

Lecture 8.1

Processes

fork, exec, exit and wait

Md Mahbub E Noor

Assistant Professor, Department of CSE
University of Barishal

```
./fork1 (real)
before fork: I am PID 199
child : fork returned 0, my PID 200,
        parent 199, x = 15
parent: fork returned 200, my PID 199,
        x = 10
# one call, two returns
```

What you will learn today



Say what a process is, and what the kernel stores about it in `task_struct`.



Name the process states and make a process enter each one.



Use `fork`, `exec`, `exit` and `wait`, and explain zombies and orphans.



Explain how threads differ from processes, in the kernel's terms.

Today's plan (60 min)

0-5

Recap of 7.2

5-15

What a process is

15-22

States

22-42

`fork`, `exec`, `exit`, `wait`

42-52

Threads

52-60

Checkpoint

Lab 8.1: `mysh.c`, a small shell.

Quick recap of Lecture 7.2

Q1

Which function makes PID 1, and how?

`rest_init()` calls `user_mode_thread(kernel_init, ...)`; later `kernel_execve("/init")` turns it into our program.

Q2

What did our init do to run kinfo?

`fork()`, then `execv("/bin/kinfo")` in the child, then `waitpid()` in the parent. Today: why each step is needed.

Q3

Why must PID 1 never return?

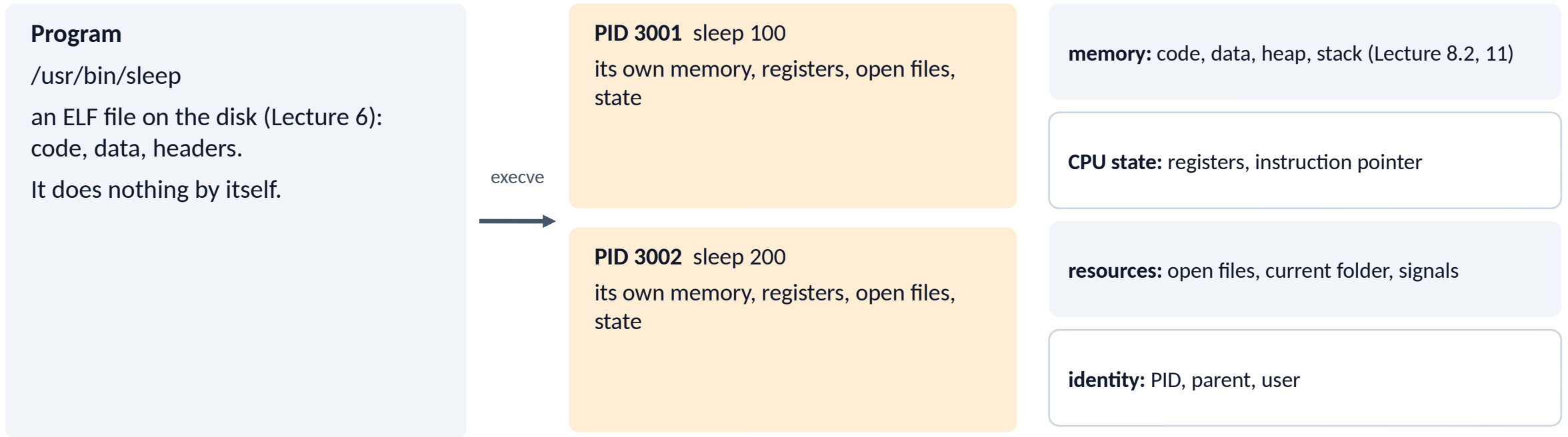
The kernel panics: "Attempted to kill init!". Today: PID 1 also adopts orphans.

PART 1

What a process is

A program in execution, and the struct the kernel keeps for it.

A program is a file; a process is a program running



SAME AS C

Like a function and its calls

One function, many calls: each call has its own local variables. One program, many processes: each has its own memory.

DIFFERENT FROM C

But fully separated

Two calls of a function share global variables. Two processes share nothing unless they ask the kernel (Lecture 10).

