C++ Software Engineering

for engineers of other disciplines

Module 10

"C++ Parallelism"

4th Lecture: Multiprocessing

ALTEN

Spring 2022

Gothenburg, Sweden

Process

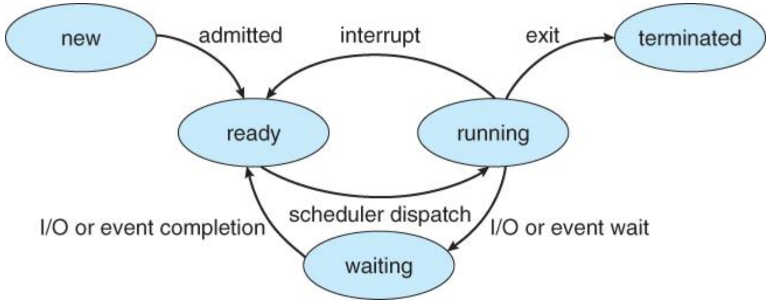
- Process is a running executable which consumes resources *through* operating system interfaces – both APIs and ABIs.

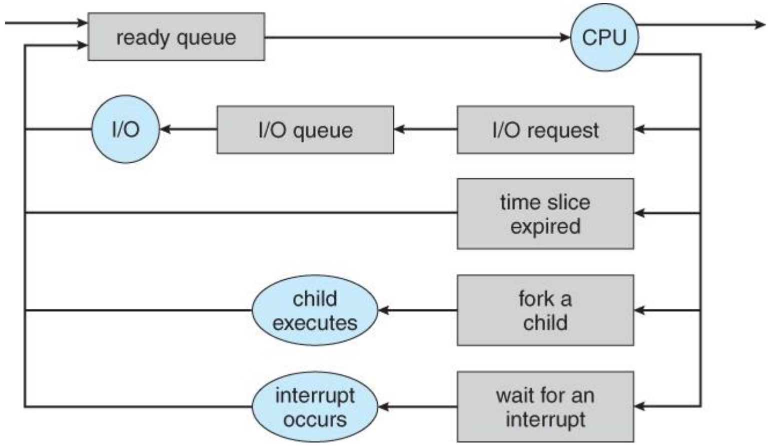
- Unix systems inherit **sysV** APIs.

- **POSIX** is an effort by IEEE to standardized OS interfaces for the purpose of portability.

- Processes can execute instructions, request I/O, wait for an interrupt or spawn a child.

> *In computer software, an application binary interface (ABI) is an interface between two binary program modules; often, one of these modules is a library or operating system facility, and the other is a program that is being run by a user.* https://en.wikipedia.org/wiki/Application_binary_interface



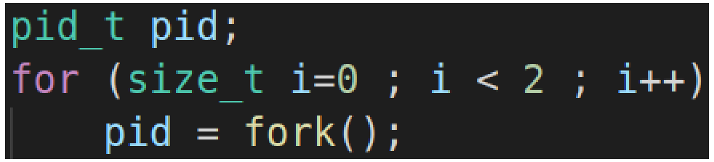


https://www.cs.uic.edu/~jbell/CourseNotes/OperatingSystems/3_Processes.html

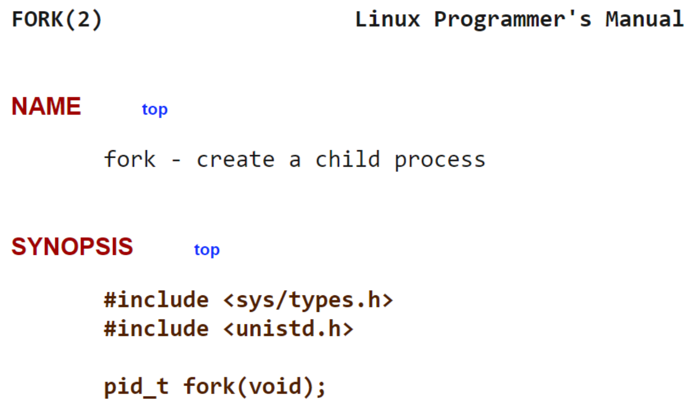
Fork

- Child process is a copy of its parent except:
  - It has its own unique *process **identification*** a.k.a. **pid**.
  - Its parents **pid** is different – obviously!
  - It is a fresh copy! (empty ques, *reseted* flags for *locks,* timers, fcntl, …)
- `fork` is the function to create a child process.
- Children should report their return status to the parent, or they become a zombie process – it is the parent responsibility to `wait` for its children.
- Defunct process are those which have been terminated but their *exit status* has not been read by their parents – POSIX provides *automatic reaping* mechanism.

```
signal(SIGCHLD, SIG_IGN);
```



```
pid_t pid;
for (size_t i=0 ; i < 2 ; i++)
    pid = fork();
```



```
FORK(2)                    Linux Programmer's Manual
```

