IT628: Systems Programming

Course outline, process creation

What's this course all about?

- Operating systems provide a set of constructs and well-defined primitives to simplify application development
- You will learn how to write applications that exploit important OS features
- Often we will present an OS concept first and then the system calls associated with that concept

Main topics we'll cover

- **Concurrent programming**
  - processes, signals, pipes, threads, synchronization
- **Network programming**
  - sockets and servers

System calls

- **A request for the OS to do something on behalf of the user's program**
- **Example**
  - fork() /* create a child process */
  - exec() /* executing a program */
- **Don't confuse system calls with libc calls**
  - strcpy()

Main categories of system calls

- **File system**
  - Low-level file I/O
  - E.g., creat, open, read, write, lseek, close
- **Multi-tasking mechanisms**
  - Process control
  - E.g., fork, wait, exec, exit, signal, kill
- **Inter-process communication**
  - E.g., pipe, dup, dup2

Classroom organization

- Standard lecture format with exercises sprinkled in for class discussion
- We'll also demo code examples in class
- Questions are ALWAYS encouraged (both directions)

Classroom Etiquette

- Come on time to both class and labs
- Talking, cell phones, etc. will not be tolerated

Reference books

- **Keith Haviland, Dina Gray and Ben Salama**
  - "UNIX System Programming", Addison-Wesley
- **Randal Bryant and David O'Hallaron**
  - "Computer Systems: A Programmer's Perspective", Pearson India
- **Brian Kernighan and Dennis Ritchie**
  - "The C Programming Language, Second Edition", Prentice Hall India

Grade breakdown

- **Exams (70%): two insem exams and final**
  - two insems weighted 35%
  - final weighted 35% (covers entire course)
- **Lab exercises (checkoffs) and homework (30%)**
  - weighted 20%, 10%
  - lab attendance is mandatory

Processes

- **A process is the basic unit of execution in an OS**
- **A process is a running instance of a program**
  - Program = static file (image)
  - Process = executing program = program + execution state
- **Two processes are said to run concurrently when instructions of one process are interleaved with the instructions of the other process**

Context switching

- **Transferring control from the current process to another process is called a context switch**
  - OS saves the state of the current process, restores the state of the other process, and passes control to the new process
- **From a human observer's point of view, many processes appear to proceed simultaneously**

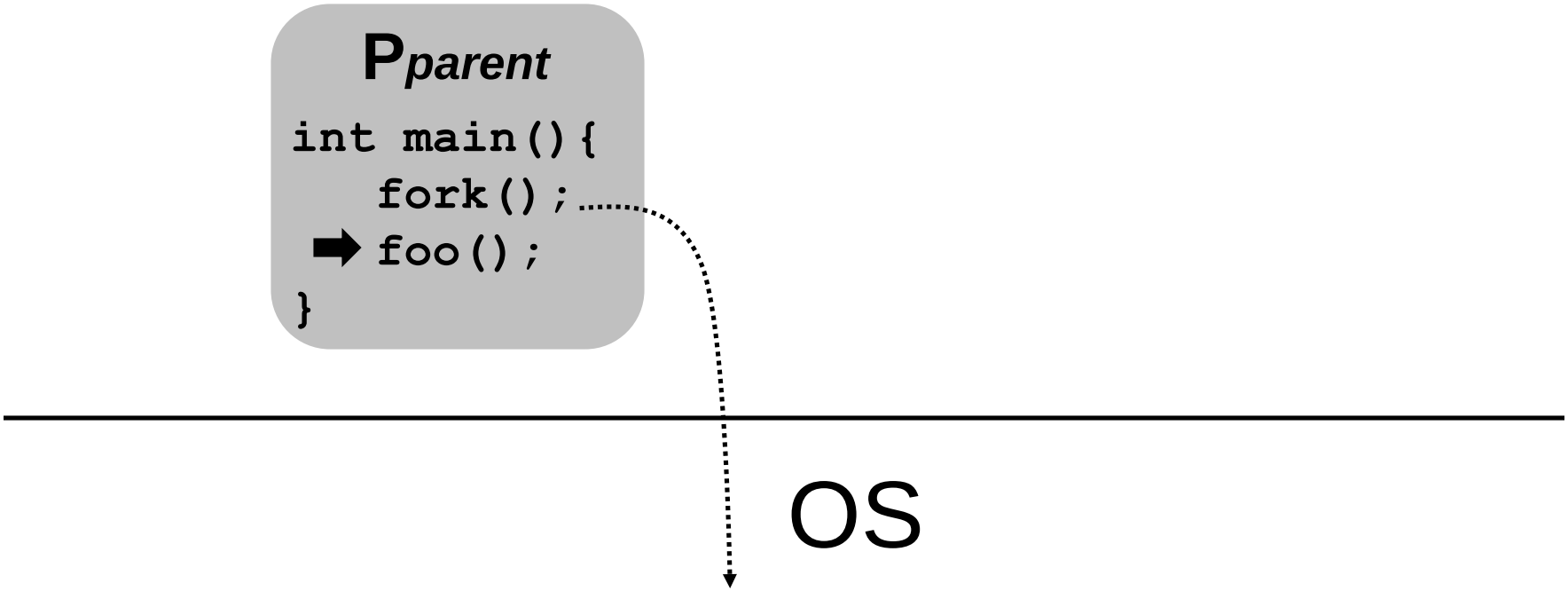
fork()