Book: Love, LKD Ch. 3 (the process); Bovet & Cesati Ch. 3

task_struct: everything the kernel knows about one task

include/linux/sched.h, Linux 6.8 (real lines)

```
748 struct task_struct {
756     unsigned int    __state;    /* R, S, D, T ... */
767     void            *stack;     /* kernel stack */
797     struct sched_entity se;     /* scheduler data */
880     struct mm_struct *mm;       /* the memory */
884     int             exit_state;
885     int             exit_code;
975     pid_t           pid;
976     pid_t           tgid;       /* process ID */
989     struct task_struct *real_parent;
997     struct list_head children;
1085    char             comm[16];   /* name */
1101    struct files_struct *files;   /* open files */
1565 };
```

818 lines

of C between the braces

10 176 bytes

per task in Ubuntu's 6.8 kernel
(slabinfo)

pointers

most parts live in other structs: mm,
files, signal

C link: a struct full of pointers to other structs, like a linked data structure from your data-structures course.

Process or task? The kernel calls every running thing a **task**. pid is the task's own ID; tgid (thread group ID) is what user space calls the process ID: getpid() returns tgid.

Book: Love, LKD Ch. 3 (process descriptor and the task structure)

PART 2

Process states

Running, waiting, stopped, and the strange state of the dead.

Five states you can produce and see (states.c, real)

```
./states (ps -o pid,ppid,stat,wchan,comm)

PID  PPID  STAT  WCHAN          COMMAND
270   254   S     do_wait        states # parent, in system()
271   270   R     -              states # for (;;) ;
272   270   S     do_sys_pause   states # pause()
273   270   T     do_signal_stop states # SIGSTOP
274   270   Z     -              states # exit, no wait
```

WCHAN

the kernel function where the task is waiting: `do_wait` is inside `wait()`, `do_sys_pause` inside `pause()`. Real kernel names, seen from user space.

