

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

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.

How a fork() call works?

P_{parent}

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

OS

How a fork() call works?

P_{parent}

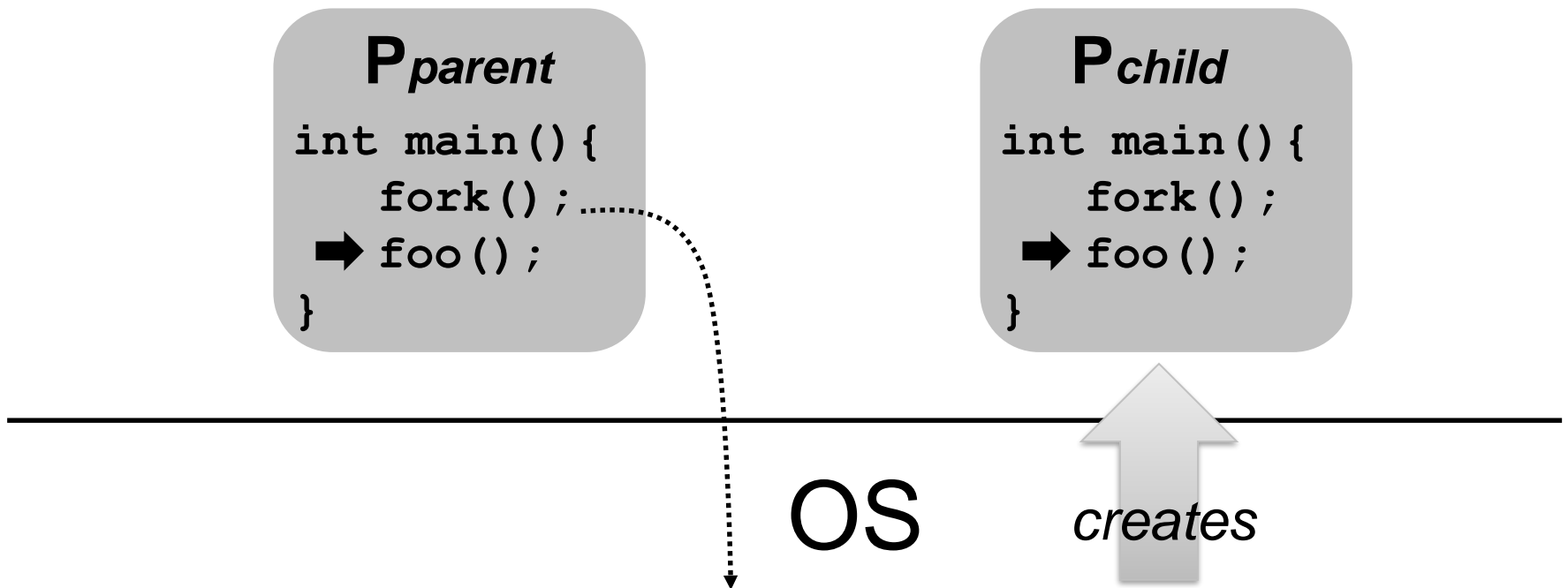
```
int main() {  
    fork();  
    ➡ foo();  
}
```



