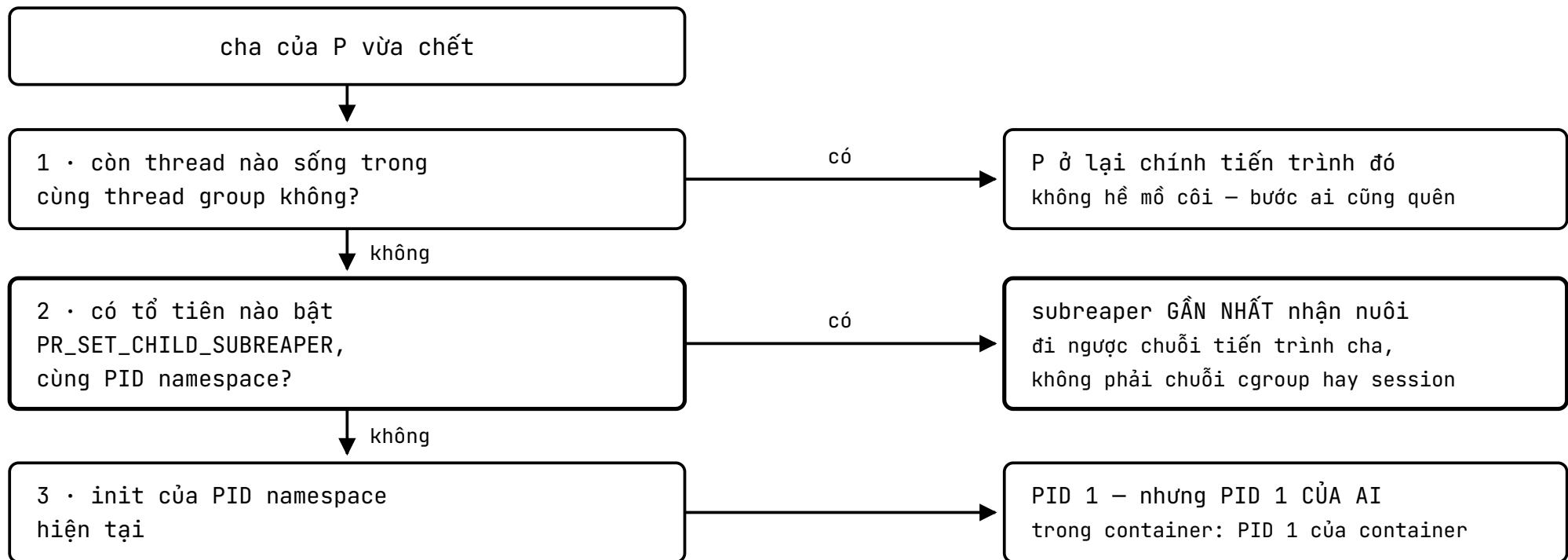
Nghiem thu Block A

- ☐ Có trong note một dòng `ps` thật với `STAT` là `Z`, kèm PID và PPID.
- ☐ Nêu được ít nhất hai dòng có trong `/proc/<cha>/status` mà `/proc/<zombie>/status` không có.
- ☐ Ghi được exit code của `kill -9` lên zombie và trạng thái sau đó, kèm một câu giải thích vì sao hai thứ đó không mâu thuẫn nhau.
- ☐ Giải thích được chênh lệch số dòng giữa chạy ra terminal và chạy ra file – bằng cơ chế, không bằng "chắc là do buffer".
- ☐ Bản 1b không để lại zombie nào và in đúng mã thoát đã chọn.

MỤC 02

Ai nhận nuôi trẻ mồ côi

Khi cha chết trước con, con thành *orphan* và được gán cha mới. Câu trả lời phổ biến – "về PID 1" – là phiên bản rút gọn tới mức sai. Luật thật có **ba bước**, và kernel viết thẳng nó thành comment ngay trên hàm quyết định:



Chỉ khi cả bước 1 và bước 2 trượt thì câu trả lời "về PID 1" mới đúng – và nó vẫn còn phụ thuộc mình đang ở namespace nào.

Ba bước, không một bước. Bước 2 là thứ làm câu trả lời khác nhau giữa hai cách khởi chạy cùng một chương trình trên cùng một máy.

Man page phát biểu bước 2 gọn nhất: *"When a process becomes orphaned ... that process will be reparented to the nearest still living ancestor subreaper"*. Cờ này có từ Linux 3.4, **không** kế thừa qua fork, nhưng **giữ** qua exec – hai tính chất trái ngược nhau đó sẽ quyết định cách bạn viết wrapper ở Block B.

CHỖ TÀI LIỆU PHỔ BIẾN NÓI SAI • "ORPHAN LUÔN VỀ PID 1"

`systemd --user` tự đặt mình làm subreaper từ **systemd 188**. Trong source hiện tại, hàm `make_reaper_process()` có một nhánh thoát sớm: nếu tiến trình gọi *chính là* PID 1 thì nó trả về ngay mà không gọi `prctl` – vì PID 1 đã là reaper của namespace theo định nghĩa. Chỉ manager *không phải* PID 1 mới thật sự bật cờ.

Nhưng – và đây là chỗ cũng dễ nói sai theo chiều ngược lại – điều quyết định không phải "bạn đang trong session của user nào", mà là **chuỗi tiến trình cha có đi qua** `systemd --user` hay không:

- Trên desktop hiện đại, ứng dụng được `systemd --user` khởi chạy trong một scope riêng ⇒ nó là tổ tiên ⇒ orphan về nó.
- **Vào bằng ssh:** shell là con của `sshd`, mà `sshd` là con của PID 1. `systemd --user` tồn tại nhưng nằm ở một *nhánh song song*, không phải tổ tiên ⇒ orphan về **PID 1**.
- Bên trong một service systemd: chuỗi cha là service → PID 1, không có subreaper trung gian ⇒ **PID 1**.

