Part 2
Lecture 2: More on Processes

Review From Last Lecture (1)

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

int main()
{
    pid_t value; // process identification used to represent process id
    value = fork(); //call the fork function to start a separate process
    printf("In main: value =%d\n", value); //print the id of the process
}
```

```
kaleel@CentralCompute:~$ gcc test.c -o test
kaleel@CentralCompute:~$ ./test
In main: value =809
In main: value =0
```

- We discussed the idea of processes and the fork function to create multiple processes.
- When fork is called in the parent code, a child is created which is a clone of the parent. The child starts running AFTER the line where the fork function was called.

Review From Last Lecture (2)

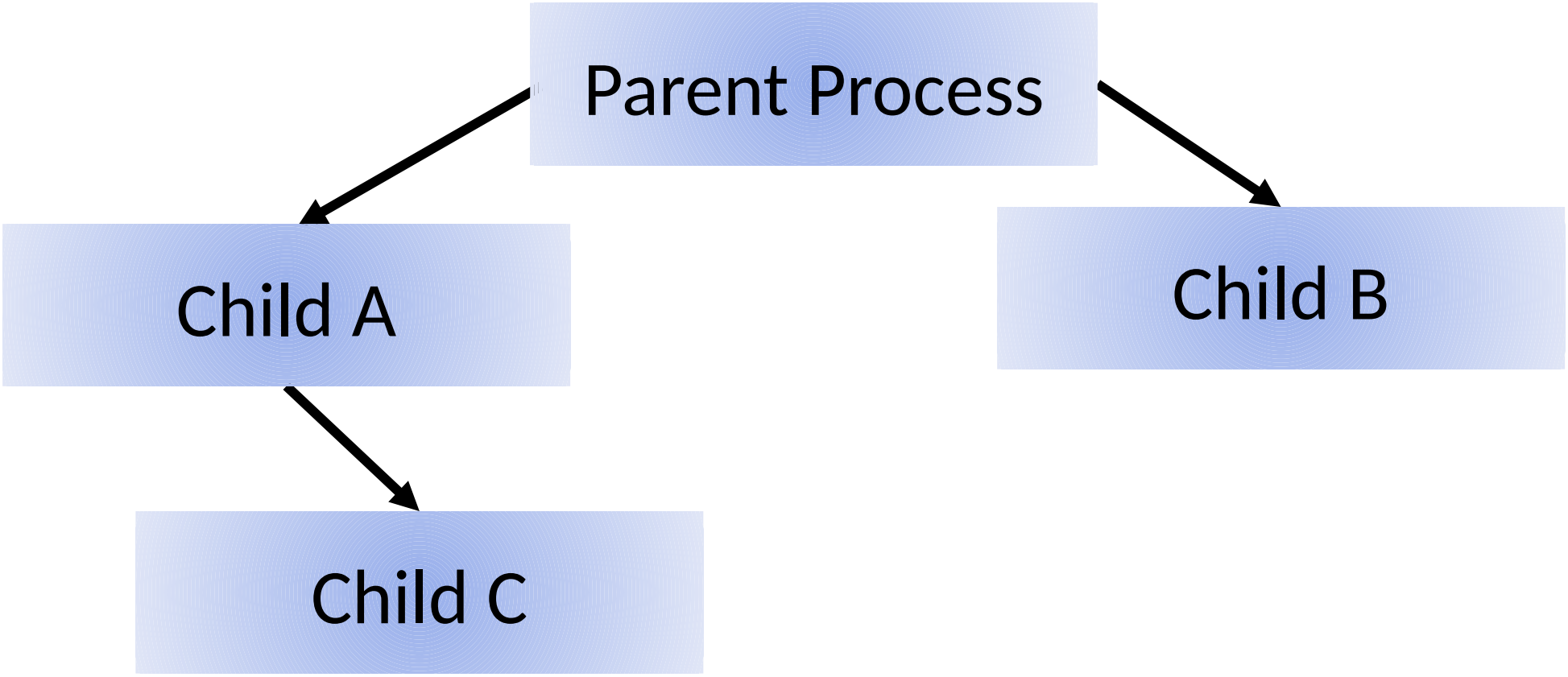
- When we create processes, we usually do so with the goal of making each process run some DIFFERENT piece of code.
- Essentially, we want processes to run in parallel.
- How did we make the parent and child run different tasks? Using the process ID.

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

int main()
{
    pid_t value; // process identification used to represent process id
    value = fork(); //call the fork function to start a separate process
    if(value != 0)//this means we are in the parent process
    {
        printf("Let's do the first load of laundry.");
    }
    else if(value == 0) //this means we are the child process
    {
        printf("Let's do the second load of laundry.");
    }
}
```

Review From Last Lecture (3)

How to keep track of children and parents if you want to call fork multiple times?

- Short Answer: It is not pretty.



How to keep track of children and parents if you want to call fork multiple times?

- Short Answer: It is not pretty.

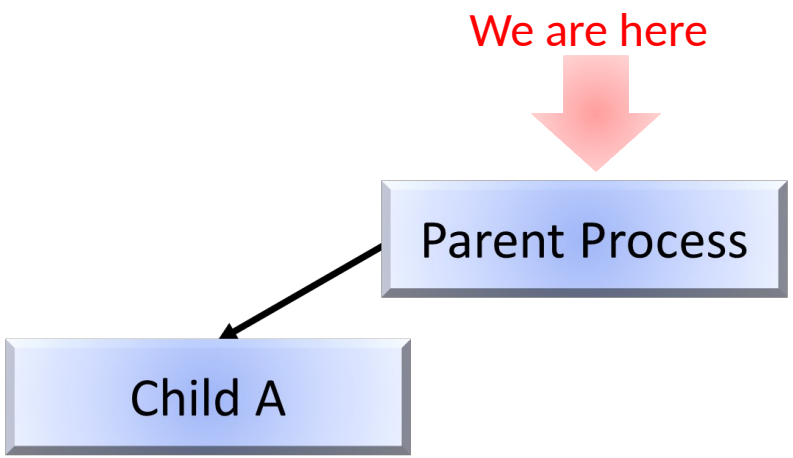
```c
#include <stdio.h>
#include <unistd.h>

int main(){
    pid_t value; //process identification used to represent process id
    value = fork();
    if(value != 0 ){ //in the parent process
```

```c
    else if(value == 0){
```

How to keep track of children and parents if you want to call fork multiple times?

