

CSE 4105 • SYSTEM PROGRAMMING • SHORT COURSE

Class 7

Processes and Scheduling

fork, exec, exit, wait, and who gets the CPU

Md Mahbub E Noor

Assistant Professor, Department of CSE

University of Barishal

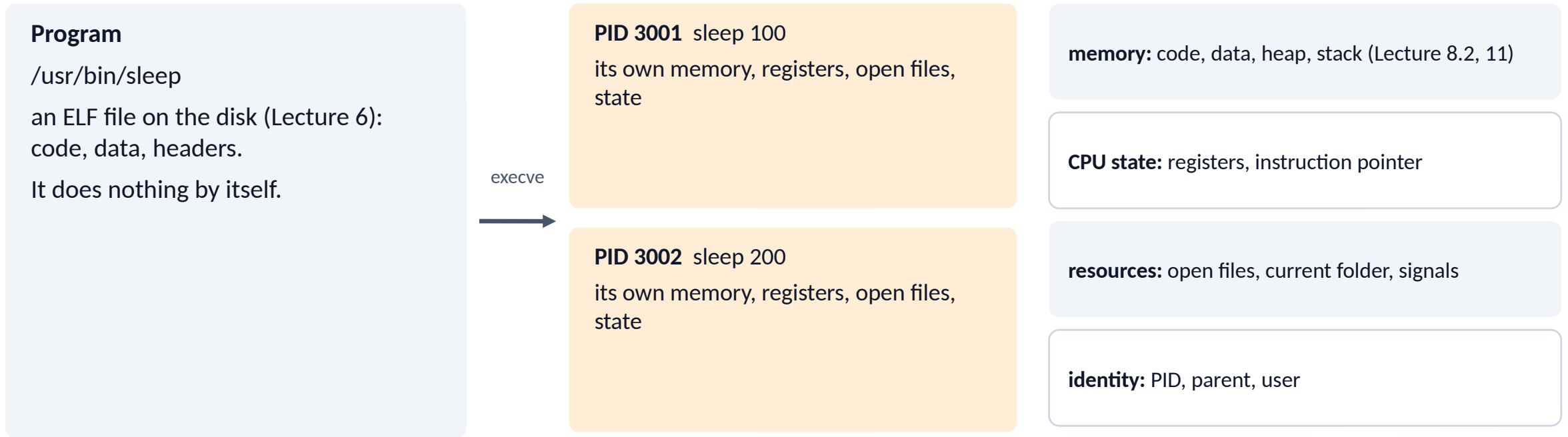
CLASS

7 / 10

17 slides, taken from the full course:

Lecture 8.1, Lecture 8.2

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)

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.

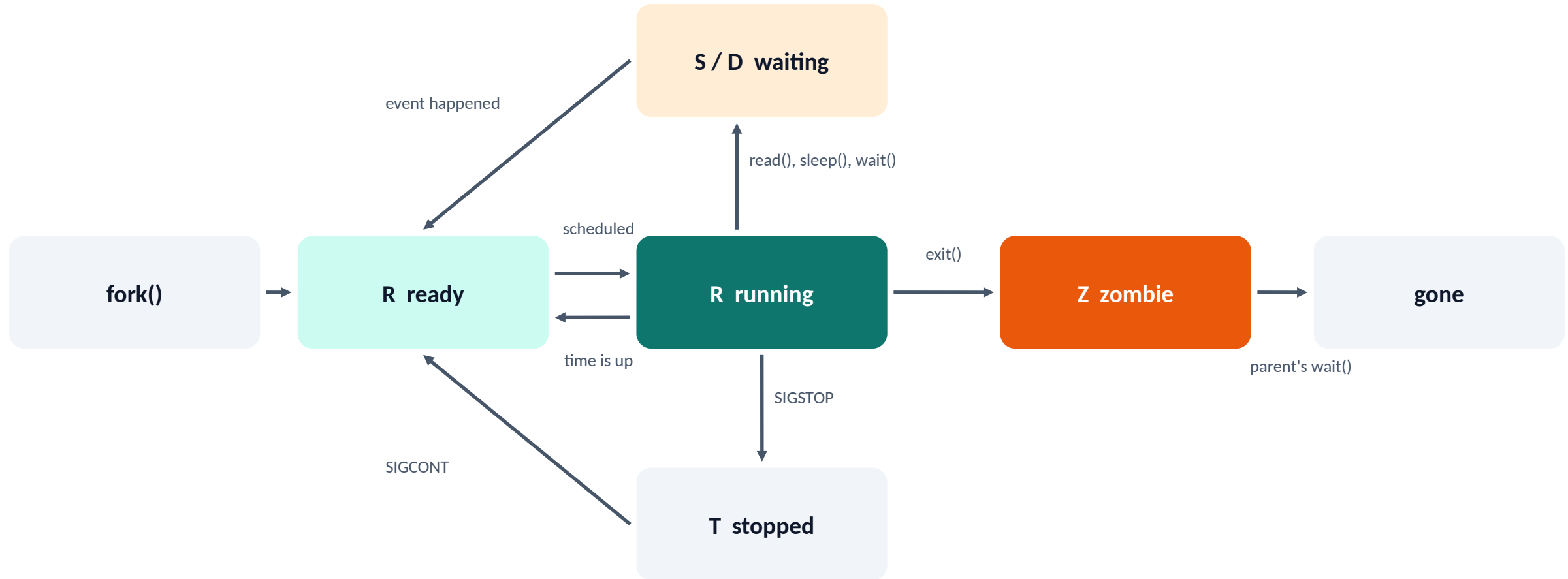
R does not mean running

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.

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)

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.