VM lab được vào bằng ssh, nên Block B sẽ cho `PPid = 1`. Đó là kết quả đúng – nhưng đúng vì chuỗi cha không có subreaper nào, *không* vì "luôn luôn là PID 1". Việc của Block B là chứng minh sự khác biệt đó bằng thực nghiệm.

VÌ SAO SYSTEMD VẪN KHÔNG MẤT DẤU TIẾN TRÌNH

Nếu reparent làm rối cây tiến trình, làm sao `systemctl stop` vẫn dọn sạch được một service có tiến trình double-fork? Vì systemd **không** theo dõi bằng quan hệ cha-con: nó theo dõi bằng **cgroup**. Reparent đổi cha, nhưng không đổi cgroup – muốn ra khỏi cgroup phải có thao tác tường minh. Đây là ý tưởng sẽ quay lại ở Phase 4 và là lý do `KillMode=` mặc định là `control-group`.

BLOCK B • ~30 PHÚT

Orphan và reparent

Chương trình 2 – cha fork ra một con rồi thoát ngay; con lặp trong khoảng 15 giây, mỗi giây in ra PID của chính nó và PID cha *hiện tại*.

Trả lời trước khi gõ:

- Cha thoát ngay ⇒ shell thấy lệnh đã xong và trả prompt về, nhưng con vẫn đang ghi vào cùng terminal. Điều đó có làm bạn đọc nhầm output không, và cách gọn nhất để tách hai luồng ra là gì?
- PID cha *hiện tại* phải hỏi lại mỗi vòng lặp, không phải lưu một lần lúc bắt đầu. Vì sao – và nếu lưu một lần thì bạn sẽ thấy gì?
- Output của con có ra kịp không, hay nằm trong buffer tới lúc chương trình thoát? Đọc lại hộp về `stdio` ở mục 01. Có ít nhất hai cách xử lý; chọn một và ghi lý do.

Việc cần làm:

1. Chạy và quan sát con số PPid **nhảy**. Ghi lại giá trị trước và sau.
2. Đối chiếu cùng một sự thật bằng **ba đường độc lập**: giá trị chương trình tự in ra, `ps -o ppid=`, và dòng `PPid` trong `/proc/<pid>/status`. Ba đường phải khớp; không khớp là bạn đang nhìn nhầm tiến trình.
3. Tra xem cha mới đó thật sự là ai – tên chương trình, không chỉ con số.
4. **Phần chính**: chạy lại đúng chương trình đó, nhưng lần này dưới một tổ tiên có bật subreaper, rồi so con số PPid mới với lần trước. Hai đường để làm, chọn một:
 - Nhờ systemd làm hộ: khởi chạy nó như một scope thuộc user manager. Rẻ và không phải viết thêm code, nhưng phụ thuộc user session có sẵn – nếu lệnh báo lỗi kết nối thì kiểm `systemctl --user status` và biến `XDG_RUNTIME_DIR` trước khi kết luận là hỏng.
 - **Chương trình 3** – tự viết wrapper: bật `PR_SET_CHILD_SUBREAPER` cho chính mình, rồi khởi chạy chương trình 2 và ngồi chờ đủ lâu. Gợi ý một câu: chờ này *không* kế thừa qua fork nhưng *giữ* qua exec – điều đó ràng buộc bạn phải đặt cờ ở đâu trong trình tự fork/exec, và đó chính là câu hỏi của bài.
5. Dựng ngược cả chuỗi tổ tiên của con trong từng trường hợp, và chỉ đúng vị trí mất xích subreaper trên chuỗi đó.

■ HIỆN LỆNH QUAN SÁT · BLOCK B

BA ĐƯỜNG ĐỌC CÙNG MỘT SỰ THẬT

TRONG GUEST

```
$ ps -o ppid= -p <pid con>
$ grep -E '^(Name|Pid|PPid|State):' /proc/<pid con>/status
$ ps -p <ppid> -o pid,comm,args --no-headers    # cha moi la AI

# ca chuoai to tien, tu con nguoc len goc
$ pstree -s -p <pid con>
```

ĐƯỜNG SYSTEMD, CHO BƯỚC 4

TRONG GUEST

```
# kiem user manager co that su chay khong TRUOC khi thu
$ systemctl --user is-system-running ; echo "XDG_RUNTIME_DIR=$XDG_RUNTIME_DIR"
$ pgrep -a -u "$(id -u)" -f 'systemd --user'

# chay chuong trinh 2 duoi mot scope thuoc user manager
$ systemd-run --user --scope --quiet ./orphan
```

ĐỪNG LÀM THẾ NÀY

TRONG GUEST

```
# SAI: truong comm in ra KHONG escape, ten co dau cach hay ')' la lech het truong
$ awk '{print $4}' /proc/<pid>/stat

# dung: hoac doc status (key-value), hoac cat tu dau ')' CUOI CUNG
$ awk -F')' '{print $NF}' /proc/<pid>/stat | awk '{print $2}'
```