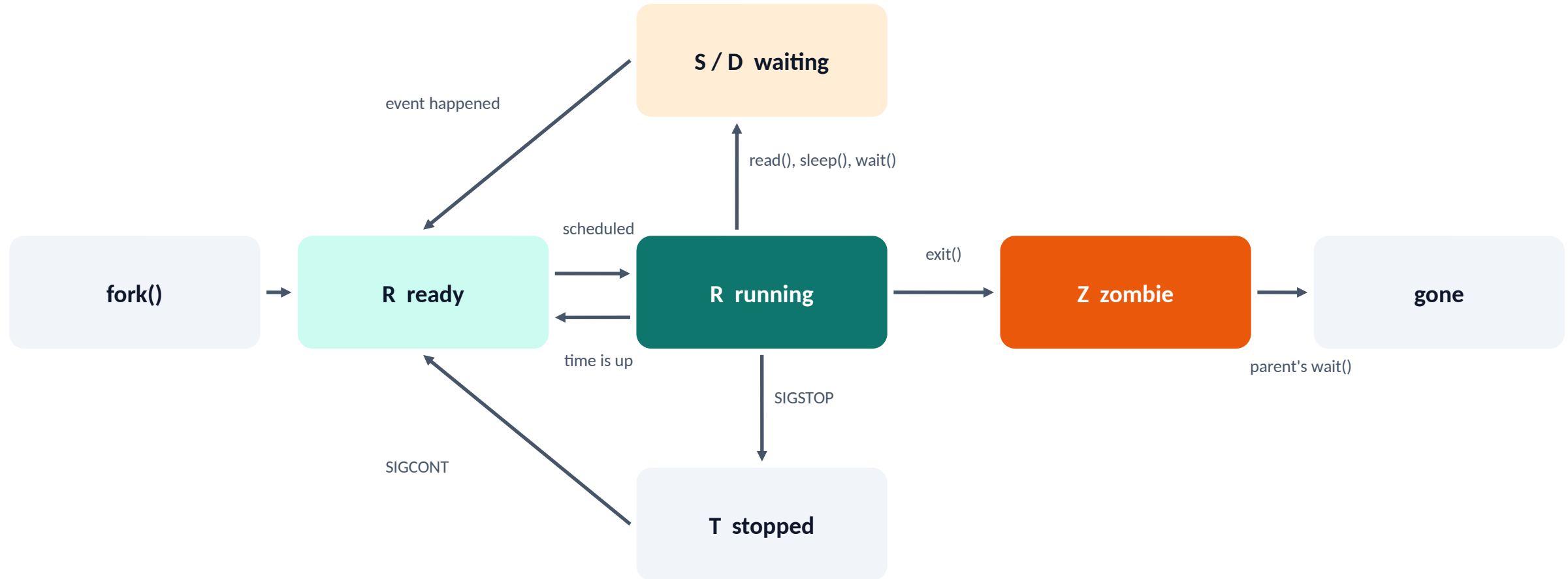
R does not mean running

R means "wants a CPU". With more R tasks than CPUs, some wait in the run queue (Lecture 8.2).

ps	Kernel name (sched.h)	Meaning
R	TASK_RUNNING 0x0	running, or ready and waiting for a CPU
S	TASK_INTERRUPTIBLE 0x1	sleeping until an event or a signal
D	TASK_UNINTERRUPTIBLE 0x2	sleeping, signals ignored (usually disk I/O)
T	__TASK_STOPPED 0x4	stopped by SIGSTOP or Ctrl-Z
Z	EXIT_ZOMBIE 0x20	finished; exit status not yet collected
I	TASK_IDLE	an idle kernel thread (Lecture 7.1's kworkers)

Book: Love, LKD Ch. 3 (process state)

How a task moves between states



Only the scheduler moves a task between **ready** and **running**; the task itself only asks to wait, to stop, or to exit.

PART 3

fork, exec, exit and wait

Four system calls make the life of every process.

fork(): one call, two returns

```
fork1.c

int x = 10;
printf("before fork: I am PID %d\n", getpid());
fflush(stdout);
pid_t pid = fork();
if (pid == 0) {                /* child */
    x = x + 5;
    printf("child : fork returned %d ...", pid, ..., x);
    return 0;
}
waitpid(pid, NULL, 0);        /* parent */
printf("parent: fork returned %d ...", pid, ..., x);
```

```
./fork1 (real)

before fork: I am PID 199
child : fork returned 0, my PID 200,
        parent 199, x = 15
parent: fork returned 200, my PID 199, x = 10
```

Two returns: The child gets 0; the parent gets the child's PID. That is the only way the code can tell them apart.

Two copies of x: The child changed its x to 15; the parent's x is still 10. All memory was copied.

SAME AS C

Like passing by value

The child gets a copy of everything, as a C function gets copies of its arguments.

DIFFERENT FROM C

No C function returns twice

fork does: once in each process. Nothing in C prepares you for that.

Book: Love, LKD Ch. 3 (process creation)

Inside the kernel: fork is copy_process()

kernel/fork.c, Linux 6.8 (real lines, shortened)

```
2984 SYSCALL_DEFINE0(fork)
    {
        struct kernel_clone_args args = {
            .exit_signal = SIGCHLD,
        };
        return kernel_clone(&args);
    }
2861 pid_t kernel_clone(...) {
2902     p = copy_process(NULL, trace, ..., args);
2933     wake_up_new_task(p); /* let it run */
2240 struct task_struct *copy_process(...) {
2327     p = dup_task_struct(current, node);
2485     retval = copy_files(clone_flags, p, ...);
2497     retval = copy_mm(clone_flags, p);
```

1

dup_task_struct: a new task_struct and kernel stack, copied from the parent (current)

2

copy_files: the same open files, so the child writes to the same terminal

3

copy_mm: a copy of the memory (next slides: copy-on-write)

4

wake_up_new_task: state R, into the run queue

current = a pointer to the task that is running now: in fork, the parent.

