

## Process - API

What is an API? Application Programming Interface.  
Before getting into Process API, let us try to understand what an API is!

It is a set of protocols to let different pieces of software communicate with each other and share data. Developers use APIs to connect chunks of code from different places in order to create the web and mobile applications we use every day.

For instance, consider the apps like Uber, Ola, Zomato, Swiggy etc..

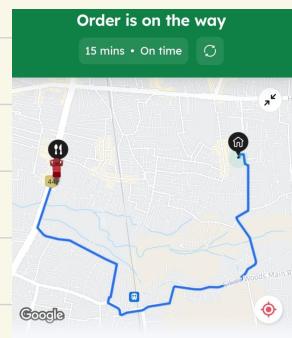
Along with placing a food order/a ride all these apps provides an option to track your order?

How do they provide it?

Do they build their own map systems? No!

Then how? → they use Google maps APIs to provide location-based services.

In particular, in order to show the restaurant location and delivery route to the customer,



Google Maps

- Swiggy sends a request to Google Maps API with source (restaurant) and destination (your home)
- Google maps respond with
  - \* best route
  - \* distance
  - \* estimated delivery time etc...

Benefit: Swiggy can focus on food delivery, not on building complex map software.

Similarly, these apps use Payment API providers like Google Pay / Paytm to integrate online payments without building a payment system from scratch.

### Advantages of using an API:

- \* re-usability → Don't re-invent the wheel
  - saves time and no need to code complex features from scratch.
- \* focus on core business
- \* enable rich features
- \* security

The APIs that we discussed above are generally termed as web APIs. Similarly, there are other types of API like System API, Library API etc..

On this course, we focus on system APIs → the APIs provided by the operating system to let programs interact with hardware and OS services.

e.g. Process API → create/manage process

File API → read/write files

Memory API → manage memory

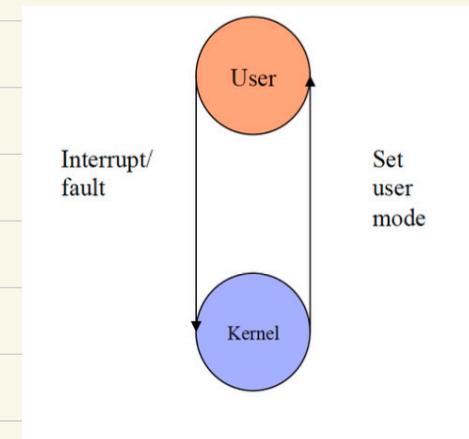
Thread API → work with multiple threads etc..

Goal in this chapter: Process API

Question: Should a process have a direct access to hardware or critical OS resources? NO!

They must request permission through "system calls"

- function call into OS code that runs at a higher privilege level of the CPU.
- sensitive operations (e.g.: access to hardware) are allowed only at a higher privilege level.



(we'll learn more about system calls in the next chapter)

Recall that in the last lecture, we talked about the conceptual steps of process creation.