Nghiệm thu Block B

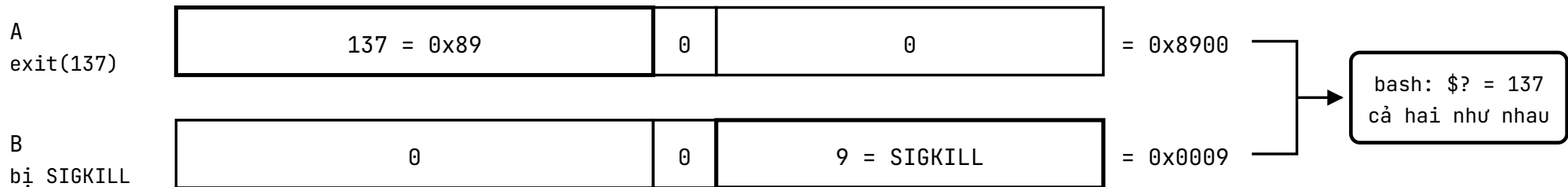
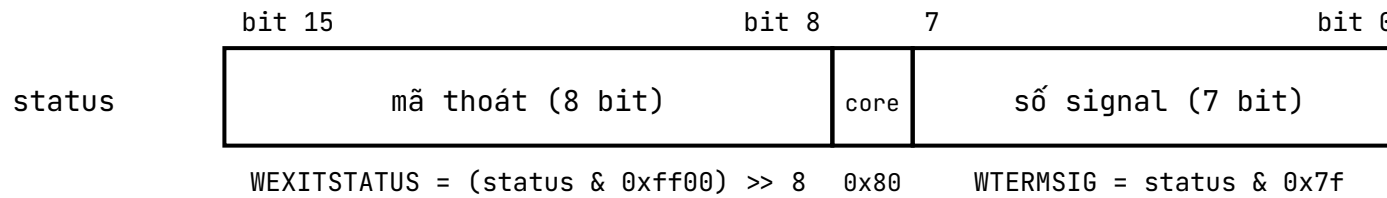
- ☐ Có **hai con số PPid khác nhau** cho cùng một chương trình, và một câu giải thích vì sao khác.
- ☐ Gọi được **tên** của reaper trong từng trường hợp, không chỉ con số.
- ☐ Chỉ ra được mắt xích subreaper nằm ở đâu trên chuỗi tổ tiên – hoặc nói được nó không tồn tại trên chuỗi đó.

MỤC 03

Exit status: 16 bit, và chỗ shell làm mất thông tin

Giá trị mà `waitpid` đặt vào `status` **không phải** mã thoát. Nó là một số nguyên gói nhiều thứ vào các vùng bit khác nhau – và cách shell trình bày lại số đó làm mất một phần thông tin.

BỐ TRÍ BIT CỦA status



Hai giá trị status khác nhau hoàn toàn. Phép cộng 128 + số signal chỉ xảy ra ở tầng shell – trong kernel không có con số 128 nào cả. Thông tin "chết vì signal hay tự thoát" mất ở đúng chỗ đó.

Cùng một `$? = 137` đến từ hai sự kiện không liên quan gì nhau. Chỉ ở tầng syscall mới phân biệt được, và đó là lý do phải dùng macro thay vì đọc con số.

BA TẦNG, BA LUẬT KHÁC NHAU

Kernel cắt mã thoát còn 8 bit rồi dịch trái 8: trong `kernel/exit.c`, `exit_group` gọi thẳng `do_group_exit((error_code & 0xff) << 8)`. Đó là lý do `exit(256)` tới tay cha thành 0 và `exit(-1)` thành 255.

POSIX cố ý không chuẩn hoá bố trí bit. Mục RATIONALE nói rõ: dùng macro thay vì bit encoding cụ thể là để “eliminate a limitation on the number of signals an implementation can support”, và bảo đảm duy nhất là status bằng 0 tương đương `_exit(0)`. Nghĩa là các con số `0xff00`, `0x7f`, `0x80` trong sơ đồ trên là chi tiết hiện thực của glibc + Linux, không phải chuẩn – biết để hiểu, nhưng viết code thì dùng macro.

Shell làm phẳng tất cả về một số 8 bit. Bash tự nhận trách nhiệm: “When a command terminates on a fatal signal whose number is N, Bash uses the value 128+N as the exit status” – chủ ngữ là Bash. POSIX chỉ bắt buộc “lớn hơn 128”, không chốt đúng `128+N`, nên shell khác có quyền chọn khác.

BẦY · DÀI 1-125 KHÔNG CÓ CHUẨN NÀO CẢ

POSIX chỉ quy định `127` (không tìm thấy lệnh), `126` (tìm thấy nhưng không chạy được) và “lớn hơn 128” cho chết vì signal. Dải `1`–`125` là không gian tự do của ứng dụng. Bộ số `64`–`78` trong `sysexit.h` là quy ước BSD, và chính man page của nó ghi “The choice of an appropriate exit value is often ambiguous.” Còn `125` là quy ước riêng của vài công cụ coreutils (`env`, `timeout`, `nice`) để tự phân biệt “lỗi của chính wrapper” với “lỗi của chương trình được gọi”. Đừng gán ý nghĩa cho một con số chỉ vì thấy nó trong một bảng trên mạng.

BLOCK C · ~20 PHÚT

Đọc đúng status thô

Việc cần làm: mở rộng chương trình 1b để nó in ra giá trị `status` thô dưới dạng hex, rồi mới phân loại bằng macro. In cả hai – con số thô là thứ làm bài này có ý nghĩa.

- Cho con thoát theo **bốn** đường khác nhau, và **đoán trước** giá trị hex rồi mới chạy:
 - thoát bình thường với mã bạn đã chọn;
 - thoát với `256`, rồi với `-1`;
 - bị `SIGKILL` gửi từ terminal thứ hai;
 - tự gây lỗi truy cập bộ nhớ – quan sát bit core dump có bật không, và kiểm tra `/proc/sys/kernel/core_pattern` để biết vì sao có hoặc không có file dump.
- Với mỗi đường, ghi vào bảng: hex thô → macro nào trả về đúng → giá trị macro đó cho ra.
- Chạy lại bốn trường hợp qua shell và ghi thêm cột `$?`. Chỉ ra đúng ô nào bị mất thông tin so với cột hex.
- Phép chứng minh cuối:** làm một chương trình thoát với mã `137`, và một chương trình bị `kill -9`. Shell cho cùng một `$?`. Trình bày bằng chứng chúng khác nhau.