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

int main(){
    pid_t value; //process identification used to represent process id
    value = fork();
    if(value != 0 ){ //in the parent process
```

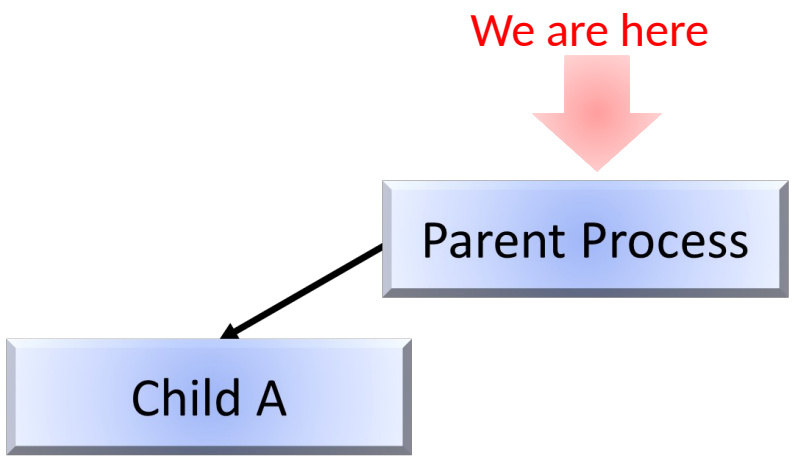
```c
    }
    else if(value == 0){
```



How to keep track of children and parents if you want to call fork multiple times?

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

int main(){
    pid_t value; //process identification used to represent process id
    value = fork();
    if(value != 0 ){ //in the parent process
        value = fork();
```

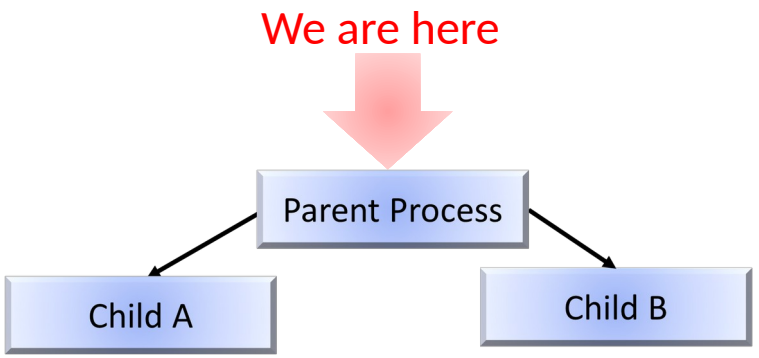
```c
    }
    else if(value == 0){
```



How to keep track of children and parents if you want to call fork multiple times?

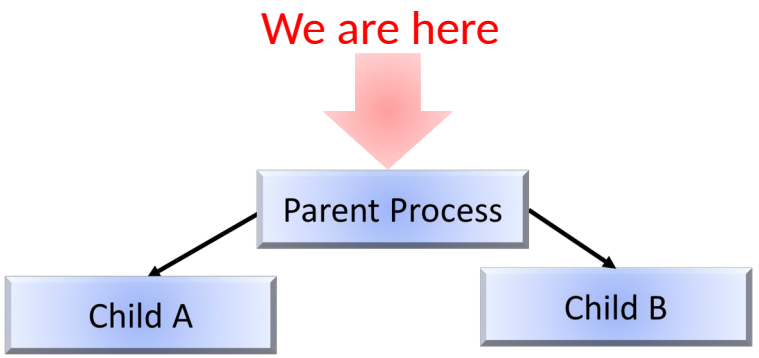
```c
#include <stdio.h>
#include <unistd.h>

int main(){
    pid_t value; //process identification used to represent process id
    value = fork();
    if(value != 0 ){ //in the parent process
        value = fork();
```

```c
    }
    else if(value == 0){
```



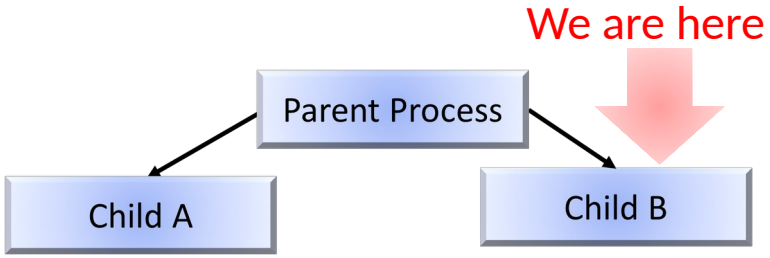
How to keep track of children and parents if you want to call fork multiple times?

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

int main(){
    pid_t value; //process identification used to represent process id
    value = fork();
    if(value != 0 ){ //in the parent process
        value = fork();
        if(value != 0 ){ //we are still the parent
            printf("Parent code finished running here!\n");
        }
        else{
            printf("In the child process B.\n");
        }
    }
    else if(value == 0){
```



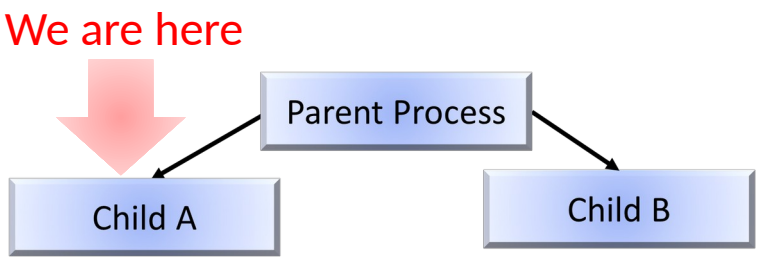
How to keep track of children and parents if you want to call fork multiple times?

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

int main(){
    pid_t value; //process identification used to represent process id
    value = fork();
    if(value != 0 ){ //in the parent process
        value = fork();
        if(value != 0 ){ //we are still the parent
            printf("Parent code finished running here!\n");
        }
        else{
            printf("In the child process B.\n");
        }
    }
    else if(value == 0){
```



How to keep track of children and parents if you want to call fork multiple times?