user

double click on an icon/ type a command to initiate a process



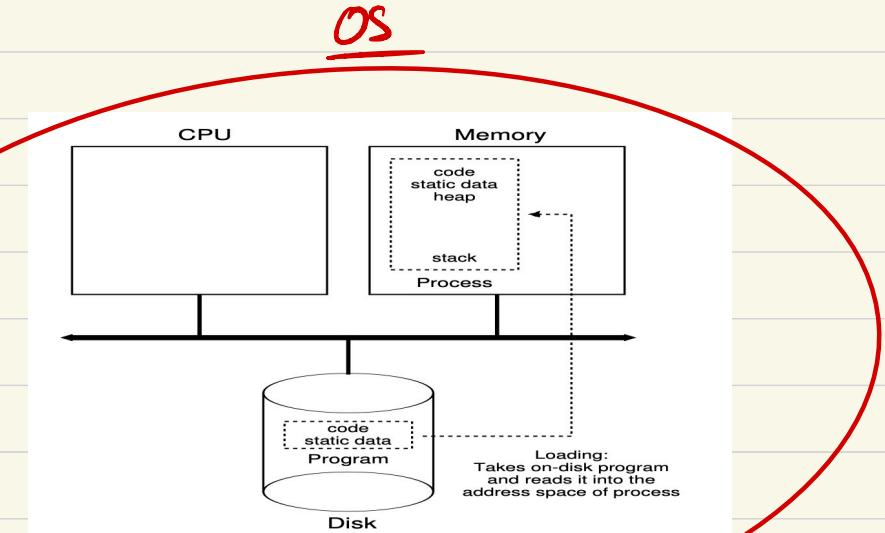
User: customer who wants to order food

Process API

how a user program tells the OS to create a process



indicating a system call: reading the menu and placing an order with waiter



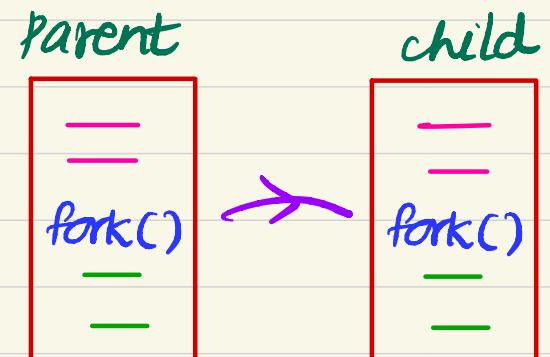
OS implements the steps  
kitchen starts preparing food.

## Process related system calls (in UNIX)

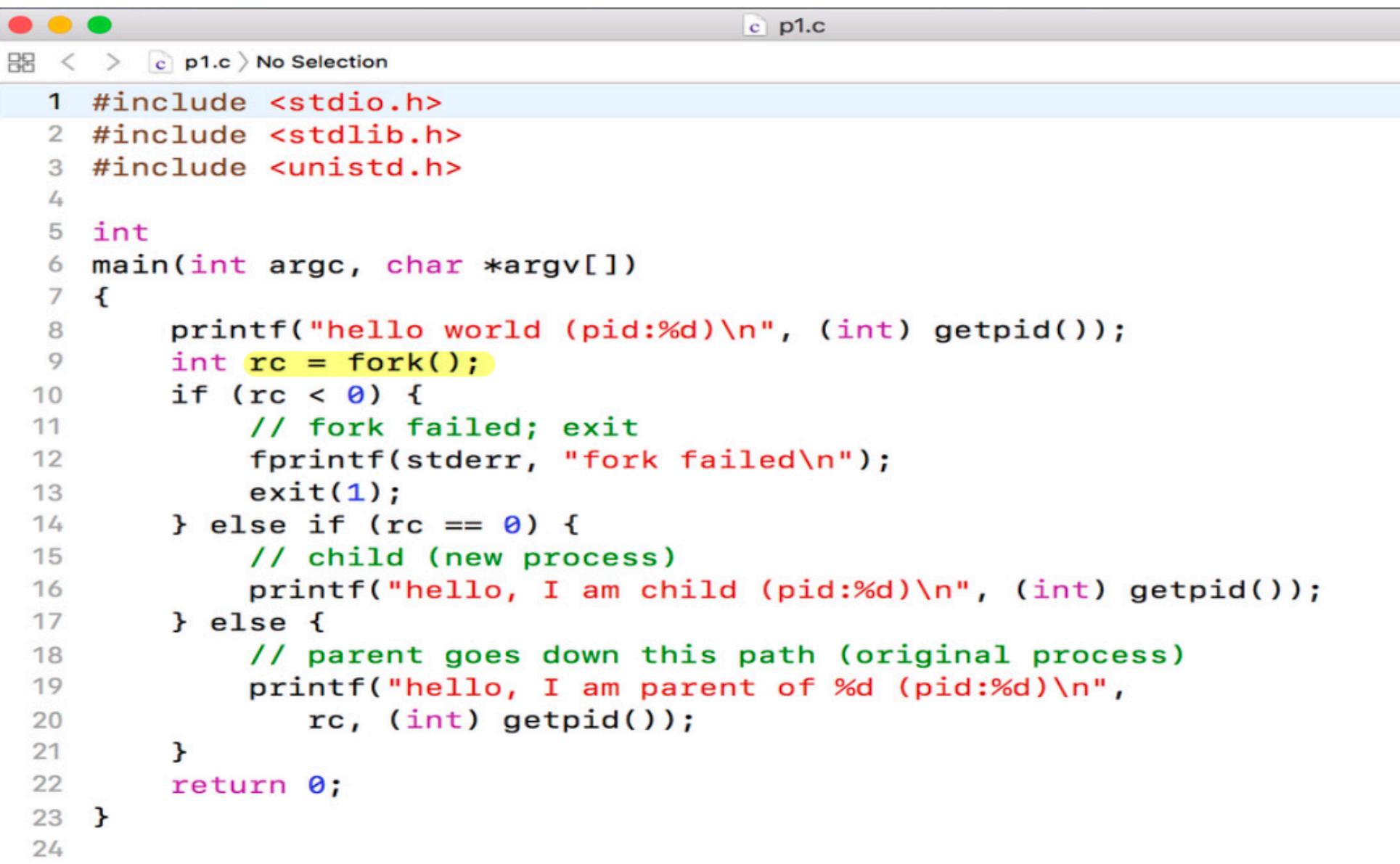
- \* fork() - creates a new child process.
  - all processes are created by forking from a parent who is the ancestor of all the processes? → the init process
- \* exec() - makes a process execute a given executable
- \* wait() - causes a parent to block until child terminates  
many variants of above also exists.