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)

The path: from `execve()` to `load_elf_binary()`

- 1 `SYSCALL_DEFINE3(execve)` (`exec.c` 2129): user space enters here (Lecture 9: how).
- 2 `do_execveat_common` (1908): open the file, asking for **execute** permission (line 917); copy `argv` and `envp` in.
- 3 `bprm_execve` → `exec_binprm` (1853, 1808): at most 5 rounds of rewriting (`#!` in `#!`), then **ELOOP** (1822).
- 4 `search_binary_handler` (1761): read the first **256 bytes** of the file into `bprm->buf`.
- 5 Ask each registered format: "is this yours?" ELF, `#!` script, ...

```
fs/exec.c, Linux 6.8 (real lines)
1776 retry:
1778 list_for_each_entry(fmt, &formats, lh) {
    ...
1783     retval = fmt->load_binary(bprm);
    ...
1787     if (bprm->point_of_no_return ||
        retval != -ENOEXEC)
1789         return retval;
1791 }

binfmt_elf.c 103:
    .load_binary = load_elf_binary,
```

C link: a list of structs holding **function pointers**, called in turn, like the comparator you pass to `qsort`.

Book: Love, LKD Ch. 3; Bovet & Cesati Ch. 20 (program execution)

Watch the kernel build cat's memory

execve: the kernel loads a program (Ubuntu 6.8 guest)

CHECKMAPSTACKRUN

the file /bin/busybox (cat)

LOAD 1 R ELF header, program headers
offset 0x000000, 0x0006c8 bytes

LOAD 2 R E code (.text)
offset 0x001000, 0x1a7865 bytes

LOAD 3 R read-only data
offset 0x1a9000, 0x0521e5 bytes

LOAD 4 RW data + .bss
offset 0x1fb820, 0x009568 bytes

entry 0x410870 · 10 program headers · no INTERP

the process's memory map (old program: sh)

sh's code, data, heap, stack ...

(all of it is about to go)

fs/binfmt_elf.c

Inside the QEMU guest the shell runs: `execve("/bin/cat", ["cat", "/proc/self/maps"], envp)`. `/bin/cat` is busybox, a static program. Watch the kernel build cat's memory.

Watch for

1. The 4 bytes **7f E L F** pick the loader.
2. **No PT_INTERP**: busybox is static.
3. Each PT_LOAD becomes **one line** of maps.
4. **.bss**: memory with no file behind it.
5. **rip = e_entry**, **rsp = argc**.

Click the video to play or pause. About 1.5 minutes, no sound.

More R tasks than CPUs: someone must choose

Each CPU has a **run queue** (struct `rq`) of tasks in state R. The scheduler, `schedule()`, picks the next one. It runs when:

- blocks** the task waits (read, sleep): **voluntary** switch
- tick** the timer says its turn is used up: **involuntary**
- wakes** a woken task deserves the CPU now: **preemption**

```
● ● ● chrt -m ; chrt -p $$ (real)
      SCHED_OTHER min/max priority   : 0/0
      SCHED_FIFO  min/max priority   : 1/99
      SCHED_RR    min/max priority   : 1/99
      SCHED_DEADLINE min/max priority : 0/0
      pid 5365's current scheduling policy: SCHED_OTHER
```

Class (checked in this order)	Policies	Used for
stop	(kernel only)	stopping a CPU, migration
deadline	SCHED_DEADLINE	tasks with a time budget
real-time	SCHED_FIFO, SCHED_RR (1-99)	audio, control loops
fair (EEVDF)	SCHED_OTHER, BATCH, IDLE	almost everything: nice -20 ... 19
idle	(kernel only)	the idle loop when nothing is R

Book: Love, LKD Ch. 4 (process scheduling)

nice → weight → share of the CPU

nice	-10	-5	0	1	3	5	10	19
weight	9548	3121	1024	820	526	335	110	15

from `sched_prio_to_weight[]`, kernel/sched/core.c 11497. Each step of nice $\approx \times 1.25$.

Two busy tasks on one CPU:
share of A = $w(A) / (w(A) + w(B))$

shares.c: nice 0 against ...		Predicted	Measured, our machine	Measured, QEMU guest (6.8)
nice 1	1024 / 1844	55.5 %	55.5 %	
nice 5	1024 / 1359	75.3 %	75.4 %	75.3 %
nice 10	1024 / 1134	90.3 %	90.3 %	90.3 %
nice 19	1024 / 1039	98.6 %	98.5 %	

A ratio, not a priority. Raising nice by 1 always costs about 10 % of the CPU, whatever the starting point. A nice 19 task still runs: it just gets 1.5 %.

C link: the kernel has no floating point here: shares come from integer weights and a fixed-point multiply.

Book: Love, LKD Ch. 4 (nice values, the weight table)

Key ideas from Class 7

- 1 A program is a file; a process is a program running, described by a `task_struct` in the kernel.
- 2 `fork()` creates a copy and returns twice: 0 in the child, the child's PID in the parent.
- 3 `exec` replaces the program inside the same process; `exit` and `wait` report how a child ended.
- 4 A zombie has exited but has not been waited for; an orphan has lost its parent.
- 5 The scheduler shares the CPU by weight: a lower nice value means a larger share.

Homework and practice

Homework 7 (bring it to the next class)

1. How many times does printf run after `for (i = 0; i < 3; i++) fork(); printf("hi\n");` ? Draw the tree.
2. Write the order of calls a shell makes to run `ls` and wait for it.
3. Two busy processes on one CPU, `nice 0` and `nice 3`: use the weights 1024 and 526 to predict each one's share.

Try it in the browser

`fork()` trees and CFS shares

[labs/lab08.html](#)

On the course website: open the lab page and choose "Try it here".

Next class: System Calls, Pipes and Signals