Process Management I

Science

Computer Science

CS 351: Systems Programming
Michael Saelee `<lee@iit.edu>`

IIT College of Science
ILLINOIS INSTITUTE OF TECHNOLOGY

§Creating Processes

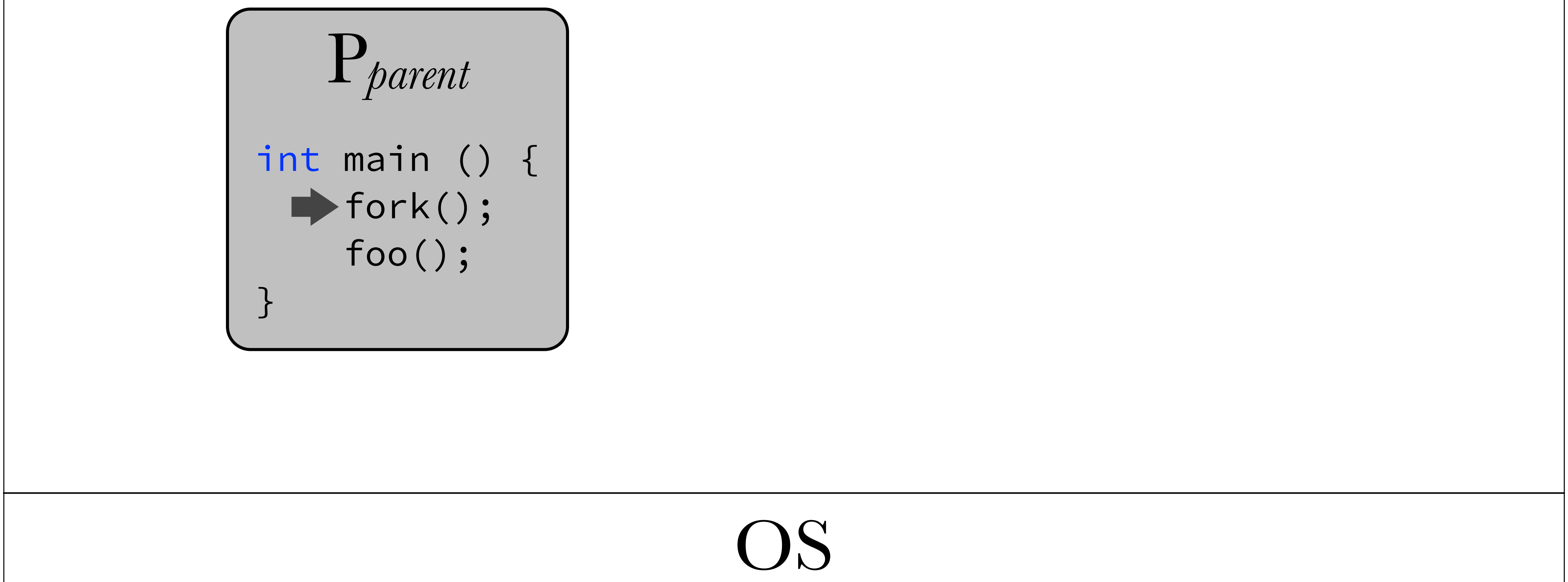
```
#include <unistd.h>

pid_t fork();
```

`fork` traps to OS to create a *new process*

… which is (mostly) a *duplicate* of the calling process!

e.g., the new (child) process runs the same program as the creating (parent) process