### The fork() system call

- \* a new process is created by making a copy of parent's memory image.
- Q. after calling fork(), you have two processes.
  - the parent process (original)
  - the child process (newly created)



- \* the child process gets a unique process ID and gets added to the OS process list and gets scheduled (i.e., ready to run)
- \* Note that the child has
  - the same code
  - a copy of the memory (stack, heap, data, etc.)
  - but a separate independent address space.  
(so changes in child do not affect the parent and vice-versa)
- \* parent and child will execute the code after the fork() call but with different return values.  
In particular, fork() returns
  - \* 0 to the child process
  - \* child's PID (as integer) to the parent
  - \* -1 if fork() failed.



```
p1.c
p1.c > No Selection

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4
5 int
6 main(int argc, char *argv[])
7 {
8     printf("hello world (pid:%d)\n", (int) getpid());
9     int rc = fork();
10    if (rc < 0) {
11        // fork failed; exit
12        fprintf(stderr, "fork failed\n");
13        exit(1);
14    } else if (rc == 0) {
15        // child (new process)
16        printf("hello, I am child (pid:%d)\n", (int) getpid());
17    } else {
18        // parent goes down this path (original process)
19        printf("hello, I am parent of %d (pid:%d)\n",
20               rc, (int) getpid());
21    }
22    return 0;
23 }
```

## Result (Not deterministic)

```
prompt> ./p1
hello world (pid:29146)
hello, I am parent of 29147 (pid:29146)
hello, I am child (pid:29147)
prompt>
```

or

```
prompt> ./p1
hello world (pid:29146)
hello, I am child (pid:29147)
hello, I am parent of 29147 (pid:29146)
prompt>
```

Depending on  
which among the  
parent and the  
child processes gets  
scheduled first!

Relation b/w parent and child processes: the wait() system call

Different types of process  
termination scenarios

→ by calling exit()  
→ os killed the process

} terminated  
process exists as  
zombies

why zombie? - it is dead, but still occupies an entry in the process table.

How to clean up zombies? → parent calls wait().  
or reap



\* What if the parent terminates before child?

The ancestor of all process will adopts orphans and reap them.

```
● ● ● p2.c No Selection
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4 #include <sys/wait.h>
5
6 int
7 main(int argc, char *argv[])
8 {
9     printf("hello world (pid:%d)\n", (int) getpid());
10    int rc = fork();
11    if (rc < 0) {
12        // fork failed; exit
13        fprintf(stderr, "fork failed\n");
14        exit(1);
15    } else if (rc == 0) {
16        // child (new process)
17        printf("hello, I am child (pid:%d)\n", (int) getpid());
18        sleep(1);
19    } else {
20        // parent goes down this path (original process)
21        int wc = wait(NULL);
22        printf("hello, I am parent of %d (wc:%d) (pid:%d)\n",
23               rc, wc, (int) getpid());
24    }
25    return 0;
26 }
```

→ on addition to the previous code, here the parent is calling wait().

Do you expect any difference in the output compared to before?

YES!

Result (Deterministic)

```
prompt> ./p2
hello world (pid:29266)
hello, I am child (pid:29267)
hello, I am parent of 29267 (wc:29267) (pid:29266)
prompt>
```

Adding a `wait()` call to the code above makes the output deterministic.

In particular, the child will print first:

even if the parent runs first, it ~~partly~~ wait for the child to finish running, then `wait()` returns, and then the parent resumes.