NAME top

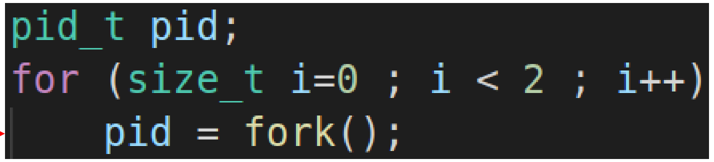
```
       fork - create a child process
```

SYNOPSIS top

```
       #include <sys/types.h>
       #include <unistd.h>

       pid_t fork(void);
```

> *"The entire virtual address space of the parent is replicated in the child, including the states of mutexes, condition variables, and other pthreads objects; the use of* **pthread_atfork** *may be helpful for dealing with problems that this can cause."* https://man7.org/linux/man-pages/man2/fork.2.html

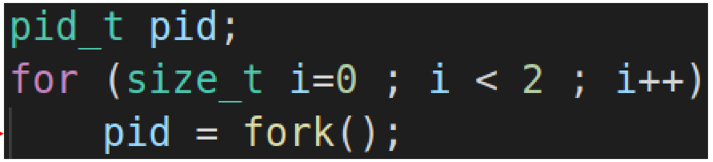
Fork



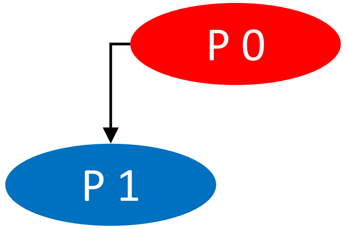
```c
pid_t pid;
for (size_t i=0 ; i < 2 ; i++)
    pid = fork();
```

- Child processes spawned using **`fork`** returns zero on the child, the returns the child's pid on the parent call.

Fork

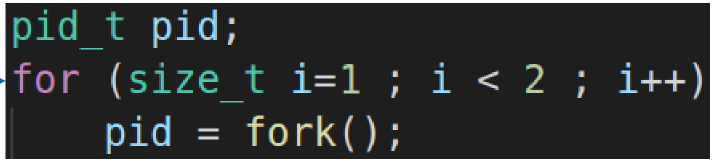


```
pid_t pid;
for (size_t i=0 ; i < 2 ; i++)
    pid = fork();
```

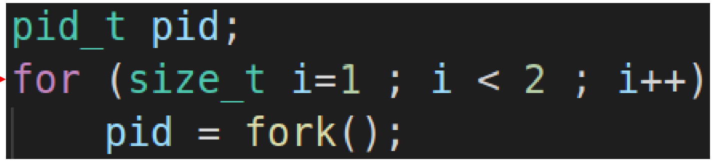


- A *fresh* copy of the process with the exact content of the virtual memory is *prepared* for the child.

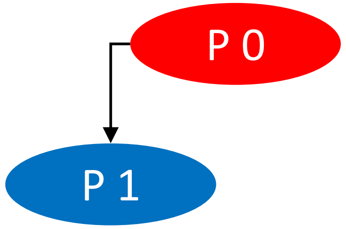
Fork



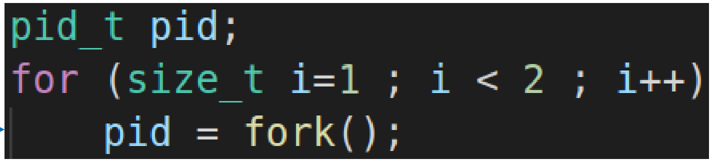
```
pid_t pid;
for (size_t i=1 ; i < 2 ; i++)
    pid = fork();
```



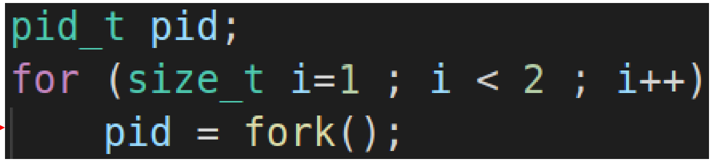
```
pid_t pid;
for (size_t i=1 ; i < 2 ; i++)
    pid = fork();
```



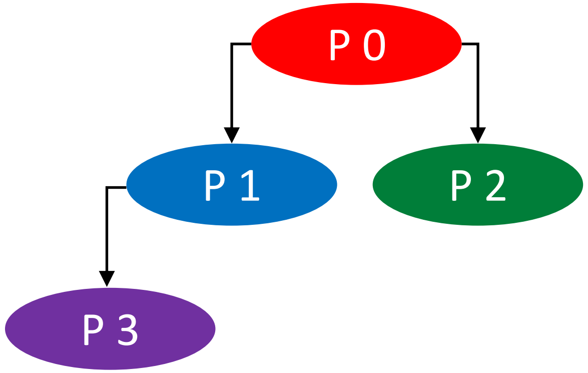
Fork



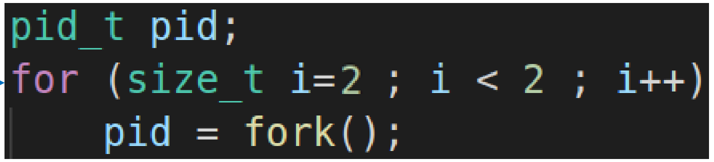
```
pid_t pid;
for (size_t i=1 ; i < 2 ; i++)
    pid = fork();
```