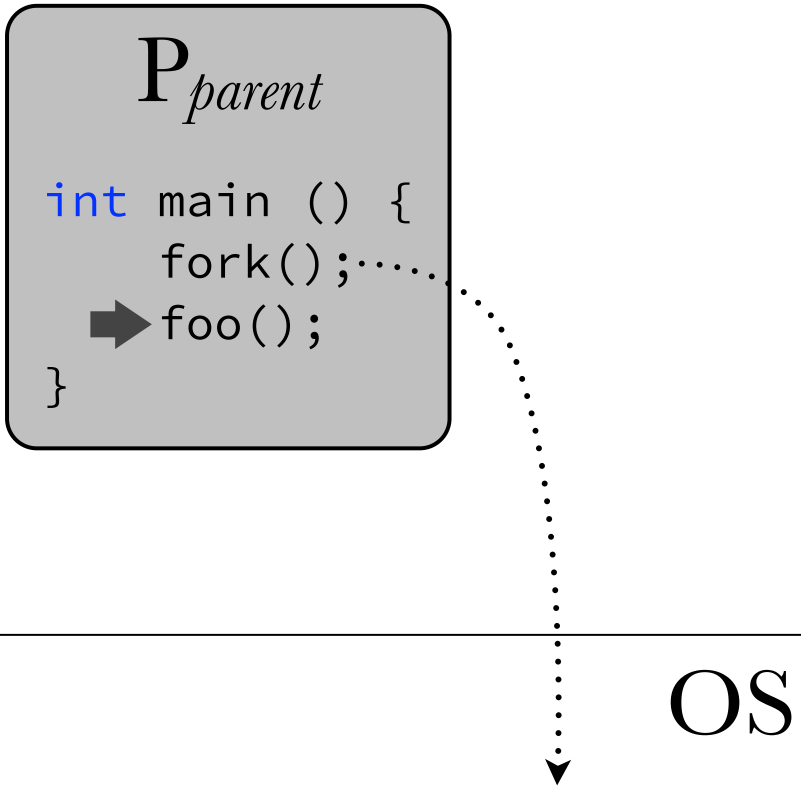
- and starts with the same PC,
- the same SP, FP, regs,
- the same open files, etc., etc.



P_{parent}

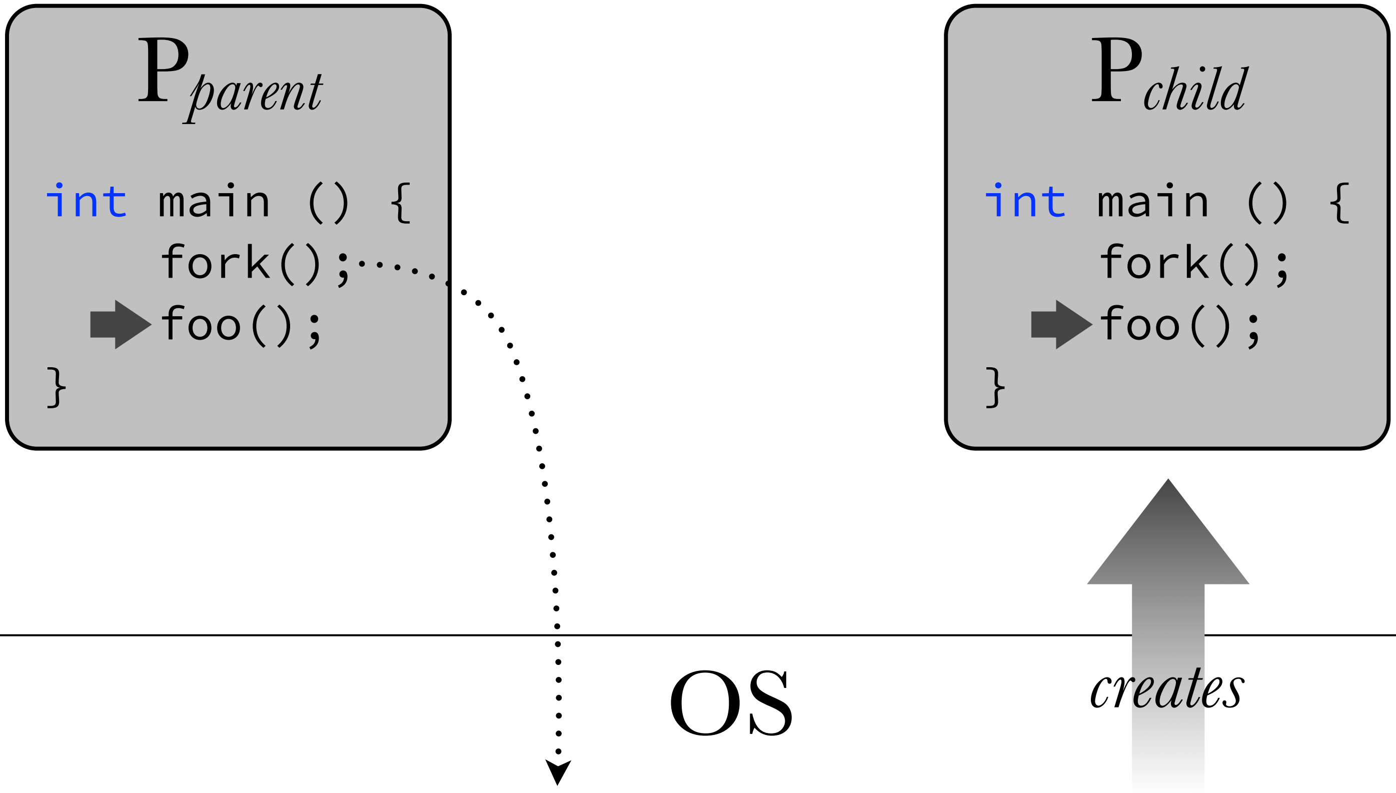
```
int main () {
   ➡fork();
    foo();
}
```