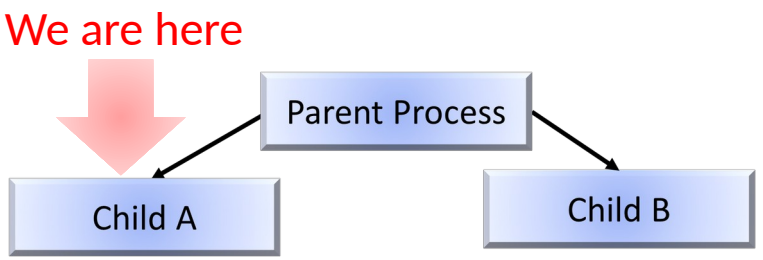
```
1   #include <stdio.h>
2   #include <unistd.h>
3
4   int main(){
5       pid_t value; //process identification used to represent process id
6       value = fork();
7       if(value != 0 ){ //in the parent process
16      else if(value == 0){
```



How to keep track of children and parents if you want to call fork multiple times?

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

int main(){
    pid_t value; //process identification used to represent process id
    value = fork();
    if(value != 0 ){ //in the parent process
```

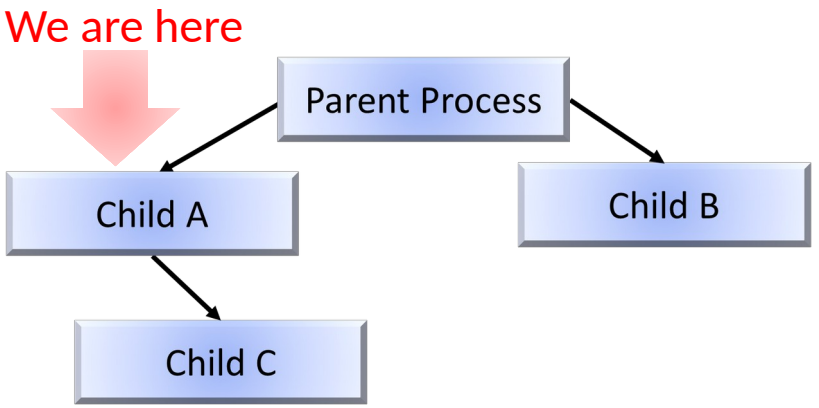
```c
    else if(value == 0){
        printf("In the child process A.\n");
        value = fork(); //note how the value will get changed in childA when calling fork
```



How to keep track of children and parents if you want to call fork multiple times?

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

int main(){
    pid_t value; //process identification used to represent process id
    value = fork();
    if(value != 0 ){ //in the parent process
```

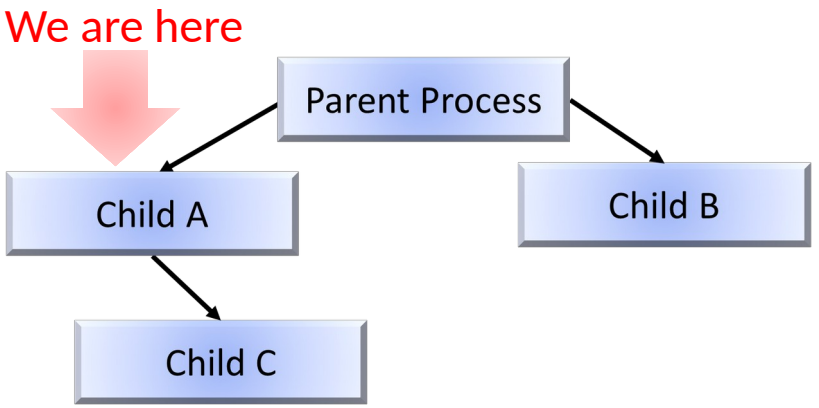
```c
    else if(value == 0){
        printf("In the child process A.\n");
        value = fork(); //note how the value will get changed in childA when calling fork
```



How to keep track of children and parents if you want to call fork multiple times?

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

int main(){
    pid_t value; //process identification used to represent process id
    value = fork();
    if(value != 0 ){ //in the parent process
```

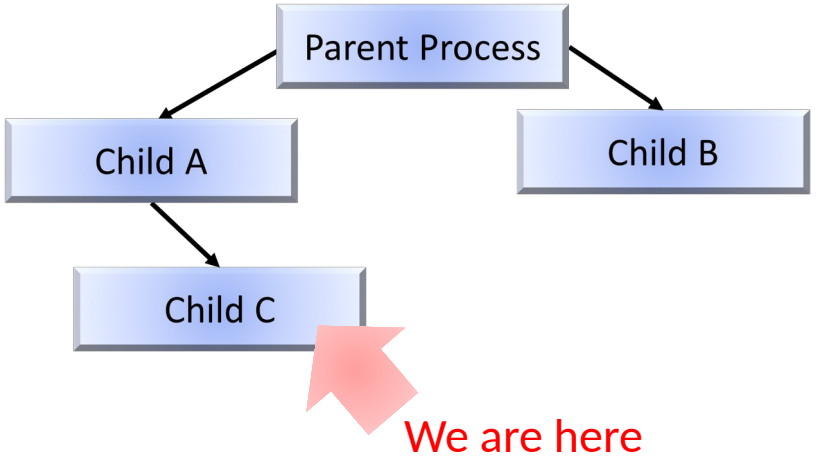
```c
    else if(value == 0){
        printf("In the child process A.\n");
        value = fork(); //note how the value will get changed in childA when calling fork
        if(value != 0 ){
            printf("Still inside child process A, but this process is now a parent.\n");
        }
        else if(value == 0){
```



How to keep track of children and parents if you want to call fork multiple times?

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

int main(){
    pid_t value; //process identification used to represent process id
    value = fork();
    if(value != 0 ){ //in the parent process
```

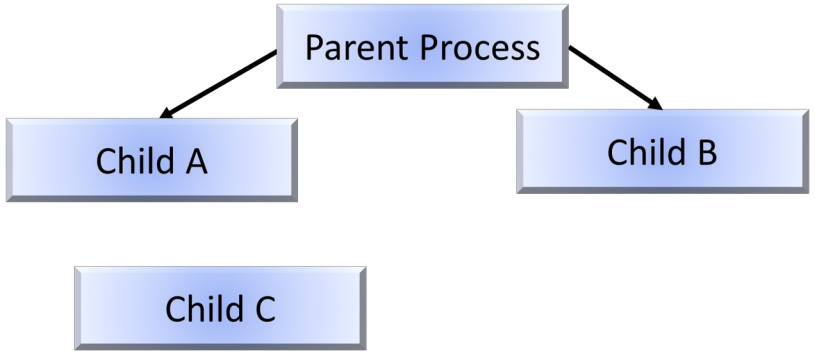
```c
    else if(value == 0){
        printf("In the child process A.\n");
        value = fork(); //note how the value will get changed in childA when calling fork
        if(value != 0 ){
            printf("Still inside child process A, but this process is now a parent.\n");
        }
        else if(value == 0){
            printf("Inside child process C.\n");
        }
    }
    return 0;
}
```



Complete Solution

One more question: could the solution be further improved upon?