ĐỐI CHIẾU SHELL VỚI TẦNG SYSCALL

TRONG GUEST

```
$ ./exitcode 42 ; echo "bash thay: $?"
$ ./exitcode 256 ; echo "bash thay: $?"
$ ./exitcode -1 ; echo "bash thay: $?"

# chet vi signal, nhìn tu shell
$ sleep 100 & kill -9 %1 ; wait ; echo "bash thay: $?"
$ kill -l 137          # bash dịch ngược số >128 thành tên signal

# core dump có được ghi không, và ghi ở đâu
$ cat /proc/sys/kernel/core_pattern ; ulimit -c
$ coredumpctl list --no-pager | tail -3

# tang syscall nơi that: si_code CLD_EXITED hay CLD_KILLED
$ strace -f -e trace=wait4,waitid ./reaper
```

Nghiem thu Block C

- ☐ Bảng bốn dòng trong note: đường thoát → status hex → macro đúng → giá trị → `$?` của bash.
- ☐ Ít nhất một ô bạn **đoán sai** trước khi chạy, kèm một câu về việc mình đã hiểu nhầm chỗ nào. Đoán đúng cả bốn nghĩa là bài quá dễ, hãy thêm một trường hợp lạ hơn.
- ☐ Một câu trả lời cho "vì sao `$? = 137` không đủ để kết luận điều gì đã xảy ra".

MỤC 04

Bẫy đã biết

Đừng thử fork bomb, kể cả trong guest, khi chưa đặt trần

Cơ chế: giới hạn bạn dùng phải *không* phải `pid_max` toàn hệ thống. Trong một hệ dùng systemd, mỗi unit có `TasksMax` — ánh xạ thẳng vào `pids.max` của cgroup v2. Vượt trần thì `fork` trả về `EAGAIN`, không phải `ENOMEM`. Nhưng phiên ssh của bạn cũng nằm trong một unit, nên cạn task nghĩa là mất luôn khả năng gõ lệnh để dọn.

Cách làm đúng: nếu muốn thử, chạy trong một scope có trần đặt sẵn thấp hơn nhiều, để chỉ scope đó chết. Đây cũng là bài tập nhỏ rất tốt cho cgroup v2 ở Phase 2.

`pid_max` trên máy bạn gần như chắc chắn không phải 32768

Cơ chế: hằng số mặc định trong kernel đúng là `0x8000` = 32768, trần trên 64-bit là 2^{22} = 4194304. Nhưng có hai thứ chỉnh nó lên: kernel tự nâng theo số CPU ngay lúc boot, và systemd từ bản 243 đặt thẳng `kernel.pid_max = 4194304` qua một file sysctl, với lý do ghi

trong `NEWS`: làm cho va chạm PID khó xảy ra hơn.

Cách phân biệt: đọc con số thật trên máy, rồi `sysctl -a --pattern pid_max` và tìm file nào đặt nó. Con số trong bài blog viết năm 2019 không nói gì về máy của bạn.

Đừng cắt trường của `/proc/<pid>/stat`

Triệu chứng: script đọc trạng thái tiến trình chạy đúng hàng tháng rồi một hôm cho kết quả vô nghĩa.

Cơ chế: trường thứ hai của file này là `comm`, nằm trong ngoặc đơn và được in **không escape** – nó chứa được cả dấu cách lẫn dấu `)`. Trong khi đó `/proc/<pid>/status` lại in tên có escape. Cùng một cái tên, hai cách in khác nhau, ở hai file cạnh nhau.

Cách sửa: đọc `status` (định dạng key-value, không phụ thuộc thứ tự), hoặc nếu buộc phải dùng `stat` thì cắt từ dấu `)` cuối cùng rồi mới tách trường.

`ps aux | grep defunct` là thói quen dở

Ba lý do: nó khớp cả chính tiến trình `grep`; chuỗi `defunct` nằm ở cột dòng lệnh nên bất kỳ lệnh nào chứa chữ đó cũng khớp; và nó phụ thuộc *chuỗi hiển thị* của `ps` chứ không phải trạng thái trong kernel. Lọc theo cột `stat` với mô neo đầu chuỗi mới là thứ dùng lại được.

Zombie không làm tăng load average

Load 1/5/15 phút chỉ đếm tiến trình ở trạng thái `R` hoặc `D`; zombie không ở trạng thái nào trong hai cái đó. Thứ có nhúc nhích là **mẫu số** của trường thứ tư trong `/proc/loadavg` – tổng số thực thể đang tồn tại. Nếu ai đó báo "zombie làm máy chậm", con số cần nhìn không phải load average.

Bảng tra nhanh

TRIỆU CHỨNG	KIỂM CÁI NÀY TRƯỚC
Một dòng chữ in ra hai lần sau khi fork	stdout đang nối vào đầu. Terminal thì line-buffered, file/pipe thì fully buffered – có <code>\n</code> cũng không cứu. <code>fflush(NULL)</code> trước fork.
Chạy tay đúng, chạy qua <code> tee</code> hoặc <code>> log</code> thì sai	Cùng nguyên nhân trên. Đây là dạng bug CI kinh điển, đáng nhớ mặt.
Code sau exec vẫn chạy	exec đã thất bại . Nó không bao giờ trả về khi thành công – kiểm <code>errno</code> ngay dòng sau.
strace không thấy fork	Bình thường: glibc gọi <code>clone</code> . Thêm <code>clone</code> và <code>clone3</code> vào <code>-e trace=</code> .
<code>kill -9</code> trả về 0 mà zombie vẫn còn	Đúng như thiết kế. Đi tìm cha của nó (PPid) – vấn đề nằm ở chỗ cha không gọi <code>wait</code> .
Zombie biến mất mà không ai gọi <code>wait</code>	Cha vừa chết ⇒ zombie được chuyển cho reaper, và reaper dọn ngay. Kiểm PPid của cha trước đó.
<code>waitpid</code> trả -1 với <code>ECHILD</code> ngay lập tức	Không còn con nào để chờ. Hai khả năng: chờ nhầm PID, hoặc đã đặt <code>SIGCHLD</code> thành <code>SIG_IGN</code> nên kernel tự dọn hết.