OS



P_{parent}

```
int main () {
    fork();
  ➡ foo();
}
```



P_{parent}

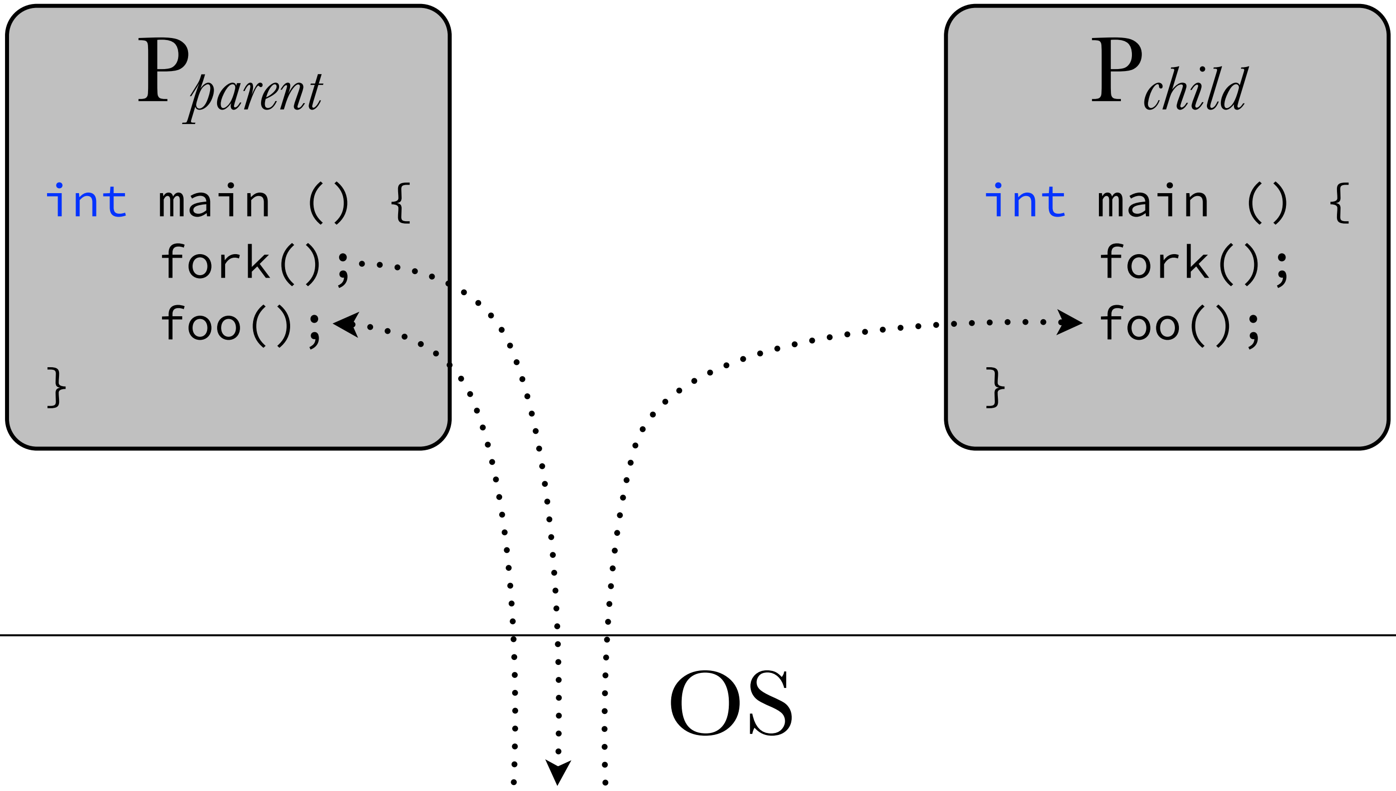
```
int main () {
     fork();
   ➡ foo();
}
```