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

int main(){
    pid_t value; //process identification used to represent process id
    value = fork();
    if(value != 0 ){ //in the parent process
        value = fork();
        if(value != 0 ){ //we are still the parent
            printf("Parent code finished running here!\n");
        }
        else{
            printf("In the child process B.\n");
        }
    }
    else if(value == 0){
        printf("In the child process A.\n");
        value = fork(); //note how the value will get changed in childA when calling fork
        if(value != 0 ){
            printf("Still inside child process A, but this process is now a parent.\n");
        }
        else if(value == 0){
            printf("Inside child process C.\n");
        }
    }
    return 0;
}
```



Question: What if we want the child do run some code that is not contained within the original file?

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

int main()
{
    pid_t value; // process identification used to represent process id
    value = fork(); //call the fork function to start a separate process
    if(value != 0)//this means we are in the parent process
    {
        printf("Let's do the first load of laundry.");
    }
    else if(value == 0) //this means we are the child process
    {
        [redacted]
    }
}
```

Run some piece of code that is in a different file.

We'll Explain This in Three Pieces:

First: Let's look at a separate piece of simple code.

Second: How can we run the code without forking?

Third: Run the code WITH forking

First: Let's look at a separate piece of simple code.

adder.c

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

int main(int argc,char* argv[]) {
    int i, sum=0;
    for(i=1;i<argc;i++)
        sum += atoi(argv[i]);
    printf("sum is: %d\n",sum);
    return 0;
}
```

First: Let's look at a separate piece of simple code.

adder.c

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

int main(int argc,char* argv[]) {
    int i, sum=0;
    for(i=1;i<argc;i++)
        sum += atoi(argv[i]);
    printf("sum is: %d\n",sum);
    return 0;
}
```

Number of arguments being passed.

- List of numbers to add together, can be any length.
- Numbers are input as strings from the command line.

First: Let's look at a separate piece of simple code.

adder.c

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

int main(int argc,char* argv[]) {
    int i, sum=0;
    for(i=1;i<argc;i++)
        sum += atoi(argv[i]);
    printf("sum is: %d\n",sum);
    return 0;
}
```

- Convert the string numbers to integer data types.
- Add the numbers to together.
- Print the result.

Second: How can we run the code without forking?

```
for(i=1;i<argc;i++)
    sum += atoi(argv[i]);
printf("sum is: %d\n",sum);
```

```
kaleel@CentralCompute:~$ gcc adder.c -o adder
kaleel@CentralCompute:~$ ./adder 10 20
sum is: 30
```

Third: Run the code WITH forking

```
1   #include <stdio.h>
2   #include <unistd.h>
3
4   int main() {
5       char *cmd1 = "./adder";
6       pid_t child = fork();
7       if (child == 0) {
8
9
10
11
12
13      } else {
14
15
16
17
18
19      }
20  }
```

Third: Run the code WITH forking

```
1   #include <stdio.h>
2   #include <unistd.h>
3
4   int main() {
5       char *cmd1 = "./adder";
6       pid_t child = fork();
7       if (child == 0) {
8
9
10
11
12
13      } else {
14
15
16
17
18
19      }
20  }
```

Immediately call the fork function to create two copies of the code.

Third: Run the code WITH forking

```
1   #include <stdio.h>
2   #include <unistd.h>
3
4   int main() {
5       char *cmd1 = "./adder";
6       pid_t child = fork();
7       if (child == 0) {
8           printf("In child!\n");
9           execl(cmd1,cmd1,"1","2","3",NULL);
10
11
12
13      } else {
14
15
16
17
18
19      }
20  }
```

This is the command to execute commands OUTSIDE of the main code.

Third: Run the code WITH forking

```
1   #include <stdio.h>
2   #include <unistd.h>
3
4   int main() {
5       char *cmd1 = "./adder";
6       pid_t child = fork();
7       if (child == 0) {
8
9
10          printf("Oops.... something went really wrong!\n");
11          perror(cmd1);
12          return -1;
13      } else {
14
15
16          printf("Oops.... something went really wrong!\n");
17          perror(cmd1);
18          return -1;
19      }
20  }
```

Do some basic error handling.

What does this look like in practice?

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

int main() {
    char *cmd1 = "./adder";
    pid_t child = fork();
    if (child == 0) {
        printf("In child!\n");
        execl(cmd1,cmd1,"1","2","3",NULL);
        printf("Oops.... something went really wrong!\n");
        perror(cmd1);
        return -1;
    } else {
        printf("In parent!\n");
        execl(cmd1,cmd1,"4","6",NULL);
        printf("Oops.... something went really wrong!\n");
        perror(cmd1);
        return -1;
    }
}
```

What does this look like in practice?

