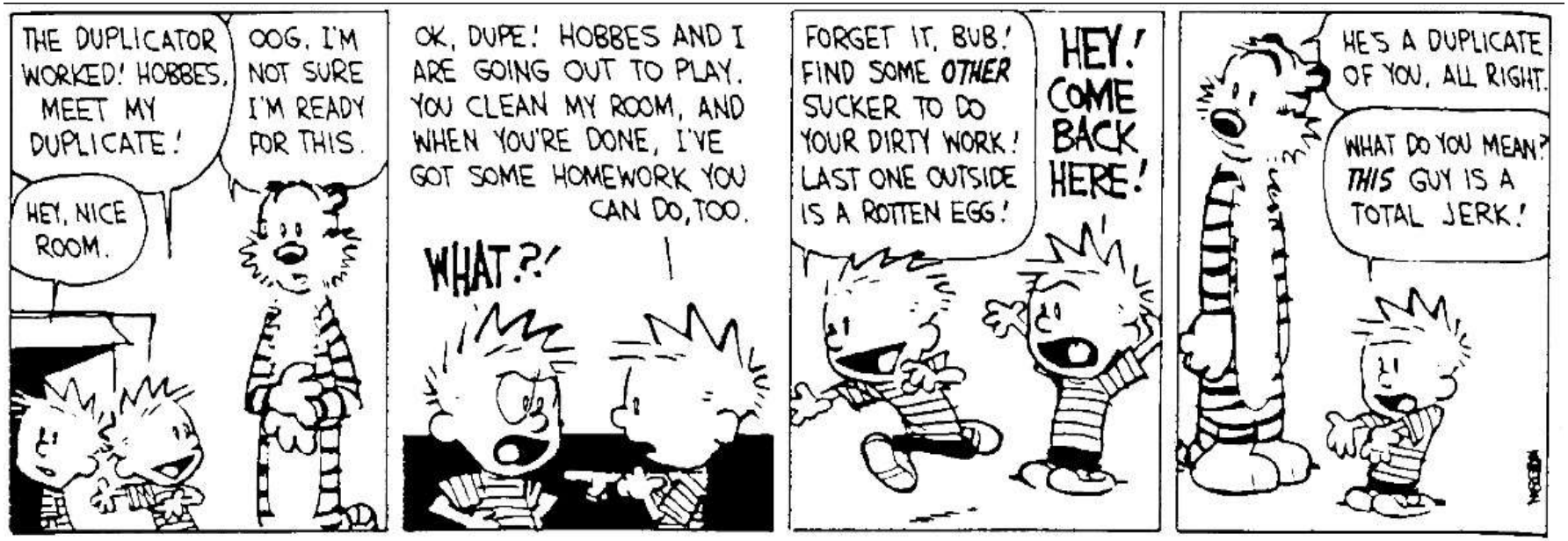
Forking processes

Systems Software



Process Control

- A Process is a running instance of a program
- Many processes may run concurrently
- May have duplicate instances of the same program running concurrently

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()