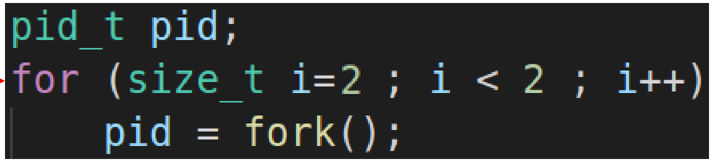
```
pid_t pid;
for (size_t i=1 ; i < 2 ; i++)
    pid = fork();
```



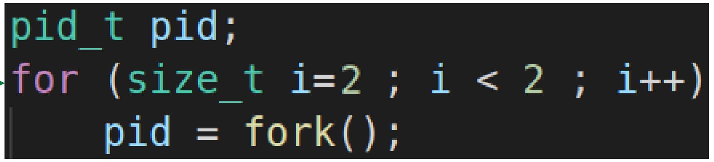
Fork



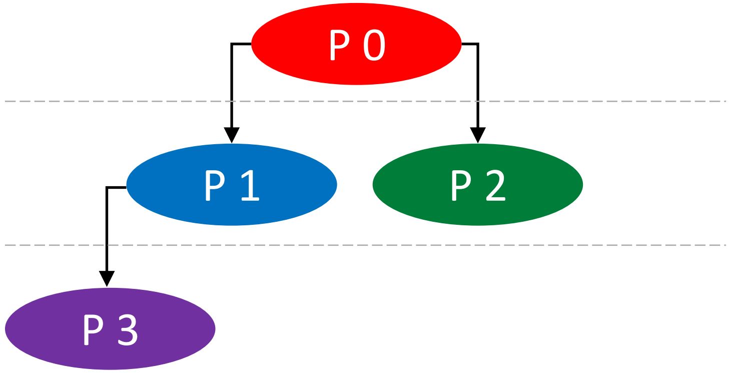
```
pid_t pid;
for (size_t i=2 ; i < 2 ; i++)
    pid = fork();
```



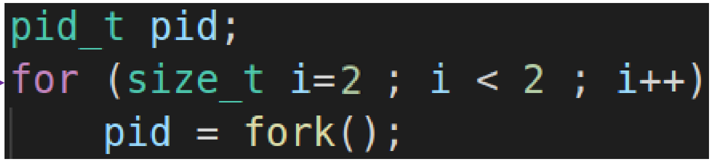
```
pid_t pid;
for (size_t i=2 ; i < 2 ; i++)
    pid = fork();
```



```
pid_t pid;
for (size_t i=2 ; i < 2 ; i++)
    pid = fork();
```



Process Tree -- 2^n

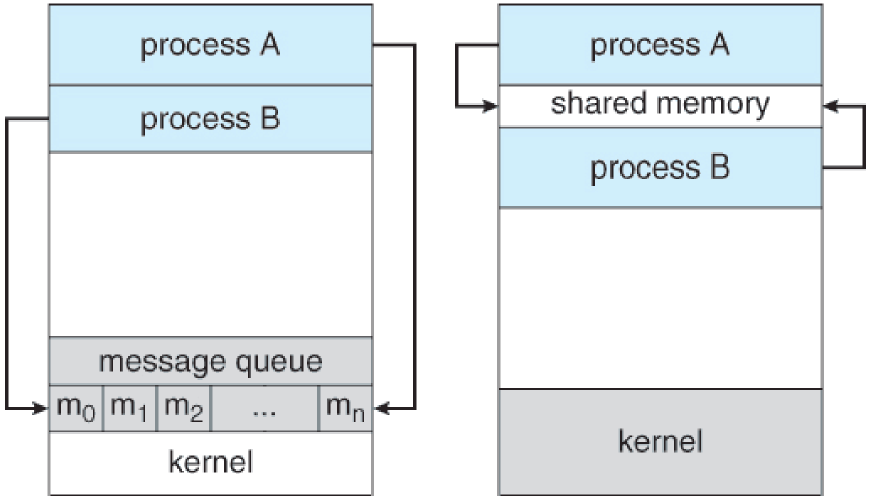


```
pid_t pid;
for (size_t i=2 ; i < 2 ; i++)
    pid = fork();
```

- Child processes spawned using **`fork`** will form a tree – children's return exit shall be read by their parents.