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

int main() {
    char *cmd1 = "./adder";
    pid_t child = fork();
    if (child == 0) {
        printf("In child!\n");
        execl(cmd1,cmd1,"1","2","3",NULL);
        printf("Oops.... something went really wrong!\n");
        perror(cmd1);
        return -1;
    } else {
        printf("In parent!\n");
        execl(cmd1,cmd1,"4","6",NULL);
        printf("Oops.... something went really wrong!\n");
        perror(cmd1);
        return -1;
    }
}
```

What does this look like in practice?

Parent Process (child variable has non-zero value)

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

int main() {
    char *cmd1 = "./adder";
    pid_t child = fork();
    if (child == 0) {
        printf("In child!\n");
        execl(cmd1,cmd1,"1","2","3",NULL);
        printf("Oops.... something went really wrong!\n");
        perror(cmd1);
        return -1;
    } else {
        printf("In parent!\n");
        execl(cmd1,cmd1,"4","6",NULL);
        printf("Oops.... something went really wrong!\n");
        perror(cmd1);
        return -1;
    }
}
```

Child Process (child variable has 0 value)

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

int main() {
    char *cmd1 = "./adder";
    pid_t child = fork();
    if (child == 0) {
        printf("In child!\n");
        execl(cmd1,cmd1,"1","2","3",NULL);
        printf("Oops.... something went really wrong!\n");
        perror(cmd1);
        return -1;
    } else {
        printf("In parent!\n");
        execl(cmd1,cmd1,"4","6",NULL);
        printf("Oops.... something went really wrong!\n");
        perror(cmd1);
        return -1;
    }
}
```

What does this look like in practice?

Parent Process (child variable has non-zero value)

```
1   #include <stdio.h>
2   #include <unistd.h>
3
4   int main() {
5       char *cmd1 = "./adder";
6       pid_t child = fork();
7       if (child == 0) {
8           printf("In child!\n");
9           execl(cmd1,cmd1,"1","2","3",NULL);
10          printf("Oops.... something went really wrong!\n");
11          perror(cmd1);
12          return -1;
13      } else {
14          printf("In parent!\n");
15          execl(cmd1,cmd1,"4","6",NULL);
16          printf("Oops.... something went really wrong!\n");
17          perror(cmd1);
18          return -1;
19      }
20  }
```

In the parent we will go to line 13 and start executing code.

Child Process (child variable has 0 value)

```
1   #include <stdio.h>
2   #include <unistd.h>
3
4   int main() {
5       char *cmd1 = "./adder";
6       pid_t child = fork();
7       if (child == 0) {
8           printf("In child!\n");
9           execl(cmd1,cmd1,"1","2","3",NULL);
10          printf("Oops.... something went really wrong!\n");
11          perror(cmd1);
12          return -1;
13      } else {
14          printf("In parent!\n");
15          execl(cmd1,cmd1,"4","6",NULL);
16          printf("Oops.... something went really wrong!\n");
17          perror(cmd1);
18          return -1;
19      }
20  }
```

In the child we will go to line 8 and start executing code.

What does this look like in practice?

Parent Process (child variable has non-zero value)

```
1   #include <stdio.h>
2   #include <unistd.h>
3
4   int main() {
5       char *cmd1 = "./adder";
6       pid_t child = fork();
7       if (child == 0) {
8           printf("In child!\n");
9           execl(cmd1,cmd1,"1","2","3",NULL);
10          printf("Oops.... something went really wrong!\n");
11          perror(cmd1);
12          return -1;
13      } else {
14          printf("In parent!\n");
15          execl(cmd1,cmd1,"4","6",NULL);
16          printf("Oops.... something went really wrong!\n");
17          perror(cmd1);
18          return -1;
19      }
20  }
```

At line 15 we'll go into a new code and execute. Its important to note, we DO NOT return.

Child Process (child variable has 0 value)

```
1   #include <stdio.h>
2   #include <unistd.h>
3
4   int main() {
5       char *cmd1 = "./adder";
6       pid_t child = fork();
7       if (child == 0) {
8           printf("In child!\n");
9           execl(cmd1,cmd1,"1","2","3",NULL);
10          printf("Oops.... something went really wrong!\n");
11          perror(cmd1);
12          return -1;
13      } else {
14          printf("In parent!\n");
15          execl(cmd1,cmd1,"4","6",NULL);
16          printf("Oops.... something went really wrong!\n");
17          perror(cmd1);
18          return -1;
19      }
20  }
```

At line 9 we'll go into a new code and execute. Its important to note, we DO NOT return.

What does this look like in practice?

Parent Process (child variable has non-zero value)

```
1   #include <stdio.h>
2   #include <unistd.h>
3
4   int main() {
5       char *cmd1 = "./adder";
6       pid_t child = fork();
7       if (child == 0) {
8           printf("In child!\n");
9           execl(cmd1,cmd1,"1","2","3",NULL);
10          printf("Oops.... something went really wrong!\n");
11          perror(cmd1);
12          return -1;
13      } else {
14          printf("In parent!\n");
15          execl(cmd1,cmd1,"4","6",NULL);
16          printf("Oops.... something went really wrong!\n");
17          perror(cmd1);
18          return -1;
19      }
20  }
```

At line 15 we'll go into a new code and execute. Its important to note, we DO NOT return.

We'll get 4+6, which should print 10.

Child Process (child variable has 0 value)

```
1   #include <stdio.h>
2   #include <unistd.h>
3
4   int main() {
5       char *cmd1 = "./adder";
6       pid_t child = fork();
7       if (child == 0) {
8           printf("In child!\n");
9           execl(cmd1,cmd1,"1","2","3",NULL);
10          printf("Oops.... something went really wrong!\n");
11          perror(cmd1);
12          return -1;
13      } else {
14          printf("In parent!\n");
15          execl(cmd1,cmd1,"4","6",NULL);
16          printf("Oops.... something went really wrong!\n");
17          perror(cmd1);
18          return -1;
19      }
20  }
```

At line 9 we'll go into a new code and execute. Its important to note, we DO NOT return.

We'll get 1+2+3, which should print 6.

What does this look like in practice?

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

int main() {
    char *cmd1 = "./adder";
    pid_t child = fork();
    if (child == 0) {
        printf("In child!\n");
        execl(cmd1,cmd1,"1","2","3",NULL);
        printf("Oops.... something went really wrong!\n");
        perror(cmd1);
        return -1;
    } else {
        printf("In parent!\n");
        execl(cmd1,cmd1,"4","6",NULL);
        printf("Oops.... something went really wrong!\n");
        perror(cmd1);
        return -1;
    }
}
```

```
kaleel@CentralCompute:~$ ./test
In parent!
In child!
sum is: 10
sum is: 6
```

For this example, where were the files such that they could be

Where were the files such that they could be accessed?