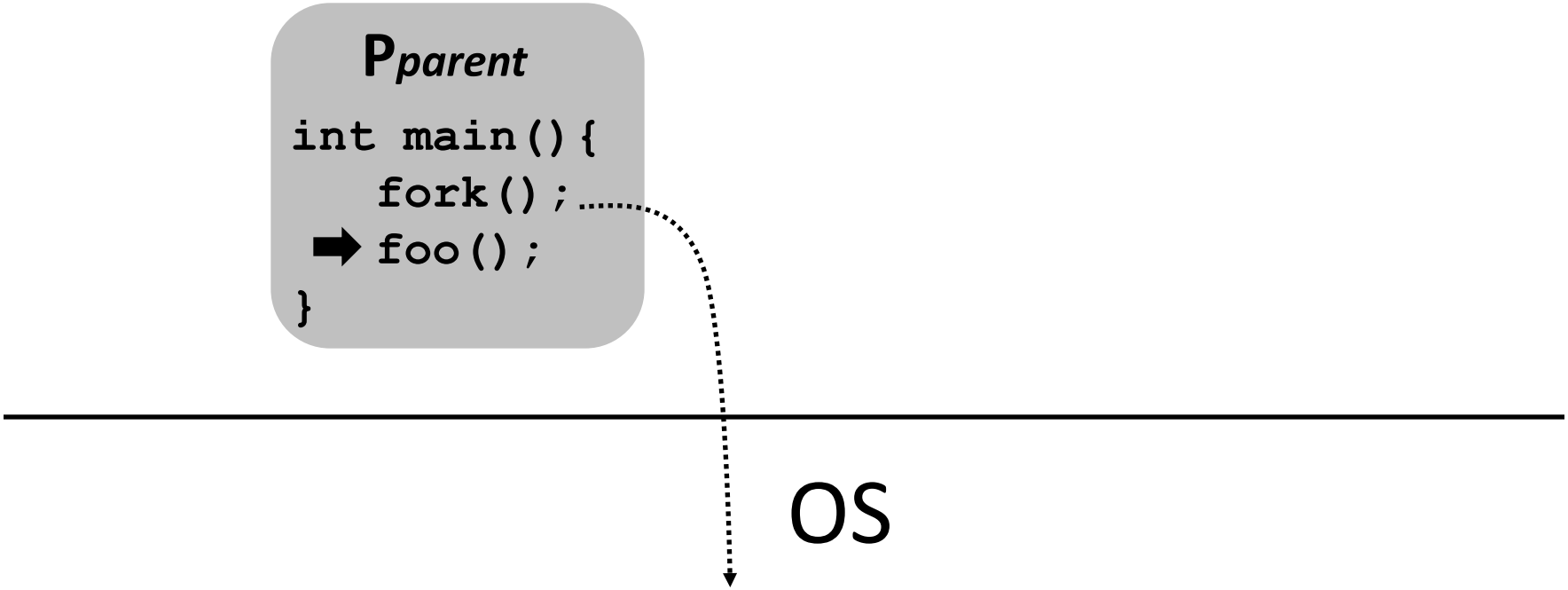
fork()

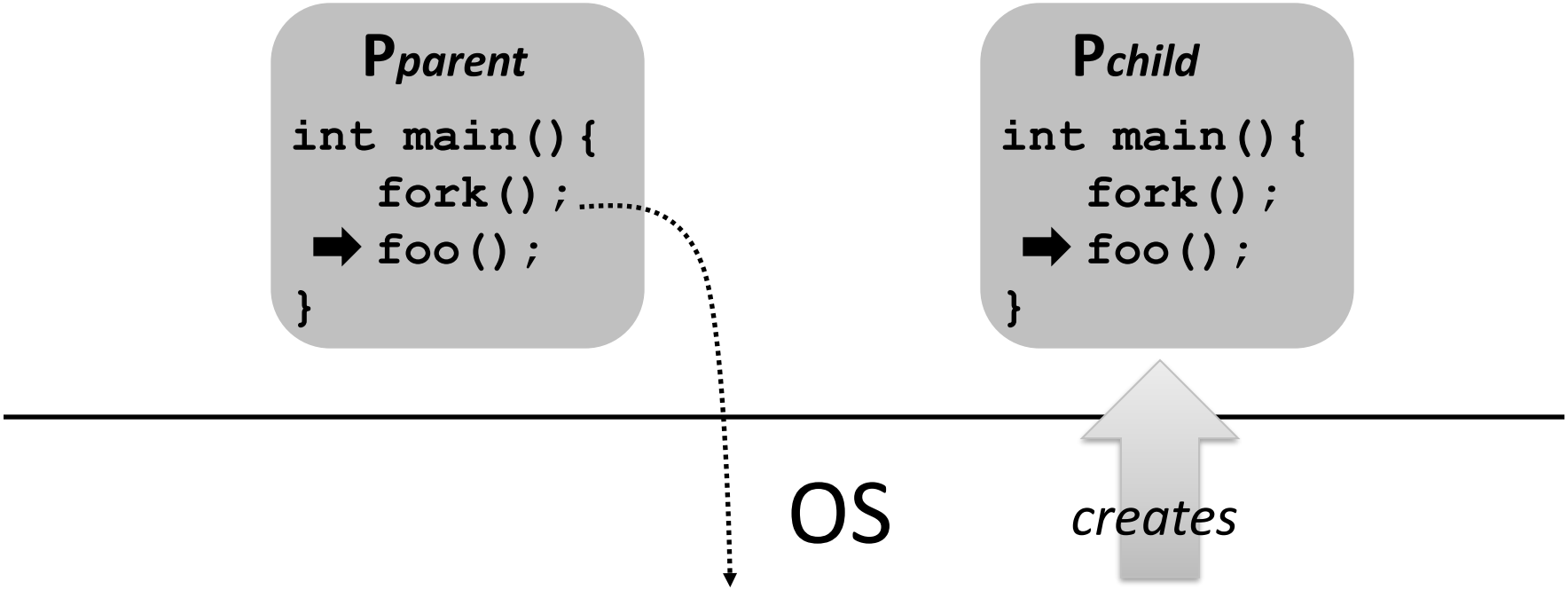
- **fork creates a new process**
- **the process created (child) runs the same program as the creating (parent) process**
  - and starts with the same PC,
  - the same %esp, %ebp, regs,
  - the same open files, etc.