P_{child}

```
int main () {
     fork();
   ➡ foo();
}
```

OS

creates



P_{parent}

```
int main () {
    fork();
    foo();
}
```

P_{child}

```
int main () {
    fork();
    foo();
}
```

OS

`fork`, when called, returns *twice*

(to each process @ the next instruction)

```c
int main () {
    fork();
    printf("Hello world!\n");
}
```

```
Hello world!
Hello world!
```

```c
int main () {
    fork();
    fork();
    printf("Hello world!\n");
}
```

```
Hello world!
Hello world!
Hello world!
Hello world!
```

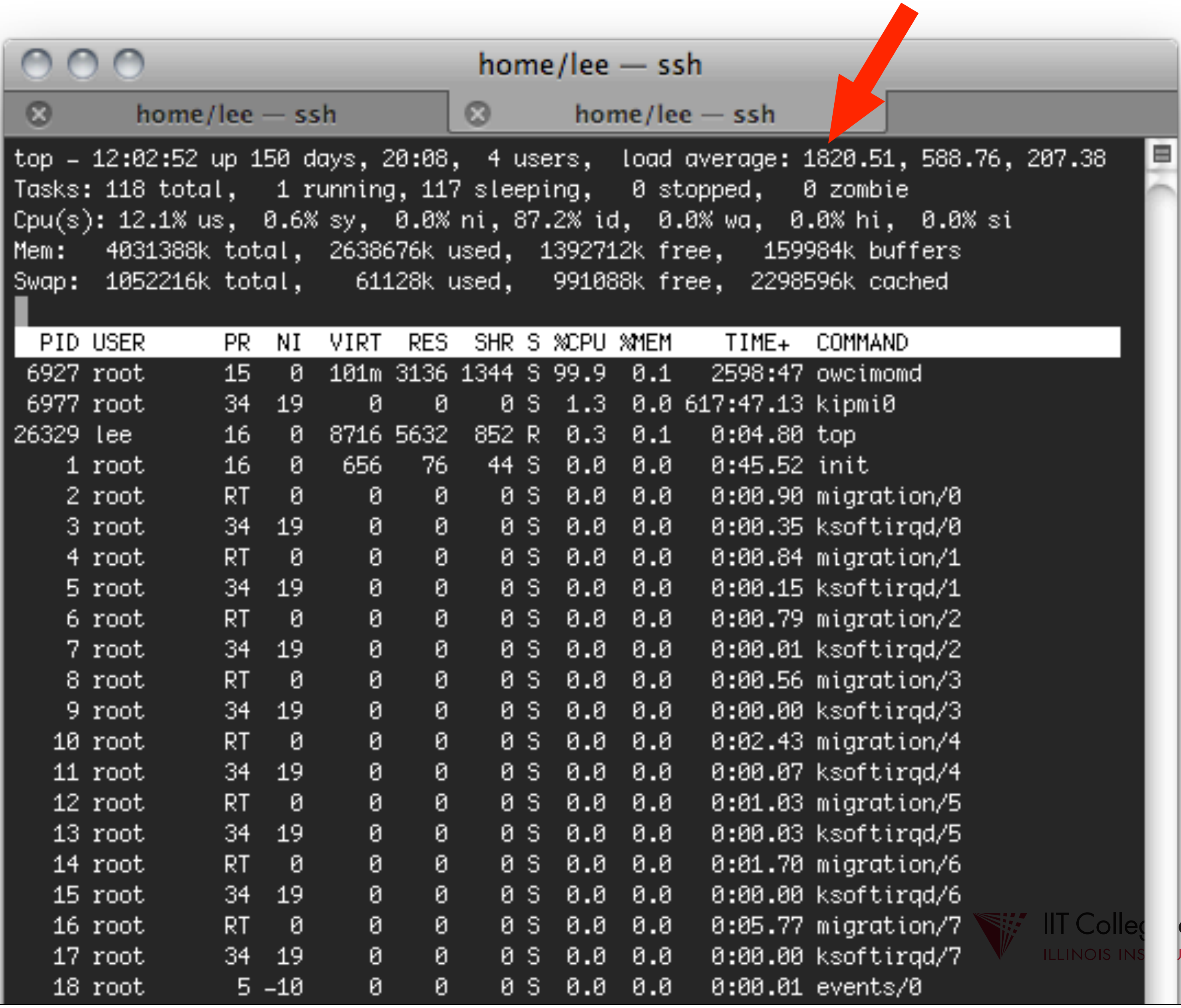
```c
int main () {
    fork();
    fork();
    fork();
    printf("Hello world!\n");
}
```

```
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
```

```
int main() {
    while(1)
        fork();
}
```