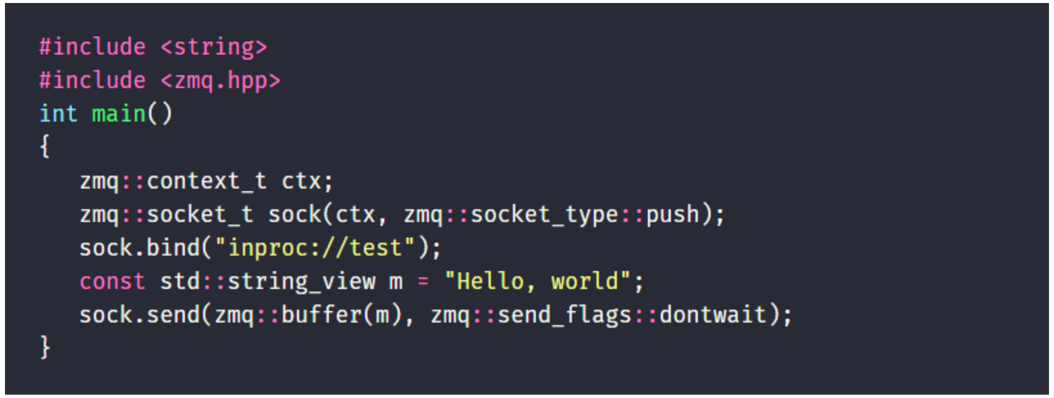
Inter Process Communication

- IPC is the mechanism used to communicate between processes.
- Process can *signal* each other or *share* data with each other.
- This communication is in Operating System *realm.*
- There are libraries and framework who have implemented IPC.



(a) Message Passing (b) Shared Memory

https://networkencyclopedia.com/wp-content/uploads/2019/09/inter-process-communications-models.png



```
#include <string>
#include <zmq.hpp>
int main()
{
   zmq::context_t ctx;
   zmq::socket_t sock(ctx, zmq::socket_type::push);
   sock.bind("inproc://test");
   const std::string_view m = "Hello, world";
   sock.send(zmq::buffer(m), zmq::send_flags::dontwait);
}
```

https://zeromq.org/languages/cplusplus/

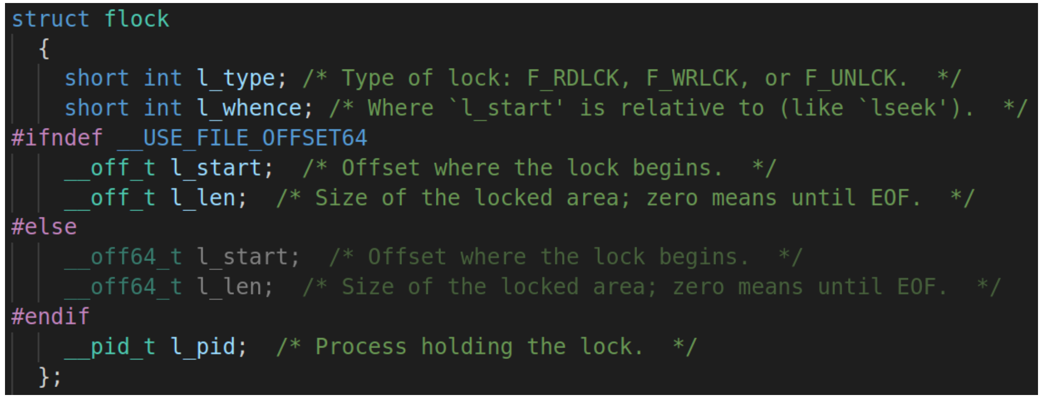
boost
C++ LIBRARIES

Chapter 18. Boost.Interprocess

https://www.boost.org/doc/libs/1_74_0/doc/html/interprocess.html

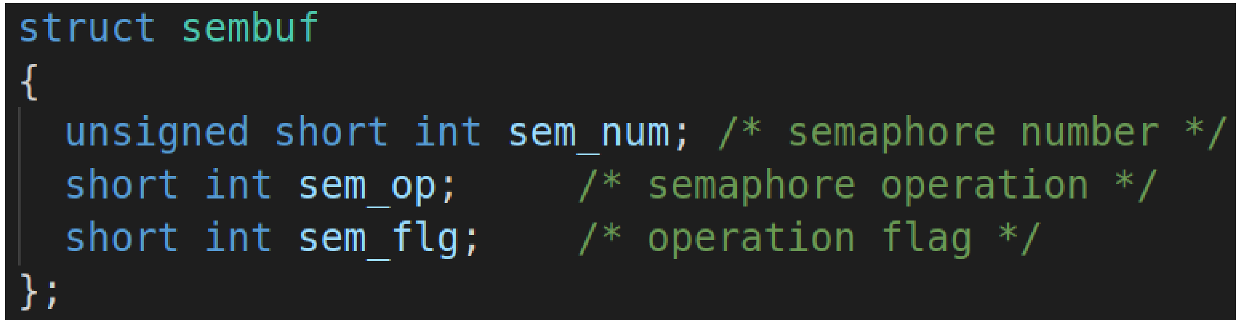
IPC

- Anything involving kernel is slow: message queues, signals, and etc. Pipes usually are used to redirect input outputs to each other, FIFO (or file pipes?) are named pipes.