adder.c

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

int main(int argc,char* argv[]) {
    int i, sum=0;
    for(i=1;i<argc;i++)
        sum += atoi(argv[i]);
    printf("sum is: %d\n",sum);
    return 0;
}
```

test.c

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

int main() {
    char *cmd1 = "./adder";
    pid_t child = fork();
    if (child == 0) {
        printf("In child!\n");
        execl(cmd1,cmd1,"1","2","3",NULL);
        printf("Oops.... something went really wrong!\n");
        perror(cmd1);
        return -1;
    } else {
        printf("In parent!\n");
        execl(cmd1,cmd1,"4","6",NULL);
        printf("Oops.... something went really wrong!\n");
        perror(cmd1);
        return -1;
    }
}
```

Main Ubuntu Directory (using Windows Subsystem Linux)

.cache	1/8/2023 10:40 PM	File folder	
.vscode	1/8/2023 10:41 PM	File folder	
.vscode-server	1/8/2023 10:39 PM	File folder	
.bash_history	1/9/2023 7:35 PM	BASH_HISTORY File	1 KB
.bash_logout	1/8/2023 10:38 PM	Bash Logout Sourc...	1 KB
.bashrc	1/8/2023 10:38 PM	Bash RC Source File	4 KB
.motd_shown	1/11/2023 7:17 PM	MOTD_SHOWN File	0 KB
.profile	1/8/2023 10:38 PM	Profile Source File	1 KB
.sudo_as_admin_successful	1/8/2023 10:45 PM	SUDO_AS_ADMIN...	0 KB
.wget-hsts	1/8/2023 10:39 PM	WGET-HSTS File	1 KB
adder	1/11/2023 9:03 PM	File	16 KB
adder	1/11/2023 8:58 PM	C Source File	1 KB
test	1/11/2023 10:33 PM	File	16 KB
test	1/11/2023 10:33 PM	C Source File	2 KB

Process upgrades

- Usually….
  - A fresh clone wants to run *different code*
- This is done by
  - Loading another executable into the process address space
    - [picked up from the file system of course]
- Note
  - **Opened files are NOT AFFECTED by the upgrade operation**

The `exec` family

Basically any call to Execl:

- The act of 'upgrading' is done by the child with a system call
  - Many variants. "man -S3 execl" for all details

```
#include <unistd.h>
int execl(const char *path, const char *arg0, ... /*, (char *) NULL */ );
```

  - The path to the executable to load inside our own address space
  - A list of arguments to be passed to the new executable
  - A final NULL pointer to give the "end of argument list"
  - If successful, `execl()` does not return! Started a new process

How is the executable found?

- Specify a path, like `/bin/ls`
- Specify a file, and the system searches in directories listed in PATH
  - `echo $PATH` in bash to see directories separated by ':'

```
int execl(const char *path, const char *arg0, ...
      /*, (char *) NULL */ );

// execlp() searches paths for file
int execlp(const char *file, const char *arg0, ...
      /*, (char *) NULL */ );
```

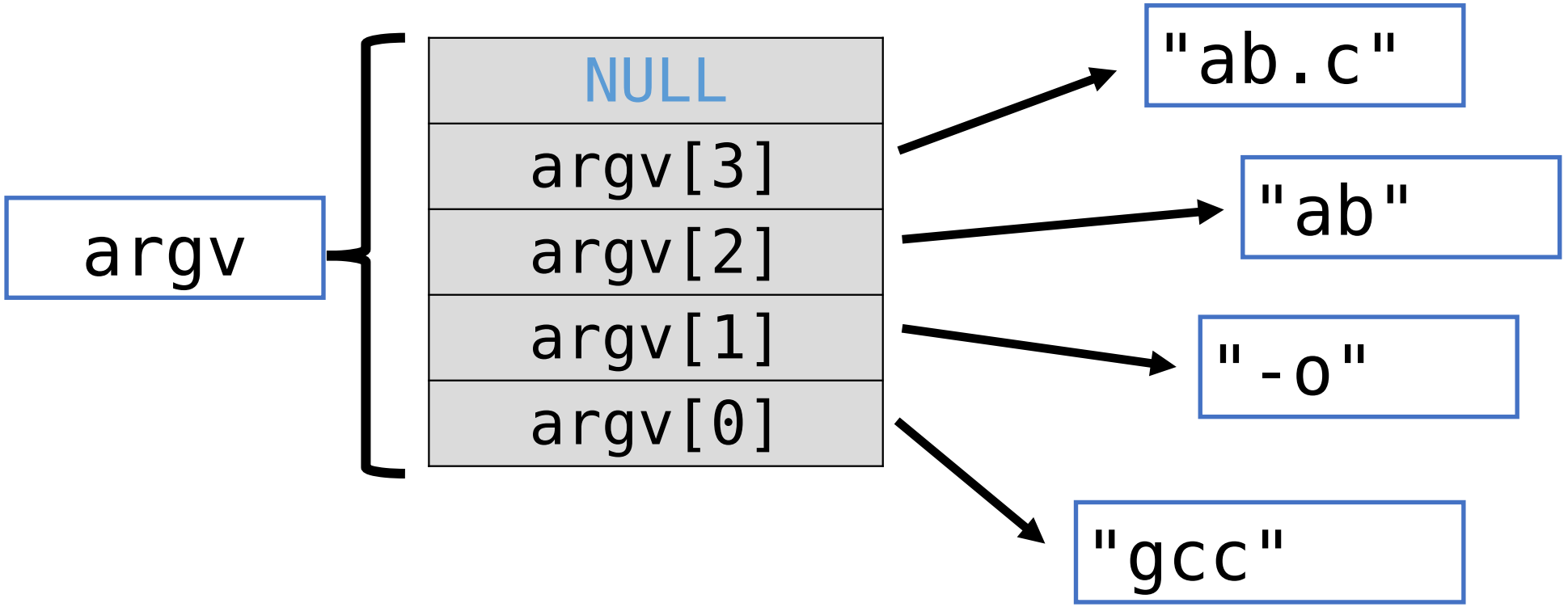
The `exec` family 2

```
// If the number of arguments is unknown at compile time
#include <unistd.h>
int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const
argv[]);
```

- The arguments in execl in are placed in an array
  - argv is the argv you see in the main function!
- execv needs a path while execvp can search file in PATH
- Start a new process if successful. Similar to execl

A small note on: argv to execv and execvp

```
$gcc –o ab ab.c
```



- Note that the last value in argv is null? Why?
- So that the code knows where to stop reading.