the “fork bomb”

(I didn’t show you this)



# processes waiting to be scheduled

```
top - 12:02:52 up 150 days, 20:08,  4 users,  load average: 1820.51, 588.76, 207.38
Tasks: 118 total,   1 running, 117 sleeping,   0 stopped,   0 zombie
Cpu(s): 12.1% us,  0.6% sy,  0.0% ni, 87.2% id,  0.0% wa,  0.0% hi,  0.0% si
Mem:   4031388k total,  2638676k used,  1392712k free,   159984k buffers
Swap:  1052216k total,    61128k used,   991088k free,  2298596k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
 6927 root      15   0  101m 3136 1344 S 99.9  0.1   2598:47 owcimomd
 6977 root      34  19     0    0    0 S  1.3  0.0 617:47.13 kipmi0
26329 lee       16   0  8716 5632  852 R  0.3  0.1   0:04.80 top
    1 root      16   0   656   76   44 S  0.0  0.0   0:45.52 init
    2 root      RT   0     0    0    0 S  0.0  0.0   0:00.90 migration/0
    3 root      34  19     0    0    0 S  0.0  0.0   0:00.35 ksoftirqd/0
    4 root      RT   0     0    0    0 S  0.0  0.0   0:00.84 migration/1
    5 root      34  19     0    0    0 S  0.0  0.0   0:00.15 ksoftirqd/1
    6 root      RT   0     0    0    0 S  0.0  0.0   0:00.79 migration/2
    7 root      34  19     0    0    0 S  0.0  0.0   0:00.01 ksoftirqd/2
    8 root      RT   0     0    0    0 S  0.0  0.0   0:00.56 migration/3
    9 root      34  19     0    0    0 S  0.0  0.0   0:00.00 ksoftirqd/3
   10 root      RT   0     0    0    0 S  0.0  0.0   0:02.43 migration/4
   11 root      34  19     0    0    0 S  0.0  0.0   0:00.07 ksoftirqd/4
   12 root      RT   0     0    0    0 S  0.0  0.0   0:01.03 migration/5
   13 root      34  19     0    0    0 S  0.0  0.0   0:00.03 ksoftirqd/5
   14 root      RT   0     0    0    0 S  0.0  0.0   0:01.70 migration/6
   15 root      34  19     0    0    0 S  0.0  0.0   0:00.00 ksoftirqd/6
   16 root      RT   0     0    0    0 S  0.0  0.0   0:05.77 migration/7
   17 root      34  19     0    0    0 S  0.0  0.0   0:00.00 ksoftirqd/7
   18 root       5 -10     0    0    0 S  0.0  0.0   0:00.01 events/0
```

```
typedef int pid_t;

pid_t fork();
```