What fork really copies: a surprise and a cost

```
●●● nobuf.c (real)
printf("hello "); /* still in the buffer */
fork();
printf("world\n");
$ ./nobuf
hello world
hello world
```

"hello " was printed **once** by the program but appears **twice**: it was still in `stdio`'s buffer, inside the process's memory, and `fork` copied the buffer. Fix: `fflush(stdout)` before `fork`.

Memory in use	fork + exit + wait	thread create + join
0 MB	0.14 ms	0.035 ms
100 MB	2.3 ms	0.035 ms
400 MB	6.9 ms	0.035 ms

forkcost.c, average of 2000 runs on our test machine (real CPU, Linux)

Copy-on-write: `fork` does not copy the bytes. Parent and child share every page, marked read-only; a page is copied only when one of them writes to it. But the kernel still copies the **page tables**: more memory, slower `fork`.

C link: `stdout`'s buffer is an ordinary array in your process. Anything in your memory at the moment of `fork` exists twice afterwards: buffers, malloc'd data, half-built strings.

exec: same process, new program

```
● ● ● exec1.c (real)
printf("exec1: my PID is %d, now I exec the shell\n", ...);
execvp("sh", "sh", "-c", "echo \"sh : my PID is $$ ...\"", NULL);
perror("execvp"); /* only if exec failed */
$ ./exec1
exec1: my PID is 240, now I exec the shell
sh : my PID is 240, and exec1 is gone
```

Function	Arguments as	Finds the program	Environment
exec1	a list: "ls", "-l", NULL	full path only	same as now
execv	an array: argv	full path only	same as now
execvp	an array: argv	searches PATH	same as now
execve	an array: argv	full path only	given: envp

Same PID

240 before and after: the process stays, the program changes. Open files stay open too.

Never returns

On success the old code is gone, so there is nothing to return to. The perror line runs only if exec failed.

Only one real call

All six exec functions in libc end in the execve system call (Lecture 8.2).

exit and wait: how a parent learns how its child ended

```
waitst.c (real)

if (fork() == 0) exit(42);           /* normal */
waitpid(p, &st, 0);
if (fork() == 0) { int *q = NULL; return *q; } /* crash */
waitpid(p, &st, 0);
$ ./waitst
child 230 exited, status 42 (raw 0x2a00)
child 231 killed by signal 11 (raw 0x000b)
```



Do not decode by hand: use the macros `WIFEXITED` / `WEXITSTATUS` and `WIFSIGNALED` / `WTERMSIG`.

SAME AS C

return from main = exit

return 42; in main and `exit(42)` do the same: libc's `__libc_start_main` calls exit with main's return value (Lecture 6).

DIFFERENT FROM C

Only 8 bits survive

`exit(256)` gives status 0, `exit(300)` gives 44: only the low byte is kept. The shell shows it as `$?` (Lecture 1: 127, 126, 139 = 128 + 11).

Zombies and orphans

● ● ● zombie.c: child exits, parent sleeps (real)

```
PID  PPID  STAT  COMMAND
247   233  S      zombie
248   247  Z      zombie
after waitpid: status 7
PID  PPID  STAT  COMMAND
247   233  S      zombie      # 248 is gone
```

● ● ● orphan.c: parent exits first (real)

```
parent 261: ending now, child is 262
child  262: my parent is 261
child  262: my parent is now 1
```

Zombie (Z)

The child has finished, but its `task_struct` stays so the parent can read the exit status. A parent that never calls `wait` leaks zombies: each one holds a PID.

Orphan

The parent ended first. The kernel gives the child a new parent: PID 1 (here, and in our QEMU guest). PID 1 calls `wait` for it: one more reason it must never exit.

On an Ubuntu desktop

The new parent may be a "subreaper" instead, such as `systemd --user`. `kernel/exit.c, find_new_reaper()`, checks `is_child_subreaper` before falling back to `init`.

Lab 8.1 tip: a shell that runs programs in the background must collect them later with `waitpid(-1, &st, WNOHANG)`, or they stay zombies.