TRIỆU CHỨNG	KIỂM CÁI NÀY TRƯỚC
PPid của orphan không phải 1	Không phải lỗi. Có subreaper trên chuỗi tổ tiên — tra xem cha mới là chương trình gì.
fork trả EAGAIN	Bốn nghi phạm theo thứ tự: <code>ulimit -u</code> , <code>TasksMax</code> của unit, <code>pid_max</code> , <code>threads-max</code> .
<code>pstree: command not found</code>	Gói <code>psmisc</code> , không phải <code>procps</code> . Sửa ở <code>lab-reset.sh</code> , đừng cài tay.

MỤC 05

Nghiem thu & câu hỏi cuối phiên

Definition of Done

- ☐ Trong note có một dòng `ps` thật với `STAT = Z`, kèm PID và PPID.
- ☐ Nêu được ít nhất hai dòng có ở `/proc/<cha>/status` mà `/proc/<zombie>/status` không có.
- ☐ Ghi được exit code của `kill -9` lên zombie và trạng thái sau đó, kèm giải thích.
- ☐ Bản có `waitpid` không để lại zombie và in đúng mã thoát đã chọn.
- ☐ Giải thích được vì sao chạy ra terminal và chạy ra file cho số dòng khác nhau.
- ☐ Hai con số `PPid` khác nhau cho cùng chương trình orphan, kèm **tên** của hai reaper.
- ☐ Bảng bốn đường thoát: hex thô → macro → giá trị → `$?`.
- ☐ Ghi được ba cặp "tên hàm C ↔ tên syscall" quan sát bằng `strace`.
- ☐ Một dòng trong note trả lời thử câu của mốc M-L1: *vì sao tiến trình trong container thấy PID 1 là chính nó?* Chưa cần đúng — ghi phiên bản hôm nay để Phiên 8 so lại.

Hết giờ mà chưa xong thì dừng đúng chỗ đang dừng. Thứ tự ưu tiên nếu phải cắt: Block A không cắt được (mọi phiên sau dựa vào nó), Block B cắt phần subreaper và giữ phần quan sát, Block C có thể lùi sang đầu Phiên 4.

Câu hỏi kiểm tra — đóng terminal, đóng trang này

- ① Một tiến trình đang ở trạng thái `Z` giữ những tài nguyên gì, và đã trả lại những gì? Nêu một cách *quan sát được* để chứng minh câu trả lời của bạn.

■ Ý CẦN CÓ

Đã trả lại: address space (nên RSS về 0), bảng file descriptor, fs context, kernel stack. Còn giữ: cấu trúc mô tả tác vụ trong kernel, mã thoát và thông tin rusage, và **một số PID** — cỡ vài KB bộ nhớ kernel, không phải RAM của chương trình. Cách chứng minh: `/proc/<pid>/status` của zombie *không có dòng* nào bắt đầu bằng `Vm` hay `Rss` (không tồn tại dòng, chứ không phải bằng 0), và `/proc/<pid>/fd/` rỗng. Điểm ăn thêm nếu nói được hệ quả thật của việc tích tụ là cạn PID chứ không phải cạn RAM.

- ② `$?` bằng 137. Có mấy khả năng đã xảy ra, và phân biệt chúng bằng cách nào?

■ Ý CẦN CÓ

Ít nhất hai: chương trình tự thoát với mã 137, hoặc nó bị giết bằng signal số 9 và shell quy về $128+9$. Ở tầng shell **không phân biệt được** — thông tin đã mất khi bash làm phẳng status thành một số. Ở tầng syscall thì phân biệt dễ: hai giá trị `status` nằm ở hai vùng bit khác hẳn nhau, và macro kiểm "thoát bình thường" với macro kiểm "chết vì signal" cho kết quả ngược nhau. Điểm ăn thêm: nói được rằng phép cộng 128 là quy ước của shell chứ không có trong kernel, và rằng POSIX chỉ yêu cầu "lớn hơn 128" nên shell khác có thể chọn số khác.

- ③ Cùng một chương trình orphan cho `PPid = 1` khi chạy qua ssh vào VM lab, nhưng ra một số khác khi chạy trên máy desktop. Cả hai đều đúng — vì sao?

■ Ý CẦN CÓ

Luật reparent không phải "về PID 1" mà là ba bước: thread còn sống trong cùng thread group → tổ tiên gần nhất có bật cờ subreaper (cùng PID namespace) → init của PID namespace. Qua ssh, chuỗi cha là shell → sshd → PID 1, không có subreaper nào trên chuỗi đó, nên rơi xuống bước ba. Trên desktop, ứng dụng thường được `systemd --user` khởi chạy nên user manager là tổ tiên và nó có bật cờ → dừng ở bước hai. Điểm ăn thêm nếu nêu được rằng điều quyết định là **chuỗi tiến trình cha**, không phải "đang ở session của ai", và rằng trong container câu trả lời là PID 1 của namespace đó.

- ④ `strace` một chương trình gọi `fork()` lại hiện ra `clone`. Vì sao, và điều đó dạy gì về cách đọc tài liệu?

■ Ý CẦN CÓ

Từ glibc 2.3.3, hàm thư viện `fork()` gọi `clone` chứ không gọi syscall cùng tên; man page nói tương đương với `clone` chỉ với `SIGCHLD`, còn cờ thêm mà `strace` hiện là sổ sách nội bộ của thư viện thread. Trên arm64 và riscv thậm chí *không tồn tại* syscall `fork`. Bài học: "tên hàm C" và "tên syscall" là hai tầng khác nhau – man section 2 mô tả giao diện, không hứa về cách hiện thực; muốn biết cái gì thật sự chạy thì hỏi `strace`.

- ⑤ Chương trình in một dòng bằng `printf` rồi `fork()`. Chạy ra terminal thấy một dòng, chạy `> file` thấy hai. Cơ chế là gì, và có mấy cách sửa?