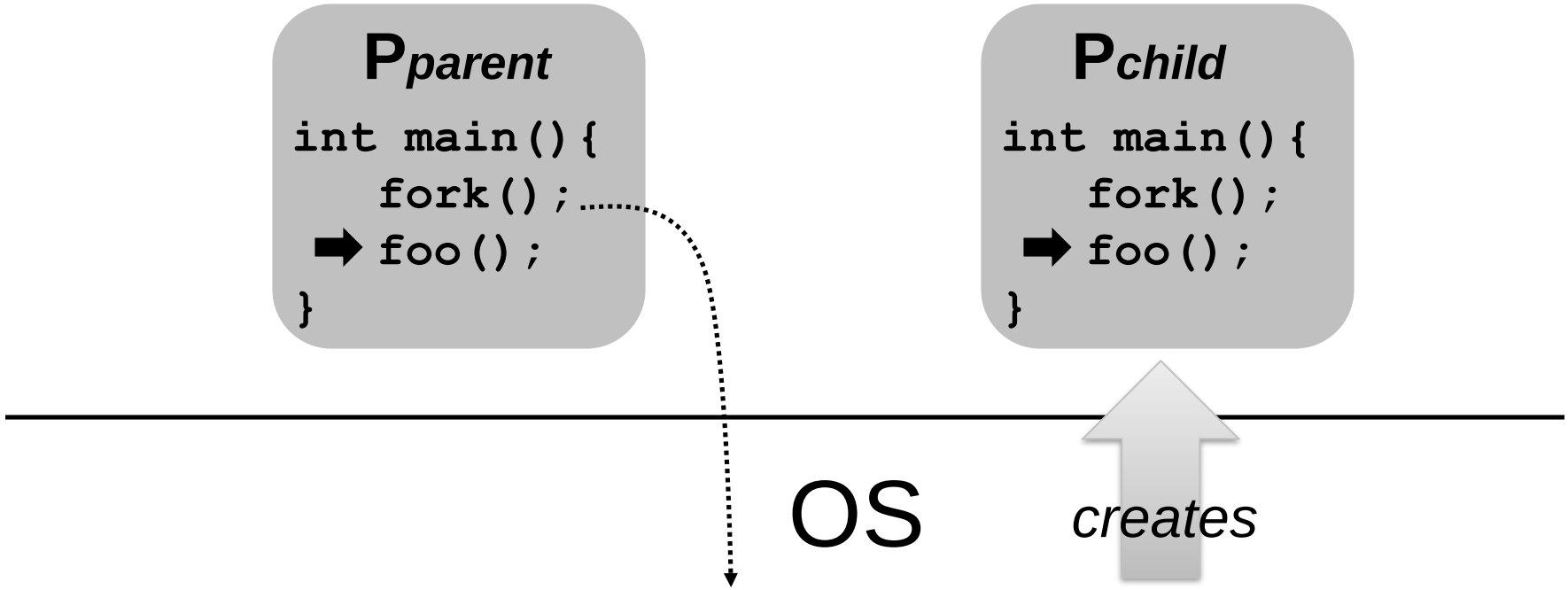
- **fork creates a new process**
- **the process created (child) runs the same program as the creating (parent) process**
  - and starts with the same PC,
  - the same CPU registers,
  - the same open files, etc.