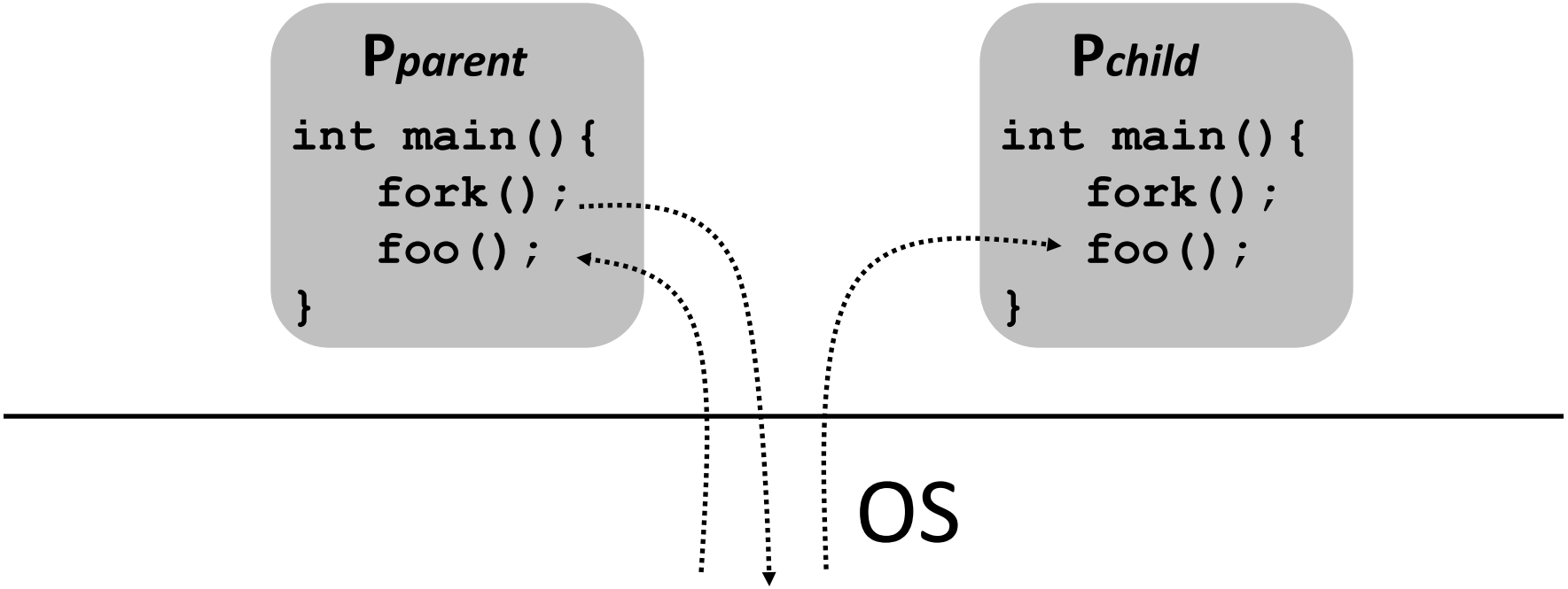
P*parent*

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

OS


Pparent
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()

```
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

```
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!
```

- **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);
   }
}
```

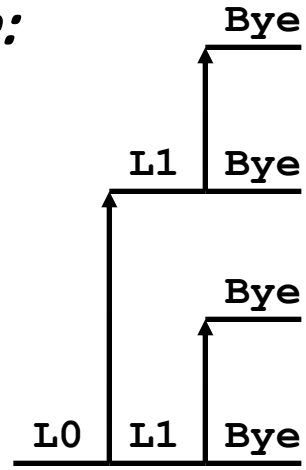
Parent has x = 0
Child has x = 2

```
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:

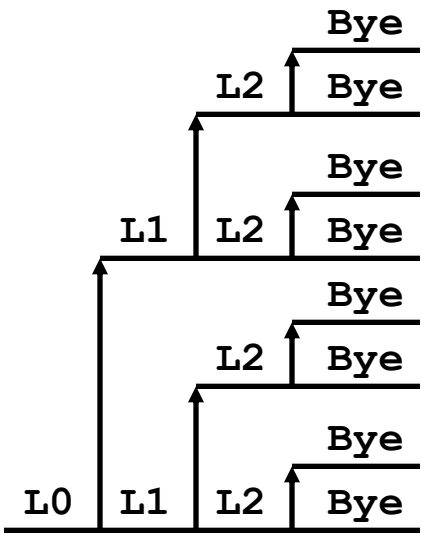


```
void fork2() {
    printf("L0\n");
    fork();
    printf("L1\n");
    fork();
    printf("Bye\n");
}
```

Which are possible?

A.	B.	C.	D.	E.
L1	L0	L0	L1	L0
L0	L1	L1	Bye	Bye
L1	Bye	Bye	Bye	Bye
Bye	Bye	Bye	L0	L1
Bye	L1	Bye	L1	L1
Bye	Bye	L1	Bye	Bye
Bye	Bye	Bye	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

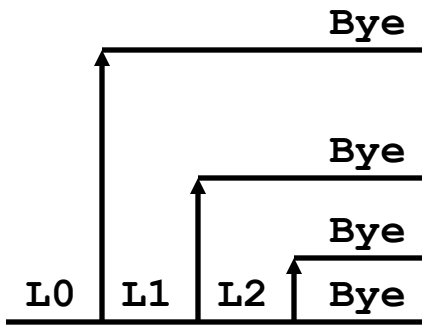
D.

L0
Bye
L1
Bye
L2
Bye
Bye

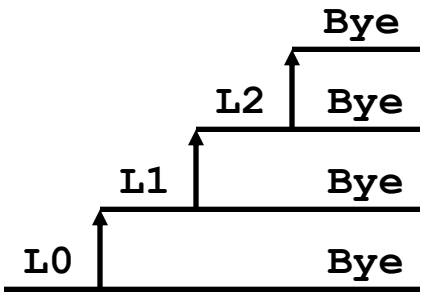
E.

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");
}
```



What is the purpose of fork?

- http://stackoverflow.com/questions/985051/what-is-the-purpose-of-fork