■ Ý CẦN CÓ

Buffer stdio nằm trong bộ nhớ user-space nên bị nhân bản nguyên trạng khi fork, và cả hai tiến trình đều flush bản của mình lúc thoát. Ra terminal thì stream là line-buffered nên `\n` đã đẩy nội dung đi trước khi fork; ra file thì fully buffered nên nội dung còn nằm lại. Hai cách sửa khác nhau về bản chất: `fflush(NULL)` trước fork làm sạch buffer trước khi nhân đôi; dùng `_exit` trong con thì con không chạy `atexit` và không flush. Điểm ăn thêm nếu chỉ ra rằng "nhớ thêm `\n`" là lời khuyên chỉ đúng một nửa.

Stretch — chỉ khi dư giờ, không tính vào DoD

- **Bỏ hẳn macro bit:** dùng `waitid` và đọc trường `si_code` – `CLD_EXITED` / `CLD_KILLED` / `CLD_DUMPED` tách bạch sẵn, không phải giải mã gì. So với `waitpid` và tự trả lời: vì sao API cũ vẫn phổ biến hơn?
- **Chứng minh SIGCHLD không xếp hàng:** cài handler đếm số lần nó chạy, fork ra hai chục con cùng chết gần nhau. Số lần handler chạy sẽ *nhỏ hơn* số con. Đây là lý do handler bắt buộc phải có vòng lặp `waitpid` với cờ không chặn.
- **Kernel tự dọn hộ:** đặt `SIGCHLD` thành `SIG_IGN` *tường minh* rồi fork. Không có zombie nào – nhưng `waitpid` sau đó trả `ECHILD`. Chú ý sự khác biệt tinh tế: mặc định của `SIGCHLD` vốn đã là "ignore", nhưng mặc định thì vẫn sinh zombie. Hai chữ "ignore" cùng tên, hai ngữ nghĩa khác nhau.
- **Tránh race tái dùng PID:** `pidfd_open` (Linux 5.3) rồi `waitid` với `P_PIDFD` (Linux 5.4). Tự trả lời: race đó xảy ra chính xác vào lúc nào, và vì sao một file descriptor thì miễn nhiễm.
- **Đo tốc độ fork:** lấy vi phân trường `processes` của `/proc/stat` theo thời gian, rồi so giữa lúc máy rảnh và lúc chạy một vòng lặp shell. Con số này sẽ có ích lại ở Phase 6.
- **Double-fork:** viết bản daemon kiểu cũ (fork, cha thoát, con `setsid`, fork lần nữa) và tự trả lời vì sao cần *hai* lần fork. Đây là cầu nối sang Phiên 6, và cũng là lý do `Type=forking` của systemd bị chính tài liệu của nó khuyên tránh.

Nguồn & đọc thêm

Đọc trong hoặc ngay sau phiên

MAN PAGE

fork(2) <https://man7.org/linux/man-pages/man2/fork.2.html>

Đọc danh sách "exact duplicate ... except for the following points" và mục NOTES về copy-on-write. Hai đoạn này là toàn bộ nền của Block A.

MAN PAGE

wait(2), waitpid(2), waitid(2) <https://man7.org/linux/man-pages/man2/wait.2.html>

Bốn nghĩa của tham số pid, bảng macro WIF*, và đoạn NOTES định nghĩa zombie. Trang chính của cả phiên.

MAN PAGE

execve(2) <https://man7.org/linux/man-pages/man2/execve.2.html>

Mục "All process attributes are preserved during an execve(), except the following" và đoạn NOTES nói vì sao "exec tạo tiến trình mới" là mô tả sai.

MAN PAGE

exit(3) <https://man7.org/linux/man-pages/man3/exit.3.html>

Ba việc exit(3) làm thêm so với _exit(2): atexit handler, flush stdio, xóa tmpfile.

MAN PAGE

PR_SET_CHILD_SUBREAPER(2const) https://man7.org/linux/man-pages/man2/PR_SET_CHILD_SUBREAPER.2const.html

Trang phát biểu luật reparent thật: "reparented to the nearest still living ancestor subreaper". Có từ Linux 3.4.

MAN PAGE

proc_pid_status(5) https://man7.org/linux/man-pages/man5/proc_pid_status.5.html

Định dạng key-value của /proc/PID/status. Đây là đường đọc PPid an toàn, khác hẳn việc cắt trường của stat.

MAN PAGE

setbuf(3) / setvbuf(3) <https://man7.org/linux/man-pages/man3/setbuf.3.html>

Luật buffering: file thường thì block-buffered, terminal thì line-buffered, stderr không buffer. Gốc của bẫy in hai lần.

MAN PAGE

strace(1) <https://man7.org/linux/man-pages/man1/strace.1.html>

Đọc -f (theo con), -e trace=, và -T. Ba cờ này đủ cho toàn bộ Phase 1.

TÀI LIỆU CHÍNH CHỦ

GNU Bash Manual – Exit Status https://www.gnu.org/software/bash/manual/html_node/Exit-Status.html

Câu "Bash uses the value 128+N as the exit status" – chủ ngữ là Bash. Đọc cả 126 và 127.

POSIX – wait, waitpid (RATIONALE) <https://pubs.opengroup.org/onlinepubs/9799919799/functions/waitpid.html>

Mục RATIONALE nói rõ vì sao POSIX cố ý KHÔNG chuẩn hoá bit layout của status. Bằng chứng cho việc phải dùng macro.

systemd-run(1) <https://www.freedesktop.org/software/systemd/man/latest/systemd-run.html>

Đọc --scope và --user: cách rẻ nhất để có một tổ tiên là subreaper mà không phải viết code.

The Linux Programming Interface – ch. 24–26 <https://man7.org/tlpi/>

Ch. 24 process creation, 25 process termination, 26 monitoring child processes. Đọc theo chương, không đọc tuyến tính.

Tra khi cần, không đọc tuyến tính

proc_loadavg(5) https://man7.org/linux/man-pages/man5/proc_loadavg.5.html

Ba số đầu chỉ đếm tiến trình ở trạng thái R và D. Trường thứ tư mới là nơi zombie xuất hiện – ở mẫu số.

exec(3) <https://man7.org/linux/man-pages/man3/exec.3.html>

Trang tự tổ chức theo chữ cái l/v/p/e – đọc một lần là hết phải đoán tên hàm nào nhận mảng, hàm nào tra PATH.