Watch our shell run ls

A shell runs ls: fork, exec, exit, wait

FORKEXECEXITWAIT

processes (task_structs in the kernel)

PID 410R

program: mysh

our shell from Lab 8.1

the terminal

mysh> ls /proc/self/fd

strace -f -tt (real, times from the clone)

Our shell (PID 410) has read the line "ls /proc/self/fd". It cannot become ls itself, or the shell would be gone. So it makes a new process.

Click the video to play or pause. About 1 minute, no sound.

Watch for

1. After fork, **two** processes run mysh.
2. The parent **sleeps** (S) in wait4.
3. **execvp** tries **PATH** folders one by one.
4. Same PID 411, **new program** ls.
5. A short **zombie** (Z) before wait4 returns.

Md Mahbub E Noor | Lecture 8.1: Processes | University of Barishal

17 / 27

The same run in strace: one command, about 6 ms

```
● ● ● strace -f -tt ./mysh (real, shortened)
410 .730533 clone(child_stack=NULL, flags=...|SIGCHLD) = 411
410 .730669 wait4(411, <unfinished ...>
411 .730758 execve("/usr/local/sbin/ls", ["ls", "/proc/self/fd"], envp) = -1 ENOENT
411 .730837 execve("/usr/local/bin/ls", ...) = -1 ENOENT
411 .730896 execve("/usr/sbin/ls", ...) = -1 ENOENT
411 .730953 execve("/usr/bin/ls", ["ls", "/proc/self/fd"], envp) = 0
411 .735887 write(1, "0\n1\n2\n3\n", 8) = 8
411 .736080 exit_group(0) = ?
410 .736232 <... wait4 resumed>[{WIFEXITED(s) && WEXITSTATUS(s) == 0}], 0, NULL) = 411
410 .736376 --- SIGCHLD {si_signo=SIGCHLD, si_code=CLD_EXITED, si_pid=411, ...} ---
```

clone = fork

glibc's fork() is the clone system call, with SIGCHLD: "tell my parent when I end".

PATH search

execvp is a loop in libc: try each folder of PATH until the kernel says yes.

0 1 2 3

Is inherited fds 0, 1, 2 from the shell (through fork and exec); 3 is the folder it opened itself.

PART 4

Threads

Tasks that share one process: one memory, many flows of control.

Threads: same PID, different TID, shared memory

```
./threads (pthread_create twice, real)

main      : getpid() = 310, gettid() = 310
thread 1:  : getpid() = 310, gettid() = 312
thread 2:  : getpid() = 310, gettid() = 313
  PID  LWP  NLWP  STAT  COMMAND
  310   310    3  Sl    threads
  310   312    3  Sl    threads
  310   313    3  Sl    threads
shared = 2 (both threads changed the same variable)
```

One mechanism: Linux has no separate "thread" object: a thread is a task created with flags that say what to **share** instead of copy.

CLONE_VM: share the memory: copy_mm just points to the parent's mm. That is why shared became 2.

CLONE_THREAD: join the parent's thread group: same tgid, so getpid() gives 310 for all three.

```
strace: the flags of clone (real)

fork()      : clone(flags=CLONE_CHILD_CLEARPID|CLONE_CHILD_SETTID|SIGCHLD)
pthread_create(): clone3(flags=CLONE_VM|CLONE_FS|CLONE_FILES|CLONE_SIGHAND|CLONE_THREAD|CLONE_SYSVSEM|...)
kernel/fork.c 1730: if (clone_flags & CLONE_VM) { mmget(oldmm); mm = oldmm; } else mm = dup_mm(...);
```

Book: Love, LKD Ch. 3 (the Linux implementation of threads)

Process or thread? The trade-offs

	Processes (fork)	Threads (pthread_create)
Memory	copied (copy-on-write)	shared: one mm_struct
Open files	copied table, same files	shared table
IDs	new PID	same PID, new TID
Creating one (our machine)	0.14 ms, more with more memory	0.035 ms, always
A crash (SIGSEGV)	kills only that process	kills the whole process, all threads
Talking to each other	needs the kernel: pipes, sockets (Lecture 10)	just use the same variables (with locks)
Examples	the shell, each command; Chrome's tabs	a web server's workers; a game engine

Rule of thumb: processes for **isolation** (one part may crash or must not see the other's data), threads for **sharing** and speed. Chrome uses a process per site so a crashing page cannot take the browser down.