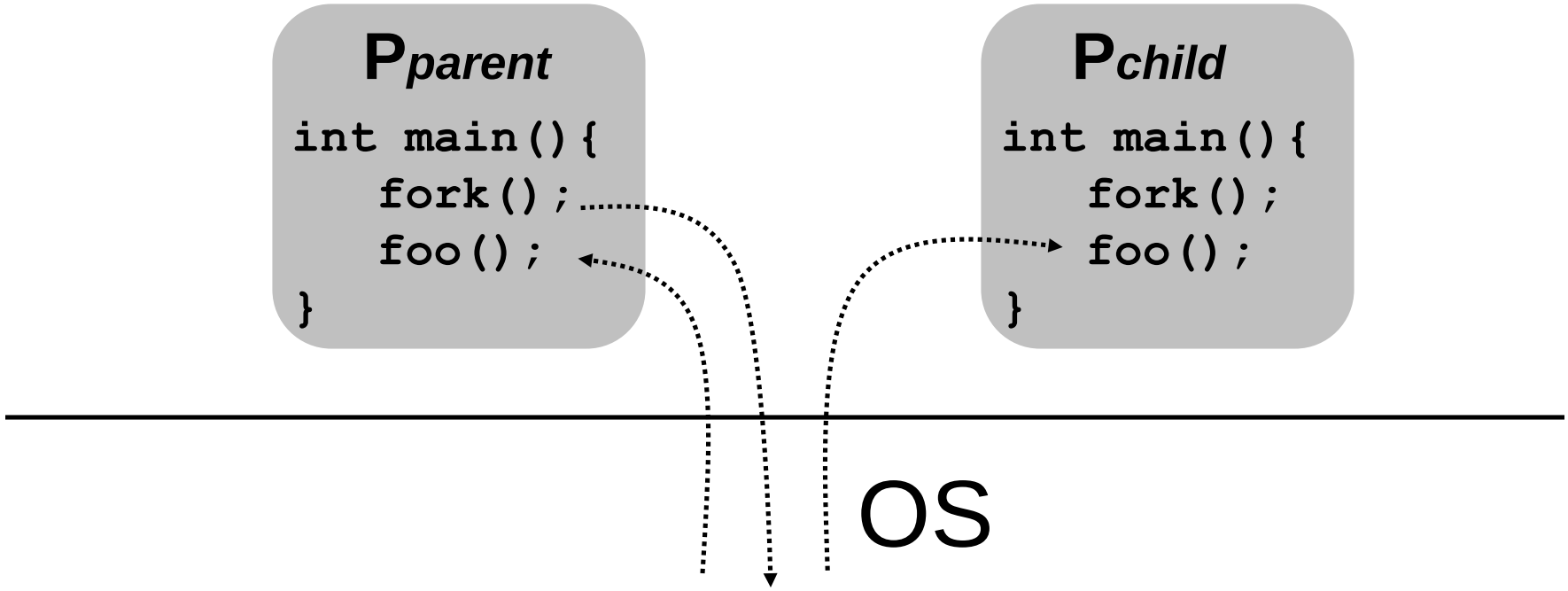
P*parent*

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

OS


P parent
int main(){
fork();
foo();
}
OS


Pparent
int main(){
fork();
foo();
}
Pchild
int main(){
fork();
foo();
}
OS
creates


Pparent
int main(){
fork();
foo();
}
Pchild
int main(){
fork();
foo();
}
OS

- **fork(), when called, returns twice**
  (to each process @ the next instruction)

