

# Understanding fork() Call

# How a fork() call works?

**P<sub>parent</sub>**

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

---

OS

# How a fork() call works?

**P<sub>parent</sub>**

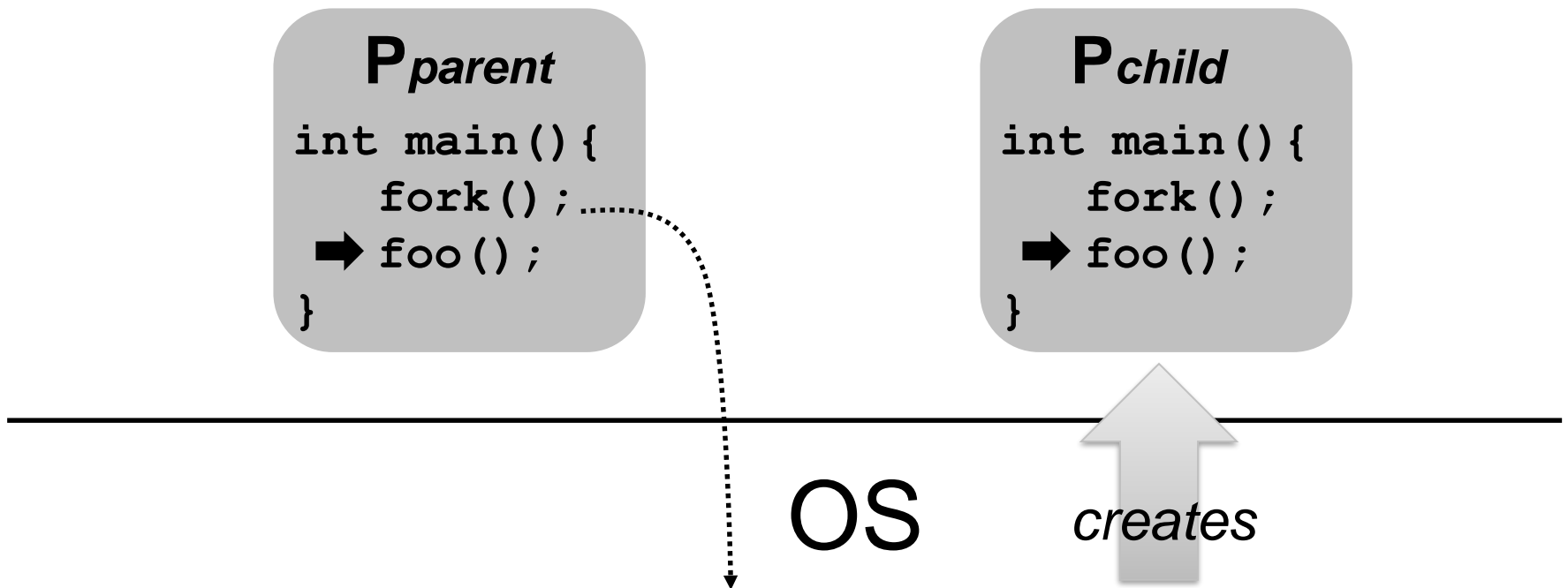
```
int main() {  
    fork();  
    ➡ foo();  
}
```



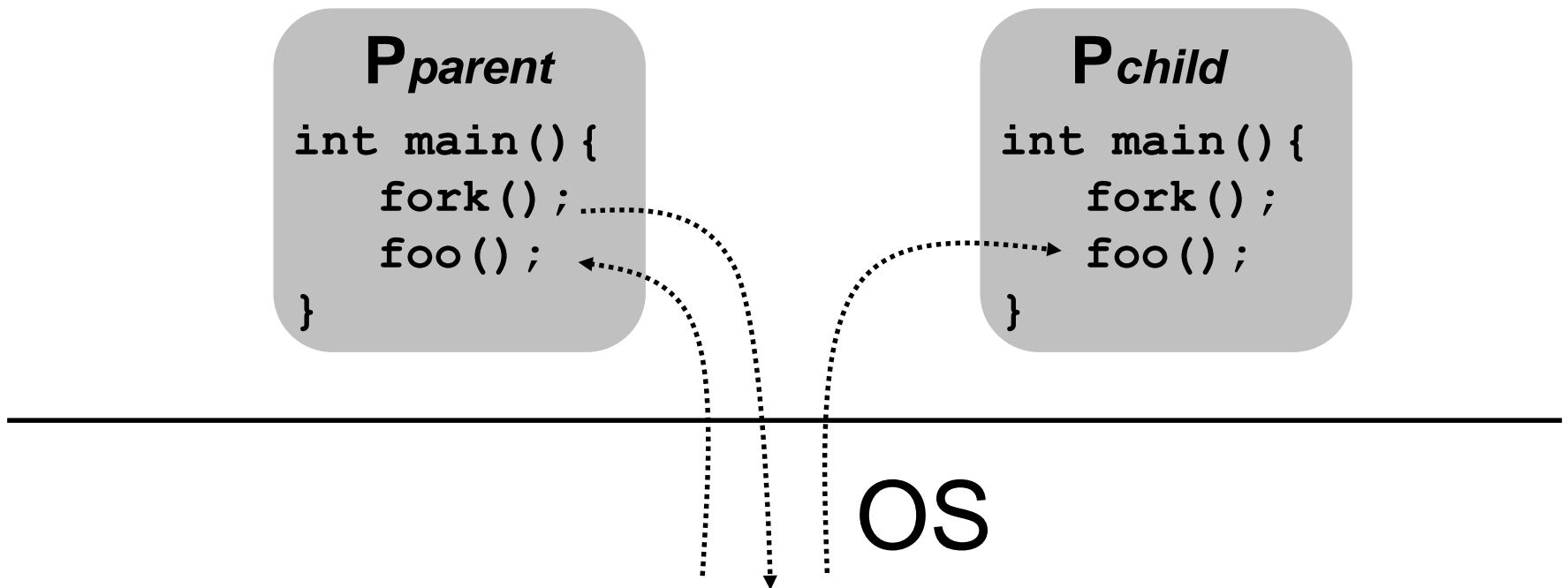
The diagram illustrates the system call interface. A horizontal line separates the user space (above) from the operating system (below). A box in the user space contains the code for a parent process. A dotted arrow originates from the `fork()` call in the code, curves downwards, crosses the horizontal line, and ends with an arrowhead pointing to the label 'OS' in the system space.

OS

# How a fork() call works?



# How a fork() call work?



# Understanding fork() call

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

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

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

# Example Program

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

Hello world!  
Hello world!  
Hello world!  
Hello world!

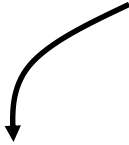
# More fork() calls in a series

```
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
- value (0) is returned in the child

# Using return value of fork() call

```
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!

# Working of fork() - Summary

- **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

# Another example program

```
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

# Example Program

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

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