Process - API

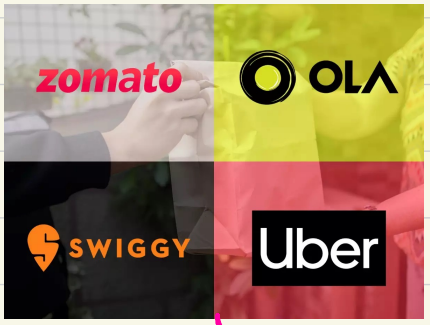
What is an API? Application Programming Interface.

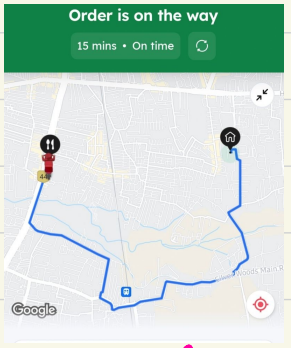
Before getting into Process API, let us try to understand what an API is!

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?

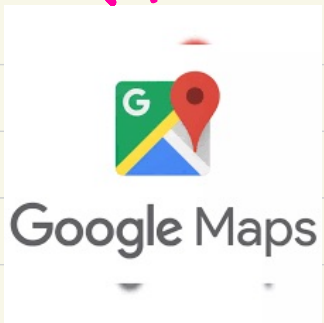




How do they provide it?

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,

- 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 bussiness
* enable rich features
* security.

The API's that we discussed above are generally termed as web API's. 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.

eg:- Process API → create/manage process

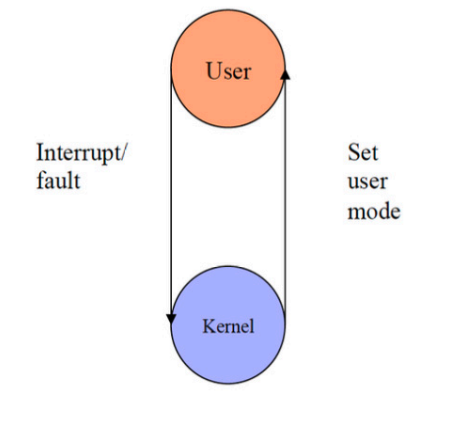
File API → read/write files
Memory API → manage memory
thread API → work with multiple threads etc...

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 (eg:- access to hardware) are allowed only at a higher privilege level.



(will learn more about system calls in the next chapter)

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

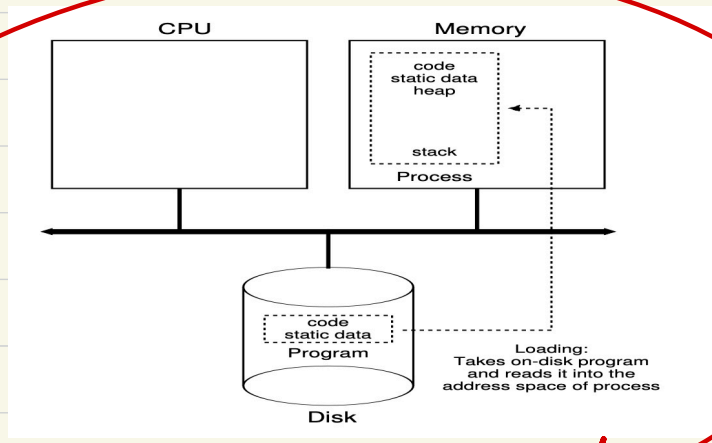
User

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

Process API

how a user program tells the OS to create a process

OS



what OS does to create a process?

User: customer who wants to order food

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

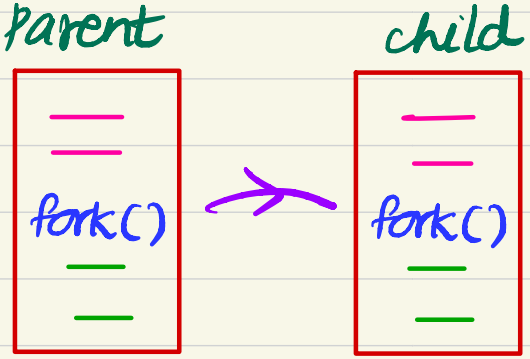
OS implements the steps kitchen starts preparing food.