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

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

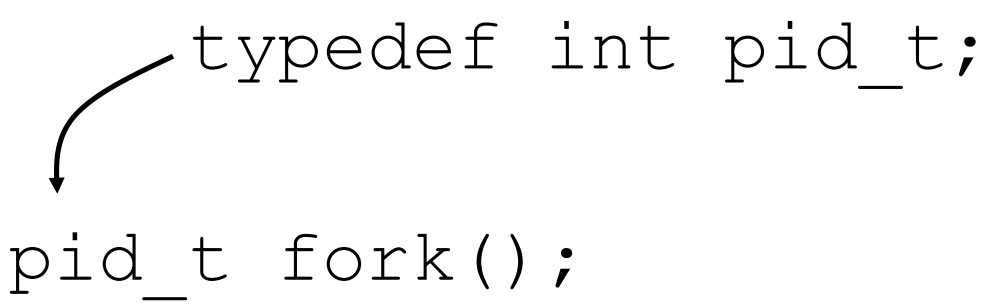
```
int main() {
    fork();
    fork();
    printf("Hello world!\n");
}
```

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

```
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!

return value of fork()



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

```
void fork0() {
   if (fork()==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!

```
void fork3() {
    int i;
    sum = 0;
    fork(); /* create a new process */
    for (i=1 ; i<=5 ; i++) {
      printf ("The value of i is %d\n", i);
      fflush(stdout);
      sum += i;
    }
    printf ("The sum is %d\n", sum);
    exit(0)
}

main(){ fork3(); }
```

- **order of execution is non-deterministic**
  - parent and child run concurrently
- **Important: post fork, parent and child are identical but separate!**
  - OS allocates and maintains separate data/state
  - control flow can diverge

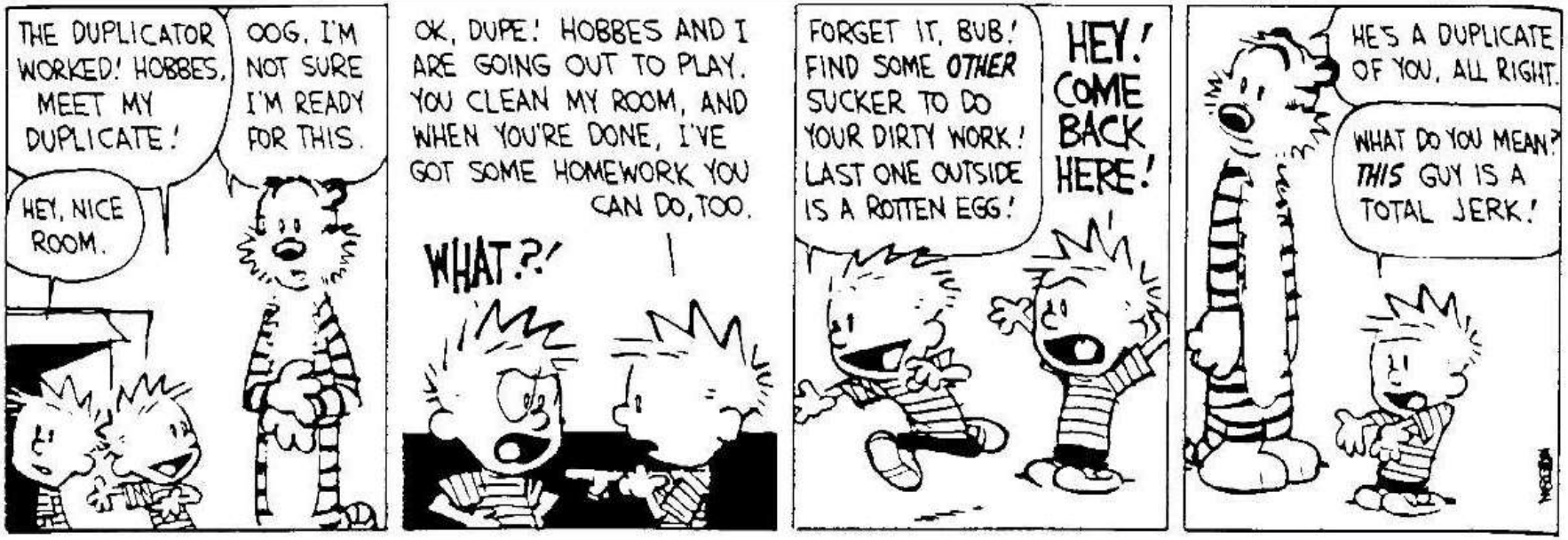
```
void fork1() {
   int x = 1;
   if (fork()==0){
      printf("Child has x = %d\n", ++x);
   } else {
      printf("Parent has x = %d\n", --x);
   }
}
```