Common Errors in Writing Forking Code (Example 1)

```
pid_t pid = fork();
if (pid < 0) {
     perror("fork()"); exit(1);    // exit if fork()
fails
} else if (pid == 0) {
     child_tasks();
} else {
     parent_tasks();
}
more_parent_tasks();
```

What is wrong with this code?

Example 1: Consider the child process

```
pid_t pid = fork();
if (pid < 0) {
     perror("fork()"); exit(1);    // exit if fork()
fails
} else if (pid == 0) {
     child_tasks();
} else {
     parent_tasks();
}
more_parent_tasks();
```

Example 1: Consider the child process

```
pid_t pid = fork();
if (pid < 0) {
     perror("fork()"); exit(1);    // exit if fork()
fails
} else if (pid == 0) {
     child_tasks();
} else {
     parent_tasks();
}
more_parent_tasks();
```

Example 1: Consider the child process

```
pid_t pid = fork();
if (pid < 0) {
     perror("fork()"); exit(1);    // exit if fork()
fails
} else if (pid == 0) {
     child_tasks();
} else {
     parent_tasks();
}
more_parent_tasks();
```

Example 1: Consider the child process

```
pid_t pid = fork();
if (pid < 0) {
     perror("fork()"); exit(1);    // exit if fork()
fails
} else if (pid == 0) {
     child_tasks();
} else {
     parent_tasks();
}
more_parent_tasks();
```

Visualization of the Child Process when you ask it to mow the lawn:

Example 1: Corrected Code

```
pid_t pid = fork();
if (pid < 0) {
     perror("fork()"); exit(1);    // exit if fork() fails
} else if (pid == 0) {
     child_tasks();
     exit(0);  // terminate the child process
} else {
     parent_tasks();
}
more_parent_tasks();
```

Example 2: What could go wrong?

```
pid_t pid = fork();
if (pid < 0) {
     perror("fork()"); exit(1);    // exit if fork() fails
}
else if (pid == 0) {
     // in child process
     execlp("genie", "genie", "clean the house", NULL);
}
// in parent process
online_shopping();
```

Would you ever know if execlp failed?

Example 2: Corrected Code

```
pid_t pid = fork();
if (pid < 0) {
     perror("fork()"); exit(1);    // exit if fork() fails
}
else if (pid == 0) {
     // in child process
     execlp("genie", "genie", "clean the house", NULL);
     printf("Something went really wrong in child process!\n");
}
// in parent process
online_shopping();
```

File APIs
(API = Application Programming Interface)

- Remember the (C standard library) IO APIs
  - The "f" family (fopen,fclose, fread, fgetc, fscanf, fprintf,…)
  - All these use a FILE* abstraction to represent a file
    - Additional features: user-space buffering, line-ending translation, formatted I/O, etc.
- UNIX has lower-level APIs for file handling
  - Directly mapped to **system calls**
    - open, close, read, …
  - Use *file descriptors* **[which are just integers]**
  - Deal with bytes only

Some low level file APIs

- Read the man pages (man –s2 ...) for more functions

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

int open(const char *path, int oflag);
int open(const char *path, int oflag, int mode);
int close(int fd);

ssize_t read(int fd, void *buf, size_t nbyte);
ssize_t write(int fd, const void *buf, size_t nbyte);

off_t lseek(int fd, off_t offset, int whence);
```

How to open a file

```
#include <fcntl.h>
#include <unistd.h>

int open(const char *path, int oflag);
```

- Parameters
  - path: the path to the file to be opened/created
  - oflag: read, write, or read and write, and more (on the next slide)

- The function returns a file descriptor, a small, nonnegative integer
  - Return -1 on error

Flags in open()

- Must include one of the following:
  O_RDONLY (read only), O_WRONLY (write only), or O_RDWR (read and write)
- And or-ed (|) with many optional flags, for example,
  - O_TRUNC: Truncate the file (remove existing contents) if opening a file for write
  - O_CREAT: Create a file if it does not exist.

Example:

```
// remember open() returns -1 on error
fd1 = open("a.txt", O_RDONLY); // open for read
fd1 = open("a.txt", O_RDWR); // open for read and write
fd1 = open("a.txt", O_RDWR|O_TRUNC); // read, write, truncate the
file
```

Creating a file with open()

```
// a mode must be provided if O_CREAT or O_TMPFILE is set

int open(const char *path, int oflag, int mode);
```

mode: specify permissions when a new, or temporary, file is created.

```
open("b.txt", O_WRONLY|O_TRUNC|O_CREAT, 0600);
```

// open b.txt for write. If the file exists, clear (truncate) the contents.

// if the file does not exist, create one, and set the permission so that the owner of the file can read and write, but other people cannot.

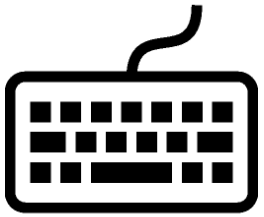
File descriptors

- A file descriptor is a nonnegative integer associated with a file.

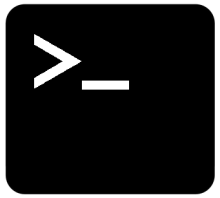
```
#include <stdio.h>
int fileno(FILE *stream);
// returns a file descriptor for a stream
```

FD	FILE *
0	stdin
1	stdout
2	stderr

The input, for example the process takes inputs from the keyboard.



Where the process gives outputs e.g. the terminal.



Where the process reports errors, e.g. could be a log file recording the errors.

File descriptors after fork and exec

- Opened files are NOT AFFECTED by the upgrade operation