Process related system calls (in UNIX)

* fork() - creates a new child process.
  - all process are created by forking from a parent
    who is the ancestor of all the process? → 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.



i.e., 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 (+ve integer) to the parent
  * -1 if fork() failed.

p1.c

p1.c > No Selection

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int
main(int argc, char *argv[])
{
    printf("hello world (pid:%d)\n", (int) getpid());
    int rc = fork();
    if (rc < 0) {
        // fork failed; exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    } else if (rc == 0) {
        // child (new process)
        printf("hello, I am child (pid:%d)\n", (int) getpid());
    } else {
        // parent goes down this path (original process)
        printf("hello, I am parent of %d (pid:%d)\n",
           rc, (int) getpid());
    }
    return 0;
}

```

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 init adopts orphans and reap them.

p2.c

p2.c > No Selection

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

int
main(int argc, char *argv[])
{
    printf("hello world (pid:%d)\n", (int) getpid());
    int rc = fork();
    if (rc < 0) {
        // fork failed; exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    } else if (rc == 0) {
        // child (new process)
        printf("hello, I am child (pid:%d)\n", (int) getpid());
    sleep(1);
    } else {
        // parent goes down this path (original process)
        int wc = wait(NULL);
        printf("hello, I am parent of %d (wc:%d) (pid:%d)\n",
            rc, wc, (int) getpid());
    }
    return 0;
}

```

→ In 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 politely 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

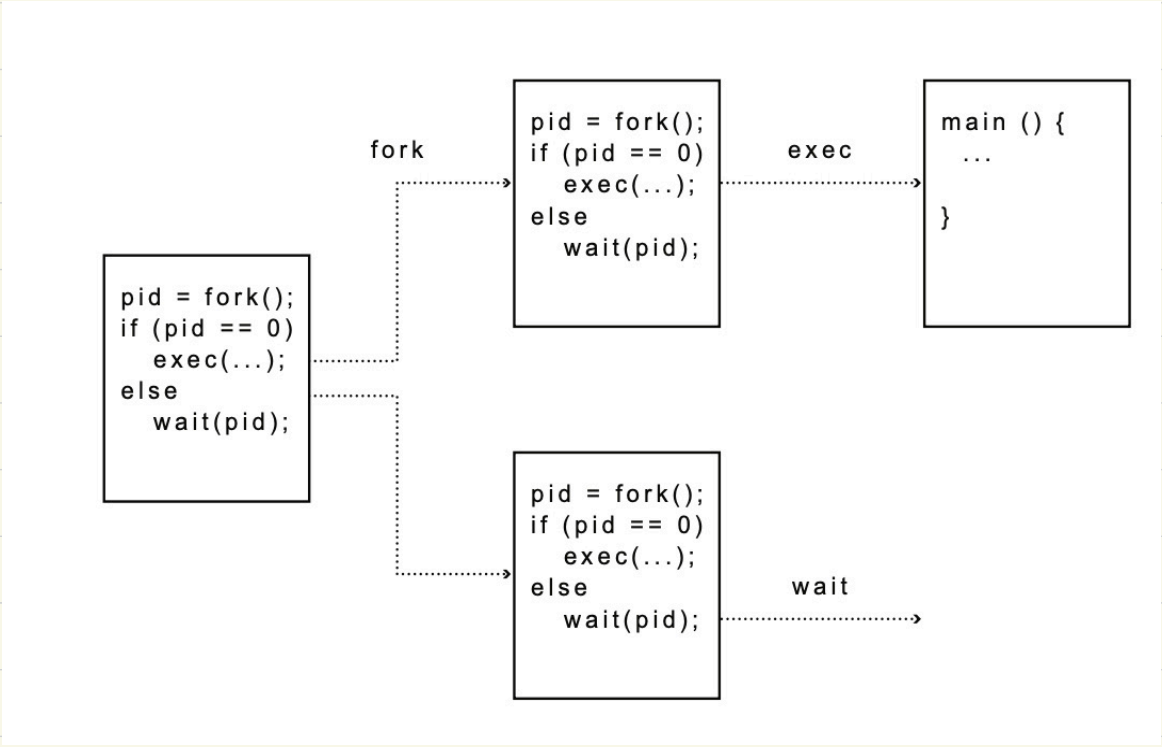
```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/wait.h>

int
main(int argc, char *argv[])
{
    printf("hello world (pid:%d)\n", (int) getpid());
    int rc = fork();
    if (rc < 0) {
        // fork failed; exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    } else if (rc == 0) {
        // child (new process)
        printf("hello, I am child (pid:%d)\n", (int) getpid());
        char *myargs[3];
        myargs[0] = strdup("wc");   // program: "wc" (word count)
        myargs[1] = strdup("p3.c"); // argument: file to count
        myargs[2] = NULL;           // marks end of array
        execvp(myargs[0], myargs);  // runs word count
        printf("this shouldn't print out");
    } else {
        // parent goes down this path (original process)
        int wc = wait(NULL);
        printf("hello, I am parent of %d (wc:%d) (pid:%d)\n",
            rc, wc, (int) getpid());
    }
    return 0;
}
```

Basically,
exec() (if executed succesfully),
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 OSTEP