Real-Time Operating Systems

Contents:

System call

Makefile

Processes

Preemption

- Preemption is the act of temporarily interrupting a currently scheduled task in favour of a higher priority task.

- Why we assume tasks are preemptible in RTOS?

- Linux System
  - Kernel programs can always preempt user-space programs
  - Until 2.6 kernels, the kernel itself was not preemtible
  - Kernel preemption has been introduced in 2.6 kernels, and one can enable or disable the pre-emption

Comment on Building and Loading Module

After building the kernel object, you will get **hello.ko**

Then you may want to insert the kernel module using

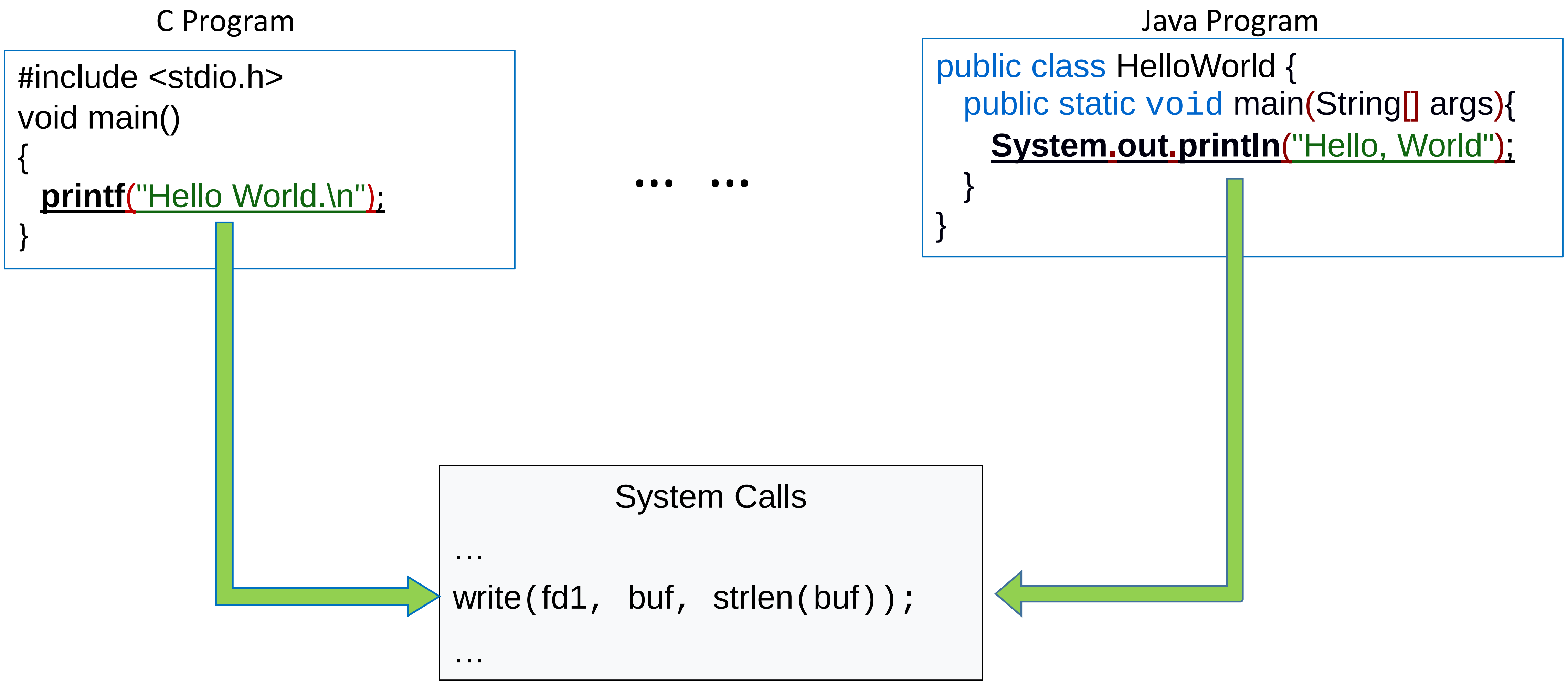
Insmod hello.ko

Can you succeed on Linux?

Why we learn kernel/module programming for real-time tasks?

System Calls

printf(), System.out.println(), and write()



C Program

```
#include <stdio.h>
void main()
{
  printf("Hello World.\n");
}
```

… …

Java Program

```
public class HelloWorld {
   public static void main(String[] args){
      System.out.println("Hello, World");
   }
}
```

System Calls

```
…
write(fd1, buf, strlen(buf));
…
```

Can You Make A RAW System Call?