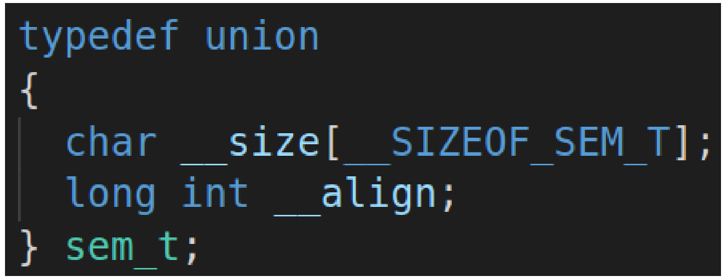


```c
struct flock
  {
    short int l_type; /* Type of lock: F_RDLCK, F_WRLCK, or F_UNLCK.  */
    short int l_whence; /* Where `l_start' is relative to (like `lseek').  */
#ifndef __USE_FILE_OFFSET64
    __off_t l_start;  /* Offset where the lock begins.  */
    __off_t l_len;  /* Size of the locked area; zero means until EOF.  */
#else
    __off64_t l_start;  /* Offset where the lock begins.  */
    __off64_t l_len;  /* Size of the locked area; zero means until EOF.  */
#endif
    __pid_t l_pid;  /* Process holding the lock.  */
  };
```

- Resources used between processes needs synchronization.
- Many IPCs happen through *fd*s – they need to lock!
- The best is to share memory between processes:
  - Some could argue Sockets are good as well!
  - A shared memory between processes is almost identical to the process's memory – zero overhead!
- Both ***sysV*** and ***POSIX*** provide API for that – semaphore is used for resource synchronization over shared memory.



```c
struct sembuf
{
  unsigned short int sem_num; /* semaphore number */
  short int sem_op;     /* semaphore operation */
  short int sem_flg;    /* operation flag */
};
```

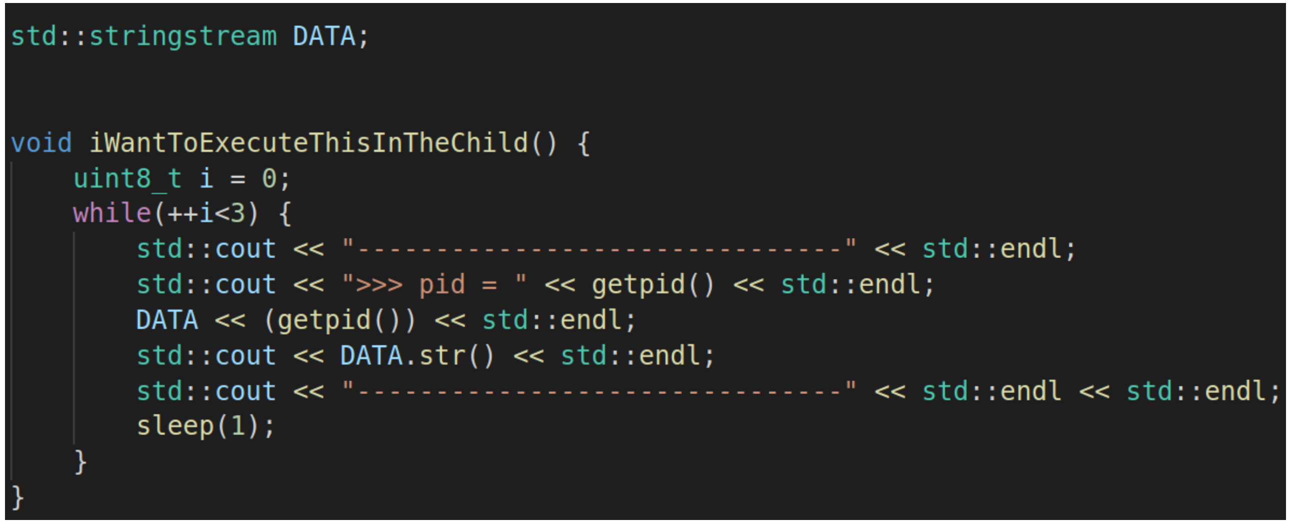


```c
typedef union
{
  char __size[__SIZEOF_SEM_T];
  long int __align;
} sem_t;
```