LAB 8.1

mysh.c: write a small shell

Read a line, fork, exec, wait. Five TODOs.

Lab 8.1 tasks, step by step

- 1 **TODO 1** split the line into argv with strtok; end it with NULL
- 2 **TODO 2** fork; in the child execvp, or print the error and `_exit(127)`
- 3 **TODO 3** waitpid in the parent and report how the child ended
- 4 **TODO 4** cd as a built-in with chdir
- 5 **TODO 5** a trailing &: do not wait; collect later with WNOHANG

Think: why can cd not be a program like ls? (Hint: whose current folder would change?) Answer in your report.

```
● ● ● the finished shell (real session)

mysh> echo hello from a child
hello from a child
mysh> cd /tmp
mysh> pwd
/tmp
mysh> nosuchprog
mysh: nosuchprog: No such file or directory
[369 exited with status 127]
mysh> ./segv
[370 killed by signal 11]
mysh> sleep 1 &
[371 running in the background]
mysh> echo waiting
waiting
[371 done]
```

Checkpoint: test yourself

Q1 fork returned 0. Are you the parent or the child?

→ the child

Q2 After a successful `execvp`, which line of your code runs next?

→ none: your program is gone; the new one starts at its `_start`

Q3 `ps` shows STAT Z. What does the parent have to do?

→ call `wait` / `waitpid` for that child

Q4 status is 0x000b after `waitpid`. What happened?

→ killed by signal 11 (SIGSEGV)

Q5 Which clone flag makes a thread share memory?

→ CLONE_VM

Key ideas from today



A process = a program running: memory, registers, files, an ID, all in `task_struct`.



States R, S, D, T, Z: only the scheduler moves a task between ready and running.



`fork` copies the process and returns twice: 0 in the child, the PID in the parent.



`exec` replaces the program, keeps the PID and open files, and never returns.



`exit` leaves a zombie until the parent waits; orphans are adopted by PID 1 (or a subreaper).



A thread is a task that shares memory (`CLONE_VM`) and the PID (`CLONE_THREAD`).

Homework 8.1 (due before Lecture 8.2)

1. Count

How many lines does this print: `for (i = 0; i < 3; i++) fork(); printf("hi\n");` Draw the process tree, then run it.

2. Same address

Parent and child print `&x` and `x` after the child sets `x = 2`. Predict both lines; explain why the addresses are equal but the values are not.

3. Zombie farm

Write a program that makes 5 zombies and keeps them for 30 s. Show them with `ps`. How do they disappear without a `wait` call?

4. Redirect

Add `> file` to `mysh`: in the child, before `exec`, open the file and use `dup2(fd, 1)`. Test with `echo hello > out.txt`.

5. Measure

Run `forkcost` (demo folder) on the lab PC with 0, 100 and 400 MB. Plot the times and explain the shape.

Next lecture and reading

NEXT: LECTURE 8.2

execve inside the kernel, and scheduling

How the kernel turns an ELF file into a running process (load_elf_binary, the initial stack), and how the scheduler shares the CPU: run queues, nice values, EEVDF.

Main text

R. Love, **Linux Kernel Development**, 3rd ed., Ch. 3 (today) and Ch. 4 (process scheduling) for 8.2.

C side

R. E. Bryant & D. R. O'Hallaron, **CS:APP**, 3rd ed., §8.4 (process control).

Source

Linux 6.8: include/linux/sched.h, kernel/fork.c, kernel/exit.c (Linux v6.8 source online: elixir.bootlin.com/linux/v6.8).

Lab files

lab8_files.zip: mysh_starter.c, the demo programs, stack and shares for Lab 8.2.