- system-wide unique process identifier
- child's pid (> `0`) is returned *in the parent*
- sentinel value `0` is returned *in the child*

```c
void fork0() {
    int pid = fork();
    if (pid == 0)
        printf("Hello from Child!\n");
    else
        printf("Hello from Parent!\n");
}

main() { fork0(); }
```

```
Hello from Child!
Hello from Parent!
```

(or)

```
Hello from Parent!
Hello from Child!
```

i.e., order of execution is *nondeterministic*

- parent & child run *concurrently*!

```
void fork1 () {
    int x = 1;

    if (fork() == 0) {
        printf("Child has x = %d\n", ++x);
    } else {
        printf("Parent has x = %d\n", --x);
    }
}
```

```
Parent has x = 0
Child has x = 2
```

important: post-fork, parent & child are identical, but *separate*!

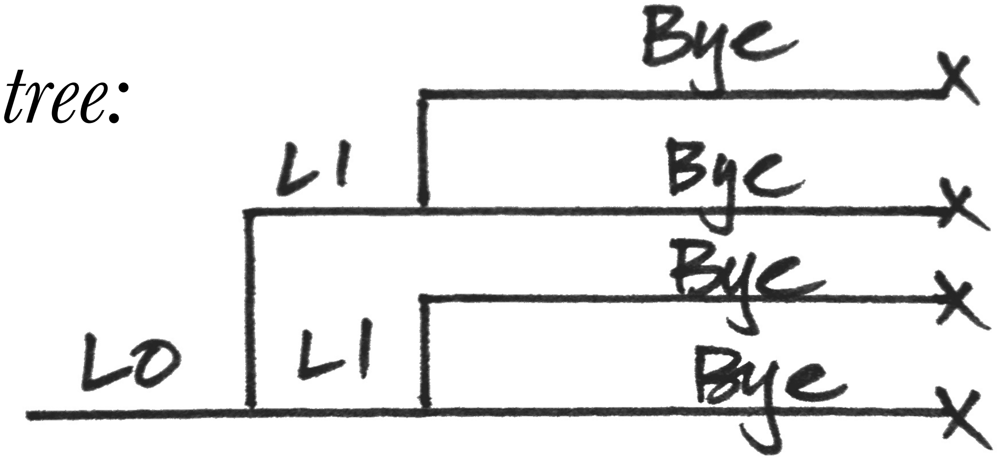
- OS allocates and maintains separate data/state
- control flow can diverge

```c
void fork2() {
    printf("L0\n");
    fork();
    printf("L1\n");
    fork();
    printf("Bye\n");
}
```

```
L0
L1
L1
Bye
Bye
Bye
Bye
```

```
void fork2() {
    printf("L0\n");
    fork();
    printf("L1\n");
    fork();
    printf("Bye\n");
}
```

process tree:



```c
void fork2() {
    printf("L0\n");
    fork();
    printf("L1\n");
    fork();
    printf("Bye\n");
}
```

Which are possible?