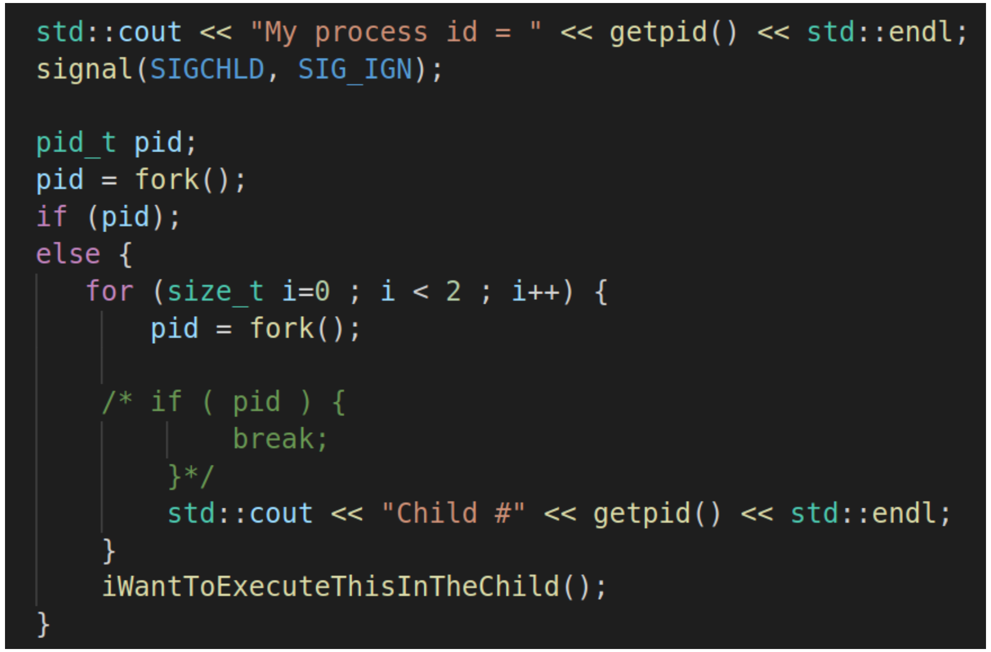
DEMO!

Fork

© M. Rashid Zamani

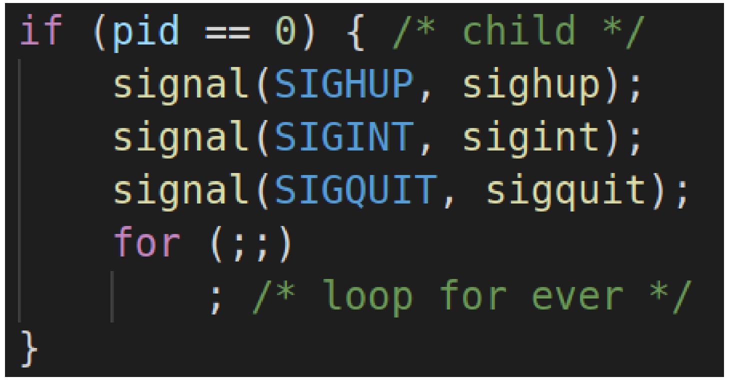


```
std::stringstream DATA;


void iWantToExecuteThisInTheChild() {
    uint8_t i = 0;
    while(++i<3) {
        std::cout << "-------------------------------" << std::endl;
        std::cout << ">>> pid = " << getpid() << std::endl;
        DATA << (getpid()) << std::endl;
        std::cout << DATA.str() << std::endl;
        std::cout << "-------------------------------" << std::endl << std::endl;
        sleep(1);
    }
}
```