```
pid_t pid = fork();
assert(pid >= 0);
if (pid == 0) {
    // Child process can access FDs 0, 1, and 2
    // if execl() is successful, gcc can access FDs 0, 1, and 2
    execlp("gcc", "gcc", "a.c", NULL);
    // If control gets here, execlp() failed.
    // Remember to terminate the child process!
    return 1;
}
```

Visualization of File Descriptor with Fork

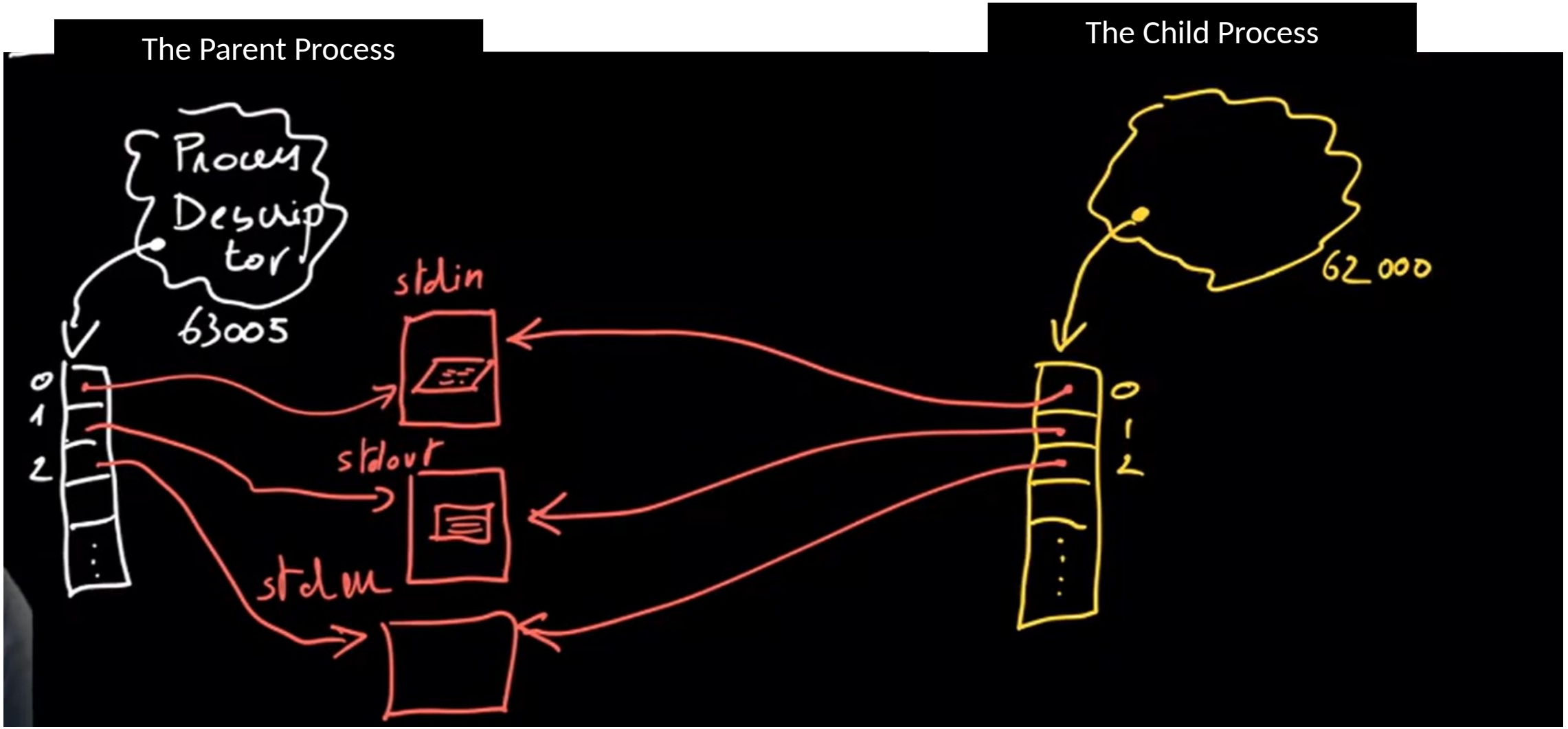
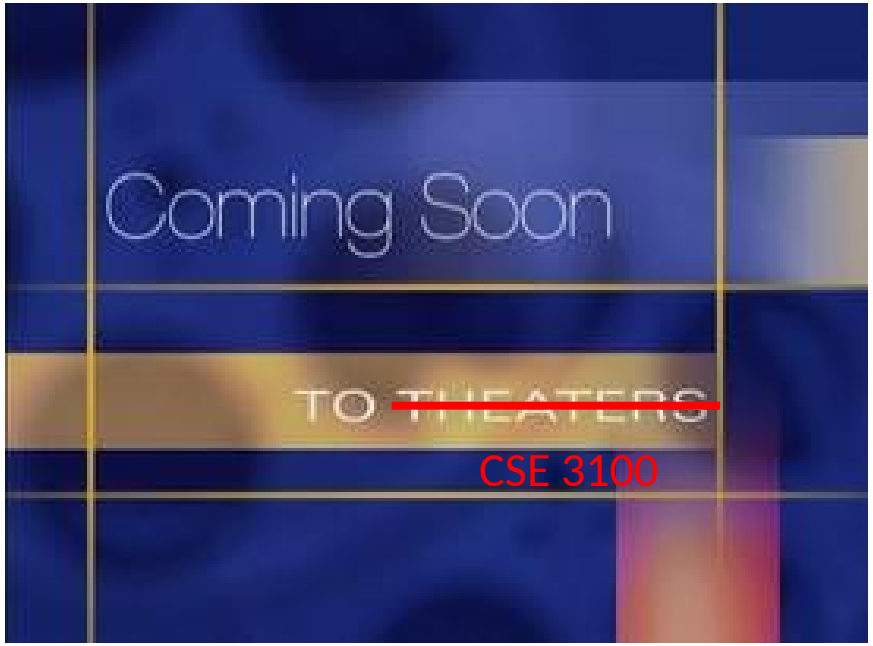


Figure Source: Professor Laurent Michel Youtube Lecture Videos

Preview of the future



- You already know that child processes copy the variables from the parent code.
- It follows that the file descriptors are also copied over to the child.
- So why all the focus on file descriptors? This will come in handy in the future when we want processes to communicate with each other.

Figure Sources

1. https://i.kym-cdn.com/entries/icons/facebook/000/019/404/upgradddd.jpg
2. https://www.mememaker.net/static/images/memes/4761591.jpg
3. https://thumbs.gfycat.com/HollowPositiveAnophelesmosquito-max-1mb.gif
4. https://iconarchive.com/download/i87838/icons8/ios7/Computer-Hardware-Keyboard.ico
5. https://icons.iconarchive.com/icons/paomedia/small-n-flat/1024/terminal-icon.png
6. https://cdn-icons-png.flaticon.com/512/1388/1388902.png
7. https://static.seekingalpha.com/uploads/2013/3/21/7360901-13638972437431467-Robert-Wagner.jpg