_exit(2) https://man7.org/linux/man-pages/man2/_exit.2.html

Mục "C library/kernel differences": từ glibc 2.3 wrapper gọi `exit_group(2)`, còn syscall `_exit` thô chỉ giết thread gọi nó.

exit_group(2) https://man7.org/linux/man-pages/man2/exit_group.2.html

Syscall mà strace sẽ hiện thay cho `exit()`. Ngắn, đọc hết trong một phút.

ps(1) <https://man7.org/linux/man-pages/man1/ps.1.html>

Mục PROCESS STATE CODES (D I R S t W X Z + modifier) và giải thích chuỗi <defunct> nằm ở specifier args.

proc_pid_stat(5) https://man7.org/linux/man-pages/man5/proc_pid_stat.5.html

Bảng ký tự trạng thái theo từng kernel version (l có từ 4.14) và thứ tự 52 trường. Xem cả cảnh báo về trường comm.

proc_stat(5) https://man7.org/linux/man-pages/man5/proc_stat.5.html

Trường processes = số lần fork từ lúc boot. Vì phân nó theo thời gian là cách đo tốc độ fork rẻ nhất.

MAN PAGE

proc_sys_kernel(5) https://man7.org/linux/man-pages/man5/proc_sys_kernel.5.html

Mục pid_max: mặc định 32768, trần 2^{22} trên 64-bit. Nguồn đúng cho các con số này, không phải sysctl.rst.

MAN PAGE

signal(7) <https://man7.org/linux/man-pages/man7/signal.7.html>

Bảng standard signal (SIGCHLD mặc định lgn) và mục Real-time signals nói standard signal không xếp hàng.

MAN PAGE

credentials(7) <https://man7.org/linux/man-pages/man7/credentials.7.html>

Nơi phát biểu gọn nhất "PID/PPID/session ID được giữ qua execve" – dùng để đối chiếu bảng fork-vs-exec.

MAN PAGE

prctl(2) <https://man7.org/linux/man-pages/man2/prctl.2.html>

Trang mục lục của các operation; nội dung từng cờ đã tách sang trang .2const riêng.

TÀI LIỆU CHÍNH CHỦ

POSIX – exit() <https://pubs.opengroup.org/onlinepubs/9799919799/functions/exit.html>

Câu gốc "only the least significant 8 bits (that is, status & 0377) shall be available" – nguồn của việc exit code bị cắt.

TÀI LIỆU CHÍNH CHỦ

POSIX – Shell Command Language, 2.8.2 Exit Status https://pubs.opengroup.org/onlinepubs/9799919799/utilities/V3_chap02.html

POSIX chỉ bắt buộc "greater than 128" cho lệnh chết vì signal – không chốt đúng $128+N$. Đó là chỗ bash tự chọn.

SOURCE / PACKAGING

glibc – bits/waitstatus.h <https://raw.githubusercontent.com/bminor/glibc/master/bits/waitstatus.h>

Định nghĩa thật của WEXITSTATUS/WTERMSIG/WCOREDUMP. Xem WIFSIGNALED để hiểu vì sao nó không chỉ là "WTERMSIG khác 0".

SOURCE / PACKAGING

Linux – kernel/exit.c <https://github.com/torvalds/linux/blob/v6.12/kernel/exit.c>

Đọc do_exit() theo thứ tự dòng, và comment ba bước của find_new_reaper(). Đây là nguồn gốc của luật reparent.

SOURCE / PACKAGING

Linux – kernel/signal.c <https://github.com/torvalds/linux/blob/v6.12/kernel/signal.c>

prepare_signal() vút signal khi SIGNAL_GROUP_EXIT – bằng chứng vì sao kill -9 lên zombie không làm gì mà vẫn trả 0.

SOURCE / PACKAGING

Linux – include/linux/threads.h <https://github.com/torvalds/linux/blob/v6.12/include/linux/threads.h>

PID_MAX_DEFAULT 0×8000 và PID_MAX_LIMIT $4 \times 1024 \times 1024$. Hai hằng số đứng sau mọi con số pid_max bạn đọc được.

SOURCE / PACKAGING

systemd – NEWS <https://github.com/systemd/systemd/blob/main/NEWS>

Tra "subreaper" (CHANGES WITH 188) và "pid_max" (CHANGES WITH 243). Hai thay đổi này làm sai lệch phần lớn bài blog cũ.

SOURCE / PACKAGING

systemd – src/basic/process-util.c <https://github.com/systemd/systemd/blob/main/src/basic/process-util.c>

Hàm `make_reaper_process()`: PID 1 thoát sớm không gọi `prctl`, chỉ manager không phải PID 1 mới thật sự bật subreaper.

Để dành cho các phiên sau

MAN PAGE

systemd.kill(5) <https://www.freedesktop.org/software/systemd/man/latest/systemd.kill.html>

KillMode= mặc định là control-group: systemd dọn theo cgroup, không theo quan hệ cha-con. Đọc cả cảnh báo về KillMode=process.

MAN PAGE

clone(2) <https://man7.org/linux/man-pages/man2/clone.2.html>

Termination signal nằm ở byte thấp của flags; CLONE_PARENT cho thấy "ai là cha" là thứ chọn được lúc sinh. Nền cho Phase 4.

MAN PAGE

pid_namespaces(7) https://man7.org/linux/man-pages/man7/pid_namespaces.7.html

Init của namespace nhận orphan; signal mặc định không giết được nó; init chết thì cả namespace bị SIGKILL. Đọc kỹ ở Phase 4.

MAN PAGE

sigaction(2) <https://man7.org/linux/man-pages/man2/sigaction.2.html>

SA_NOCLDWAIT, SA_NOCLDSTOP, và mục HISTORY: POSIX.1-1990 từng CẤM đặt SIG_IGN cho SIGCHLD.