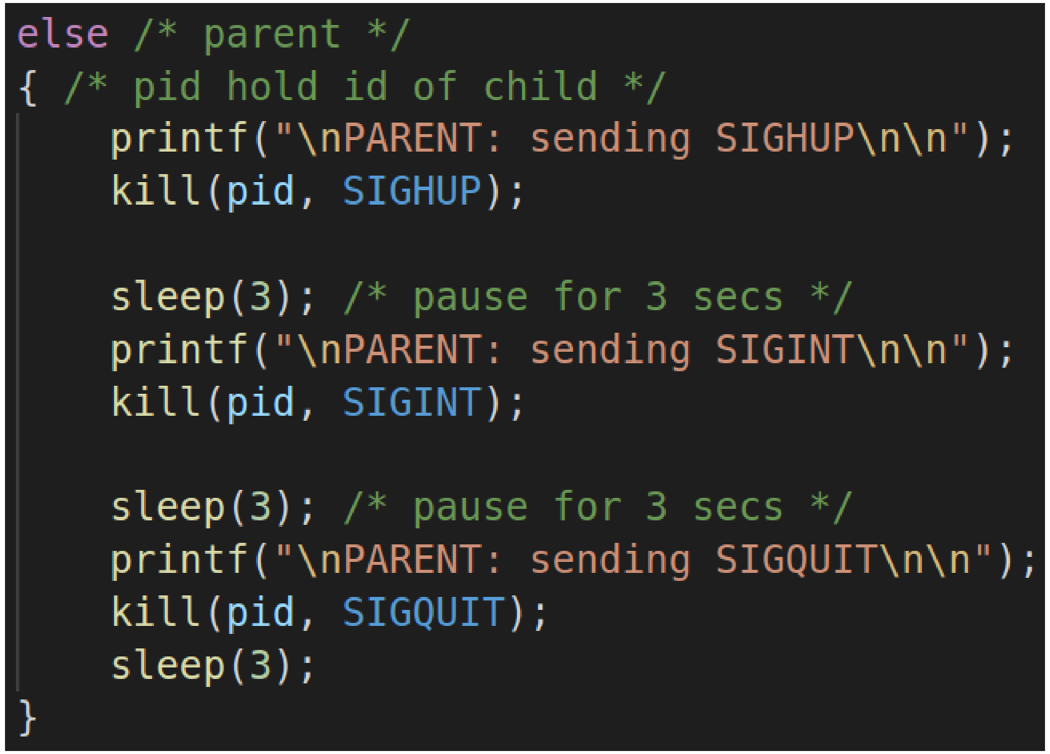


```
std::cout << "My process id = " << getpid() << std::endl;
signal(SIGCHLD, SIG_IGN);

pid_t pid;
pid = fork();
if (pid);
else {
   for (size_t i=0 ; i < 2 ; i++) {
       pid = fork();

       /* if ( pid ) {
               break;
           }*/
           std::cout << "Child #" << getpid() << std::endl;
   }
   iWantToExecuteThisInTheChild();
}
```

Signal

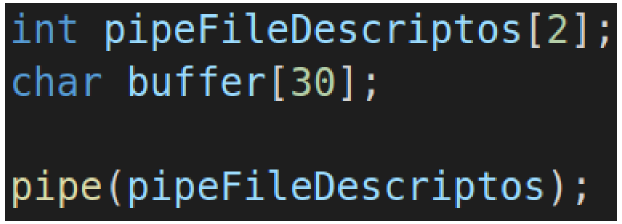


```
if (pid == 0) { /* child */
    signal(SIGHUP, sighup);
    signal(SIGINT, sigint);
    signal(SIGQUIT, sigquit);
    for (;;)
        ; /* loop for ever */
}
```

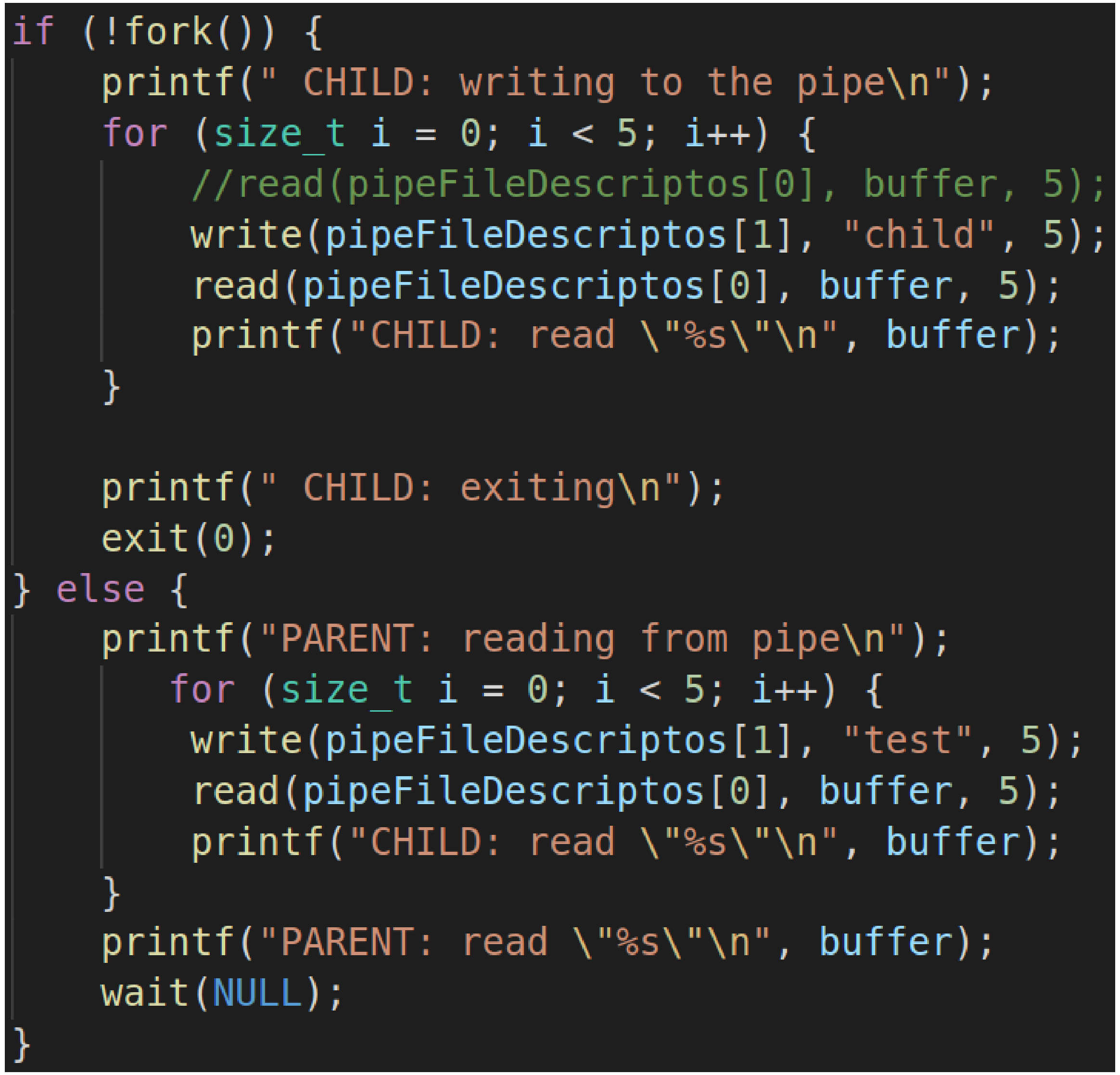


```
else /* parent */
{ /* pid hold id of child */
    printf("\nPARENT: sending SIGHUP\n\n");
    kill(pid, SIGHUP);

    sleep(3); /* pause for 3 secs */
    printf("\nPARENT: sending SIGINT\n\n");
    kill(pid, SIGINT);

    sleep(3); /* pause for 3 secs */
    printf("\nPARENT: sending SIGQUIT\n\n");
    kill(pid, SIGQUIT);
    sleep(3);
}
```