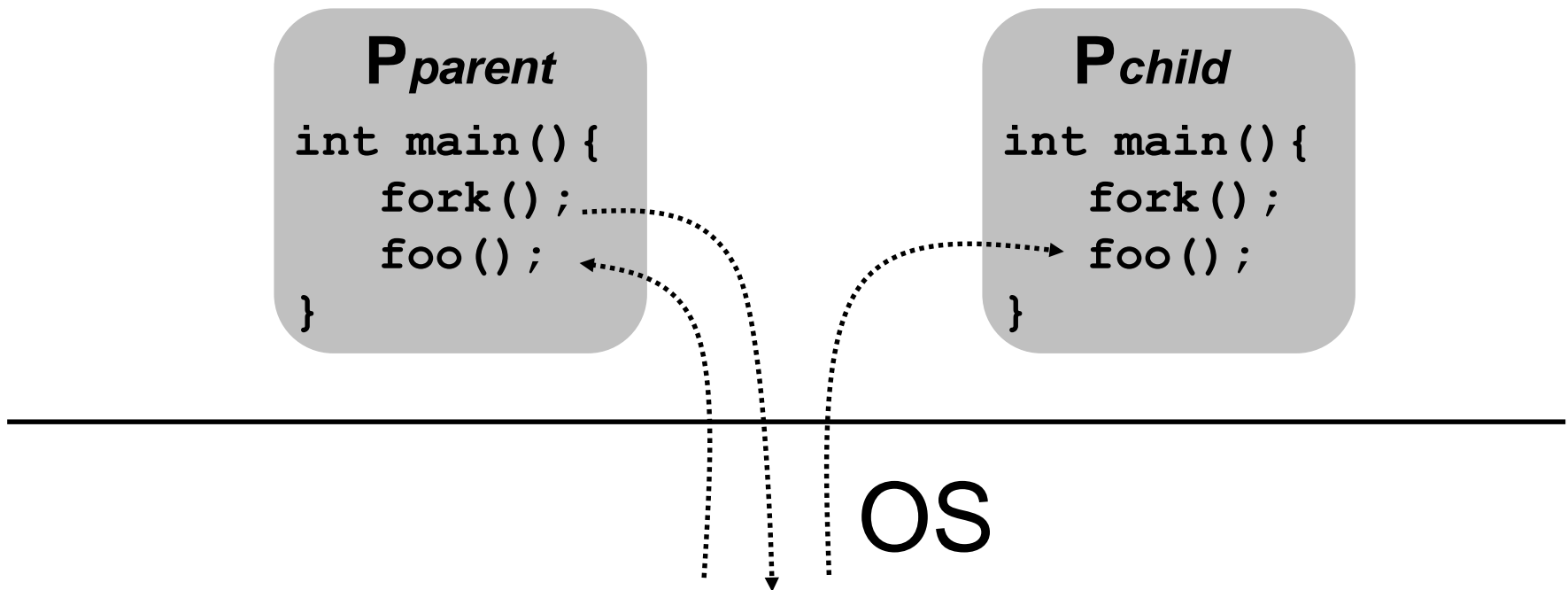
A horizontal line separates the user space from the OS. A dotted arrow originates from the `fork();` line in the parent process code, curves downwards, crosses the horizontal line, and ends with a downward-pointing arrowhead in the OS space.

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");  
}
```


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");  
}
```

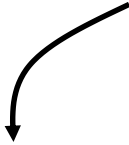
More fork() calls in a series

```
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(); }
```

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);  
    }  
}
```


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");  
}
```

Example Program

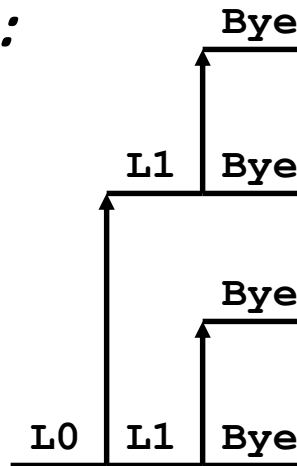
```
void fork2() {  
    printf("L0\n");  
    fork();  
    printf("L1\n");  
    fork();  
    printf("Bye\n");  
}
```

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

Visualizing fork() call As a Process tree

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

process tree:



Check output validity

```
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.

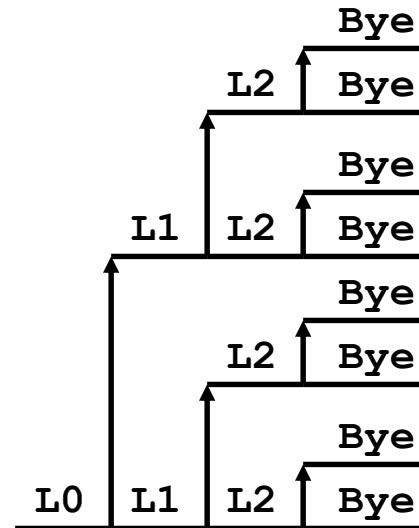
L1
Bye
Bye
L0
L1
Bye
Bye

E.

L0
Bye
Bye
L1
L1
Bye
Bye

Another example program

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



Two example programs

```
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");  
}
```

Check program output validity

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

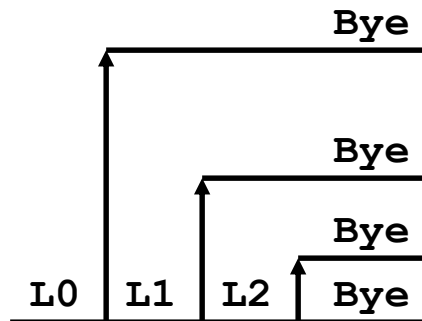
L0
L1
L2
Bye
Bye
Bye
Bye

L0
L1
Bye
Bye
L2
Bye
Bye

Bye
L0
Bye
L1
Bye
L2
Bye

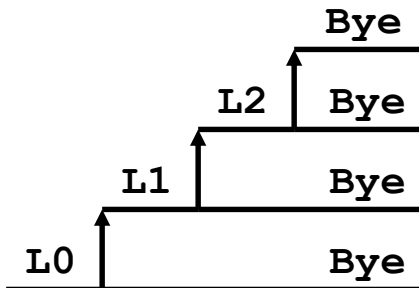
L0
Bye
L1
Bye
L2
Bye
Bye

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");  
}
```

Another example program



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