Parent has x = 0
Child has x = 2

Summary

- process != program
- Process: dynamic execution context of an executing program
- Several processes may run the same program code, but each is a distinct process with its own state
- Read http://stackoverflow.com/questions/985051/what-is-the-purpose-of-fork

Humor


THE DUPLICATOR WORKED! HOBBES, MEET MY DUPLICATE!
OOG. I'M NOT SURE I'M READY FOR THIS.
HEY, NICE ROOM.
OK, DUPE! HOBBES AND I ARE GOING OUT TO PLAY. YOU CLEAN MY ROOM, AND WHEN YOU'RE DONE, I'VE GOT SOME HOMEWORK YOU CAN DO, TOO.
WHAT?!
FORGET IT, BUB! FIND SOME *OTHER* SUCKER TO DO YOUR DIRTY WORK! LAST ONE OUTSIDE IS A ROTTEN EGG!
HEY! COME BACK HERE!
HE'S A DUPLICATE OF YOU, ALL RIGHT.
WHAT DO YOU MEAN? *THIS* GUY IS A TOTAL JERK!

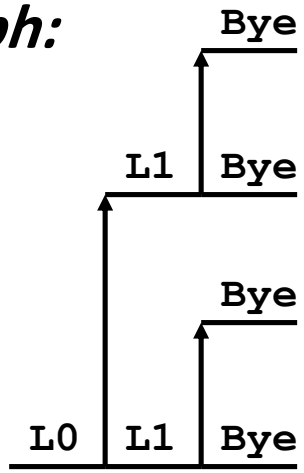
fork practice problems

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

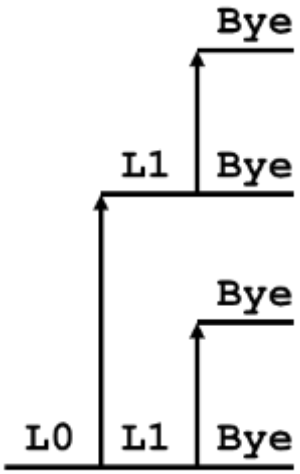
```
L0
L1
L1
Bye
Bye
Bye
Bye
```

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

process graph:



```
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

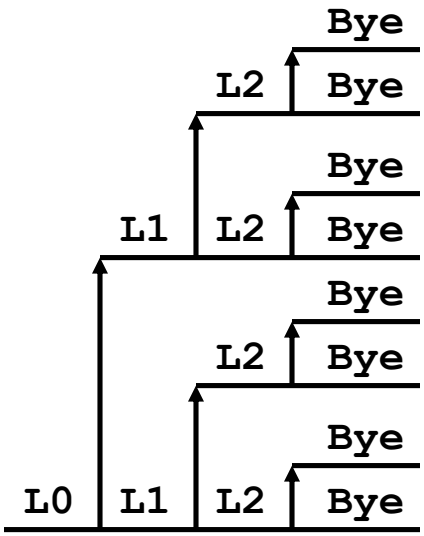
D.

L1
Bye
Bye
L0
L1
Bye
Bye

E.

L0
Bye
Bye
L1
L1
Bye
Bye

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



```
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

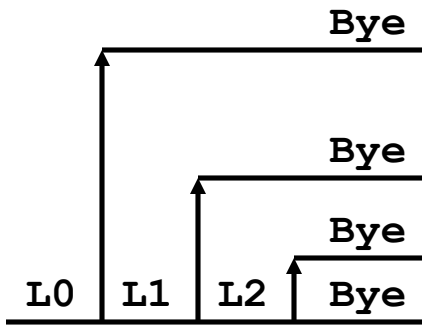
D.

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");
}
```