A.

```
L1
L0
L1
Bye
Bye
Bye
Bye
```

B.

```
L0
L1
Bye
Bye
L1
Bye
Bye
```

C.

```
L0
L1
Bye
Bye
Bye
L1
Bye
```

D.

```
L1
Bye
Bye
L0
L1
Bye
Bye
```

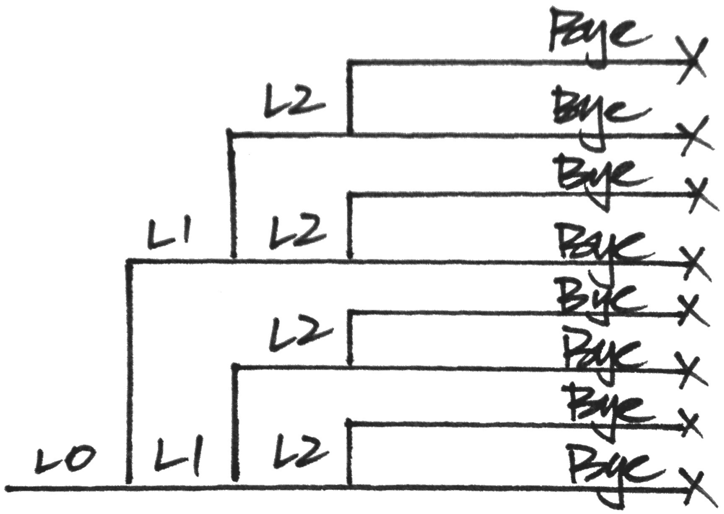
E.

```
L0
Bye
Bye
L1
L1
Bye
Bye
```

```c
main() {
    fork();
    fork();
    while(1) ;
}
```

```
$ ./a.out &
[1] 8198

$ pstree -p 8198
a.out(8198)─┬─a.out(8199)───a.out(8201)
            └─a.out(8200)
```

```
void fork3() {
    printf("L0\n");
    fork();
    printf("L1\n");
    fork();
    printf("L2\n");
    fork();
    printf("Bye\n");
}
```



```c
void fork4() {
    printf("L0\n");
    if (fork() != 0) {
        printf("L1\n");
        if (fork() != 0) {
            printf("L2\n");
            fork();
        }
    }
    printf("Bye\n");
}
```