### The `exec()` system call

↳ an important piece of the process creation API  
after `fork()` → parent and child are running the same code.  
→ which is not very useful!

\* a process run `exec()` → to load another executable to its memory image → a child run a different program from parent.

If you only use `exec()` → the process doesn't change its PID,  
but new code, stack and heap.

Thus, if you don't `fork()` before `exec()`,

your current process will be overwritten.

exec() + fork() → very powerful. ↗ a variant of exec()

In the full example, the child process calls execvp() in order to run the program wc (word count program, telling us how many words, lines, bytes, etc. are the input file, which here is the source file p3.c itself)

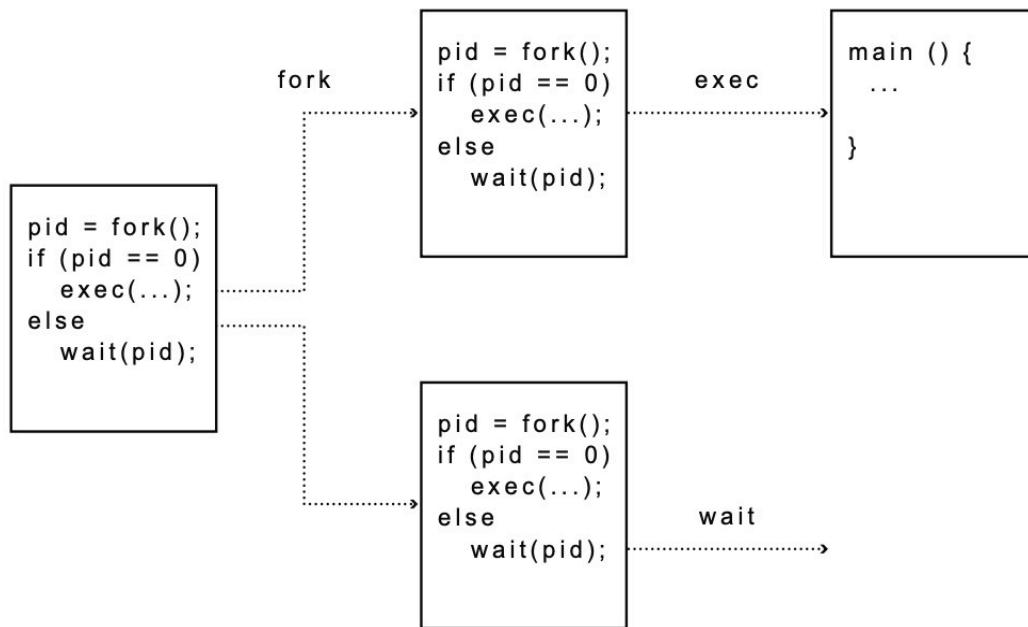
```
● ○ ●
p3.c < > p3.c No Selection
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4 #include <string.h>
5 #include <sys/wait.h>
6
7 int
8 main(int argc, char *argv[])
9 {
10     printf("hello world (pid:%d)\n", (int) getpid());
11     int rc = fork();
12     if (rc < 0) {
13         // fork failed; exit
14         fprintf(stderr, "fork failed\n");
15         exit(1);
16     } else if (rc == 0) {
17         // child (new process)
18         printf("hello, I am child (pid:%d)\n", (int) getpid());
19         char *myargs[3];
20         myargs[0] = strdup("wc");    // program: "wc" (word count)
21         myargs[1] = strdup("p3.c"); // argument: file to count
22         myargs[2] = NULL;          // marks end of array
23         execvp(myargs[0], myargs); // runs word count
24         printf("this shouldn't print out");
25     } else {
26         // parent goes down this path (original process)
27         int wc = wait(NULL);
28         printf("hello, I am parent of %d (wc:%d) (pid:%d)\n",
29                rc, wc, (int) getpid());
30     }
31     return 0;
32 }
```

Basically,  
exec() (if executed successfully),  
it transforms  
the currently  
running program  
(formerly p3)  
into a different  
running program  
(wc).

## Result

```
prompt> ./p3
hello world (pid:29383)
hello, I am child (pid:29384)
29 107 1030 p3.c
hello, I am parent of 29384 (wc:29384) (pid:29383)
prompt>
```

## Summary of fork(), wait(), exec()



Read: Chapter-5  
OSEP