MAN PAGE

vfork(2) <https://man7.org/linux/man-pages/man2/vfork.2.html>

Đọc mục CAVEATS và VERSIONS: vì sao vfork vẫn tồn tại sau khi đã có copy-on-write, và vì sao POSIX.1-2008 bỏ nó.

MAN PAGE

pidfd_open(2) https://man7.org/linux/man-pages/man2/pidfd_open.2.html

Cách hiện đại để tham chiếu một tiến trình mà không dính race tái dùng PID. Linux 5.3; waitid(P_PIDFD) từ 5.4.

MAN PAGE

signal-safety(7) <https://man7.org/linux/man-pages/man7/signal-safety.7.html>

Danh sách hàm gọi được trong signal handler – và trong tiến trình con sau fork của chương trình đa luồng.

TÀI LIỆU CHÍNH CHỦ

GNU C Library – Merged Signals https://sourceware.org/glibc/manual/latest/html_node/Merged-Signals.html

Vì sao không đếm được số con đã chết bằng số lần handler chạy, và mẫu vòng lặp waitpid trong handler SIGCHLD.

KERNEL DOC

Control Group v2 – admin guide <https://docs.kernel.org/admin-guide/cgroup-v2.html>

Mục PID: pids.max, pids.current, pids.events, và câu "fork() will return -EAGAIN". Sẽ đọc kỹ ở Phase 2 và Phase 4.

MAN PAGE

systemd.resource-control(5) <https://www.freedesktop.org/software/systemd/man/latest/systemd.resource-control.html>

TasksMax= ánh xạ thẳng vào pids.max của cgroup v2. Đây là trần bạn đặt trước pid_max.

TÀI LIỆU CHÍNH CHỦ

tini – init tối giản cho container <https://github.com/kraallin/tini>

README nói đúng hai việc PID 1 phải làm: reap zombie và chuyển tiếp signal. Chính là binary sau docker run --init.

TÀI LIỆU CHÍNH CHỦ

docker run – tùy chọn --init <https://docs.docker.com/reference/cli/docker/container/run/>

Một dòng mô tả: "forwards signals and reaps processes". Hai vế đó là hai vấn đề độc lập, xem mục Bấy.

Ba mẫu đáng nhớ

VÌ SAO VẪN CẦN VFORK SAU KHI ĐÃ CÓ COPY-ON-WRITE

Nếu COW làm fork gần như miễn phí thì `vfork` đã phải biến mất từ lâu. Nó không biến mất, và man page nói thẳng lý do: `vfork` tạo tiến trình mới "without copying the page tables of the parent process". Tức là chi phí còn lại sau COW – nhân bản bảng phân trang – đủ lớn để đáng né. Có một hệ quả nữa ít người nhắc: với chính sách overcommit chặt, một tiến trình 8 GiB muốn fork ra để chạy một helper 2 MB vẫn phải *cam kết* 8 GiB bộ nhớ trong khoảnh khắc đó, hoặc chấp nhận rủi ro gặp OOM killer. Đó là lý do `posix_spawn` của glibc, từ bản 2.24, cài đặt bằng `clone` với cờ `CLONE_VM|CLONE_VFORK` chứ không dùng `fork`. Nghịch lý dễ chịu: hàm "tạo tiến trình" chuẩn mực nhất lại là hàm không gọi `fork`.

MƯỜI GIÂY CỦA DOCKER STOP ĐẾN TỪ ĐÂU

Tiến trình PID 1 trong một PID namespace được kernel đánh dấu là không giết được: signal có hành động mặc định sẽ bị bỏ qua nếu nó *chưa cài handler* cho signal đó. Một ứng dụng bình thường chạy làm PID 1 trong container vì thế **không chết khi nhận `SIGTERM`** – nó không cài handler, và mặc định "chấm dứt" bị vô hiệu. Docker chờ hết thời gian ân hạn rồi mới gửi `SIGKILL` từ namespace ngoài, thứ duy nhất xuyên qua được. Đó là vế thứ hai, độc lập với chuyện zombie, và cả hai gói gọn trong đúng một dòng mô tả của `--init`: "forwards signals and reaps processes". Một tùy chọn, hai vấn đề chẳng liên quan gì nhau.

CON SỐ 32768 VÀ CÁI CHẾT CHẬM CỦA NÓ

`PID_MAX_DEFAULT` trong kernel là `0x8000` – 32768, di sản của thời PID là số 16 bit có dấu. Trên hệ 64-bit là 2^{22} , và comment ngay cạnh hằng số đó viết bằng giọng rất kỹ sư: “A maximum of 4 million PIDs should be enough for a while.” Rồi hai thứ lần lượt ăn mòn con số mặc định: kernel tự nâng nó theo số CPU ngay lúc boot, và systemd 243 đặt thẳng nó lên 4194304 cho toàn hệ. Kết quả là “mặc định 32768” – câu xuất hiện trong hầu hết bài viết về giới hạn tiến trình trên Linux – đúng về mặt hằng số trong source và sai về mặt con số trên máy bạn. Một ví dụ sạch sẽ cho thói quen nên giữ suốt lộ trình này: hằng số mặc định và giá trị đang chạy là hai thứ khác nhau, và chỉ có cái thứ hai mới trả lời được câu hỏi của bạn.

Mọi khẳng định kỹ thuật ở trên đã đối chiếu tài liệu và source upstream – Linux man-pages, POSIX, kernel source (tag v6.12), glibc, GNU bash, systemd – tính đến 22/08/2026. Năm chỗ lệch với tài liệu phổ biến (zombie chiếm RAM, `kill -9` dọn được zombie, orphan luôn về PID 1, `\n` ngăn được việc in hai lần, `pid_max` mặc định 32768) được đánh dấu riêng kèm bằng chứng. Riêng chuyện `kill -9` không có man page nào phát biểu thẳng – bằng chứng lấy từ kernel source và phải tự kiểm bằng thực nghiệm trong phiên.