A.

```
L0
L1
L2
Bye
Bye
Bye
Bye
```

B.

```
L0
L1
Bye
Bye
L2
Bye
Bye
```

C.

```
Bye
L0
Bye
L1
Bye
L2
Bye
```

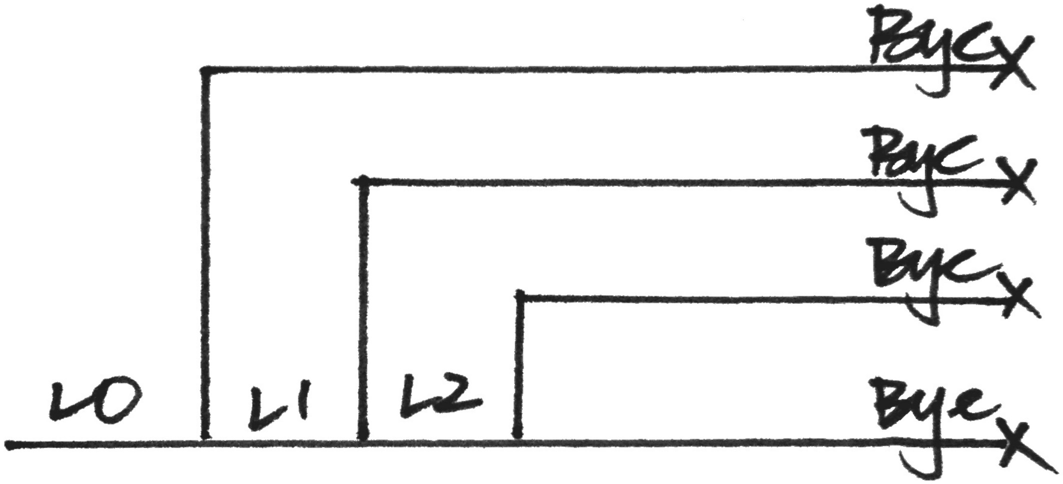
D.

```
L0
Bye
L1
Bye
L2
Bye
Bye
```

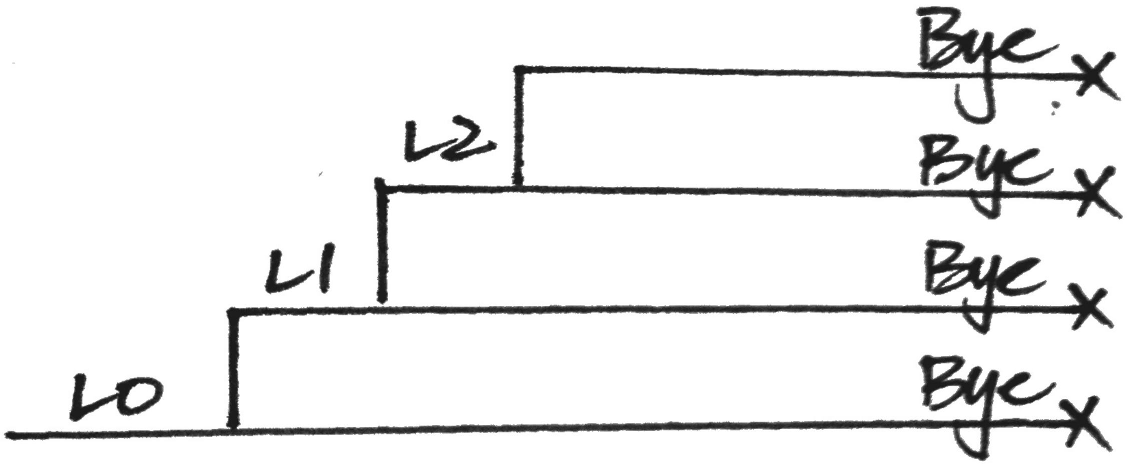
E.

```
L0
L1
Bye
Bye
Bye
L2
Bye
```

```
void fork4() {
    printf("L0\n");
    if (fork() != 0) {
        printf("L1\n");
        if (fork() != 0) {
            printf("L2\n");
            fork();
        }
    }
    printf("Bye\n");
}
```



```
void fork5() {
    printf("L0\n");
    if (fork() == 0) {
        printf("L1\n");
        if (fork() == 0) {
            printf("L2\n");
            fork();
        }
    }
    printf("Bye\n");
}
```



a good question: what if `fork` fails?

most syscalls return `-1` on failure

global var `errno` populated with “cause”

```c
#include <errno.h>

extern int errno;

/* get error string */
char *strerror(int errnum);

/* print error string w/ message */
void perror(const char *s);
```

```c
int fd = open("/etc/shadow", O_RDONLY);
if (fd == -1) {
    perror("Uh-oh");
    exit(1);
}
```

```
$ ./errtest
Uh-oh: Permission denied
```

```
$ man errno

NAME
       errno - number of last error

SYNOPSIS
       #include <errno.h>

DESCRIPTION
       ...

       E2BIG           Argument list too long (POSIX.1)
       EACCES          Permission denied (POSIX.1)
       EADDRINUSE      Address already in use (POSIX.1)
       EADDRNOTAVAIL   Address not available (POSIX.1)
       EAFNOSUPPORT    Address family not supported (POSIX.1)
       EAGAIN          Resource temporarily unavailable (may be the same
value
                       as EWOULDBLOCK) (POSIX.1)
       EALREADY        Connection already in progress (POSIX.1)
       EBADE           Invalid exchange
       ...
```

§Terminating Processes

```
int main () {
    return 0;
}
```

```
void exit(int status);
void foo() {
    exit(1); /* no return */
}

int main () {
    foo();   /* no return */
    return 0;
}
```

Unix convention:

- normal termination → exit status `0`
- other exit status values = error

```
void foo() {
    exit(1);
}

int main () {
    ...
    foo();             /* $^@#%!! */

    release(resource); /* cleanup */
    return 0;
}
```

```c
int atexit(void (*fn)());
int atexit(void (*fn)());
```

- registers function to call before exiting
- can call multiple times; functions are invoked in reverse order

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

void foo() {
    printf("Self-destructing\n");
    exit(1);
}

int main() {
    atexit(cleanup);
    foo(); /* no return */
    return 0;
}
```

```
Self-destructing
Cleaning up
```

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

void foo() {
    fork();
    exit(1);
}

int main() {
    atexit(cleanup);
    foo(); /* no return */
    return 0;
}
```

```
Cleaning up
Cleaning up
```

i.e., `atexit` handlers are "inherited" by child processes on fork

```c
void fork7() {
    if (fork() == 0) {
        printf("Terminating Child, PID = %d\n", getpid());
        exit(0);
    } else {
        printf("Running Parent, PID = %d\n", getpid());
        while (1) ; /* Infinite loop */
    }
}
```

(demo)

All terminating processes turn into *zombies*

“dead” but still tracked by OS

- pid remains in use
- exit status can be queried

§Reaping Processes (& Synchronization)