Pipe

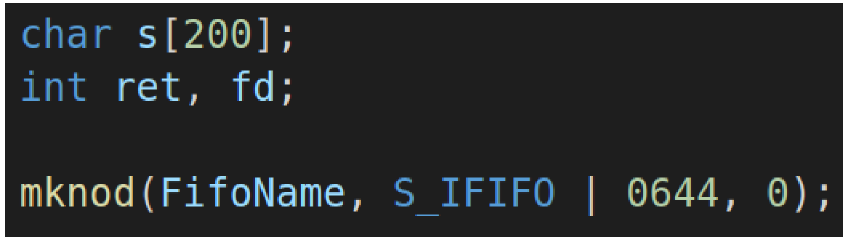


```c
int pipeFileDescriptos[2];
char buffer[30];

pipe(pipeFileDescriptos);
```

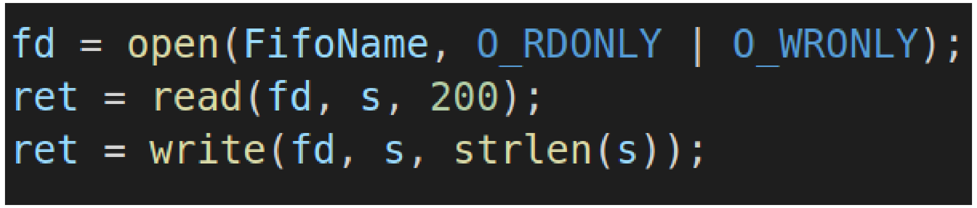


```c
if (!fork()) {
    printf(" CHILD: writing to the pipe\n");
    for (size_t i = 0; i < 5; i++) {
        //read(pipeFileDescriptos[0], buffer, 5);
        write(pipeFileDescriptos[1], "child", 5);
        read(pipeFileDescriptos[0], buffer, 5);
        printf("CHILD: read \"%s\"\n", buffer);
    }

    printf(" CHILD: exiting\n");
    exit(0);
} else {
    printf("PARENT: reading from pipe\n");
       for (size_t i = 0; i < 5; i++) {
        write(pipeFileDescriptos[1], "test", 5);
        read(pipeFileDescriptos[0], buffer, 5);
        printf("CHILD: read \"%s\"\n", buffer);
    }
    printf("PARENT: read \"%s\"\n", buffer);
    wait(NULL);
}
```

FIFO



```c
char s[200];
int ret, fd;

mknod(FifoName, S_IFIFO | 0644, 0);
```



```c
fd = open(FifoName, O_RDONLY | O_WRONLY);
ret = read(fd, s, 200);
ret = write(fd, s, strlen(s));
```