C Program

```
#include <stdio.h>
void main()
{
  printf("Hello World.\n");
}
```

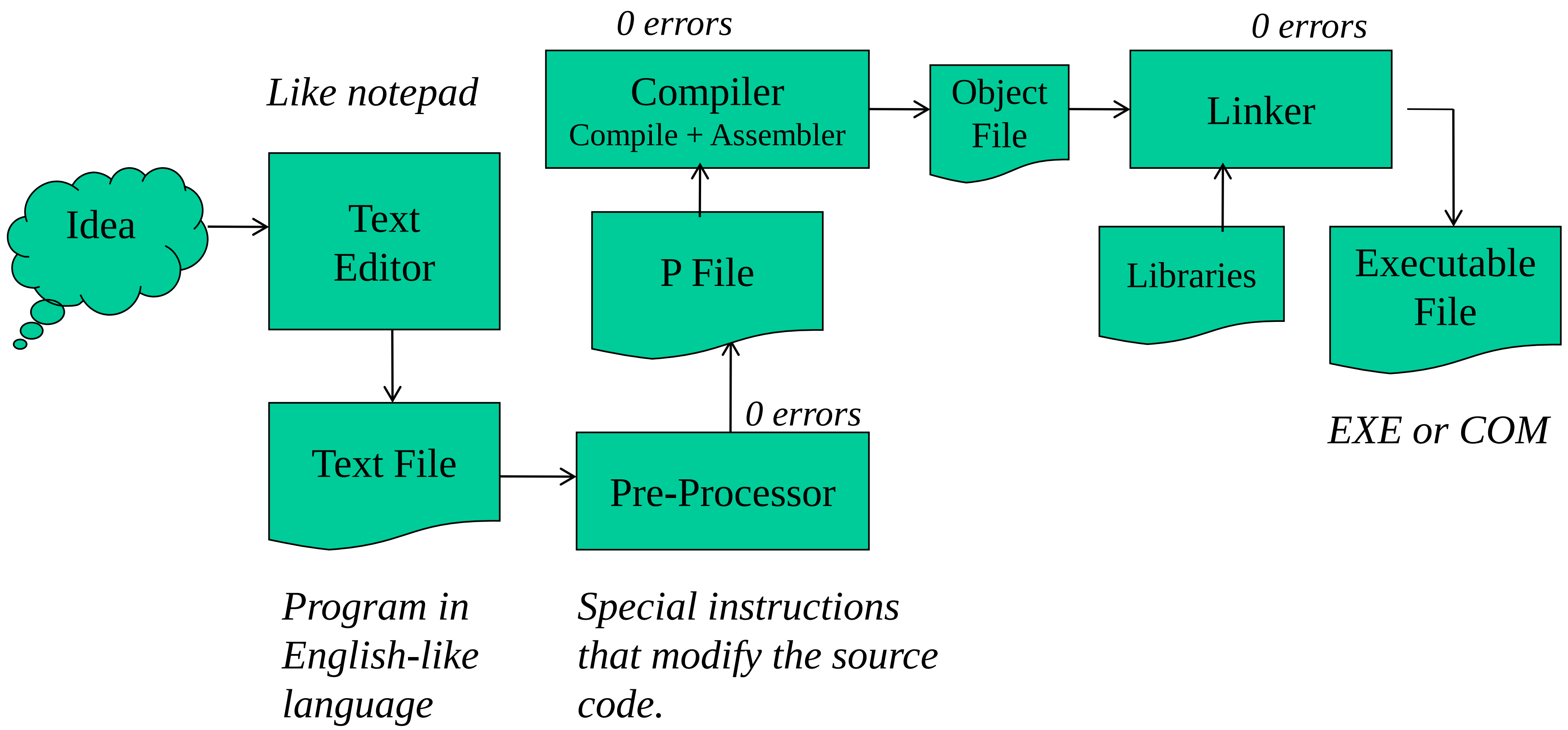
C Program

```
void main()
{
  write(1, "Hello World.\n“, 13);
}
```

- Both C program did the same thing.
- Why it is rare to make a raw system call from a user program?
  - *APIs provided in library is easy to use compared to actual system call*
  - *APIs provides some form of Security/Protection to the kernel because you are not interacting with the kernel*
  - *APIs provides better portability*
  - *APIs can be more efficient*

Makefile

The C Compilation Process



How to compile multiple source files?

file1.c

```
#include “file2.h”

int main()
{
  printName();
  return 0;
}
```

file2.c

```
#include <stdio.h>

void printName()
{
  printf(“My name is
  Wenbo.”);
}
```

file2.h

```
void printName(void);
```

Compiling Multiple Files

```
$ gcc -c file1.c file2.c

$ ls
file1.c file2.h file2.c file1.o file2.o

$ gcc file1.o file2.o
file1.c file2.h file2.c file1.o file2.o a.out
```

How to compile a large number of source files?

Makefile

- To compile a simple program,

```
gcc –o myprogram main.c
```

- To compile a large project with many source files, e.g.,

```
gcc -c file1.c
gcc -c file2.c
…
gcc -o file1.o file2.o
```

Question:

If you changed a small piece of code, do you need to compile the code all over again?

No. But, it is hard to track which parts need to be recompiled.

Makefile

```
final_target: sub_target final_target.c
        Instruction_to_create_final_target

sub_target: sub_target.c
        Instruction_to_create_sub_target
```

•Make gets its instruction from the “*makefile*” file.

–It's a collection of rules and instructions explaining how to compile your program.

•The first rule in your make file is your default rule.

–If you make a mistake building your rule, your application will not compile properly.

•You just need to type “make” at the command-line to run make. This will run the default rule.

```
make
```

–makefile is the default instruction file and is automatically used.

–To execute a specific action, specify that action as an argument.

```
make clean
```

Example

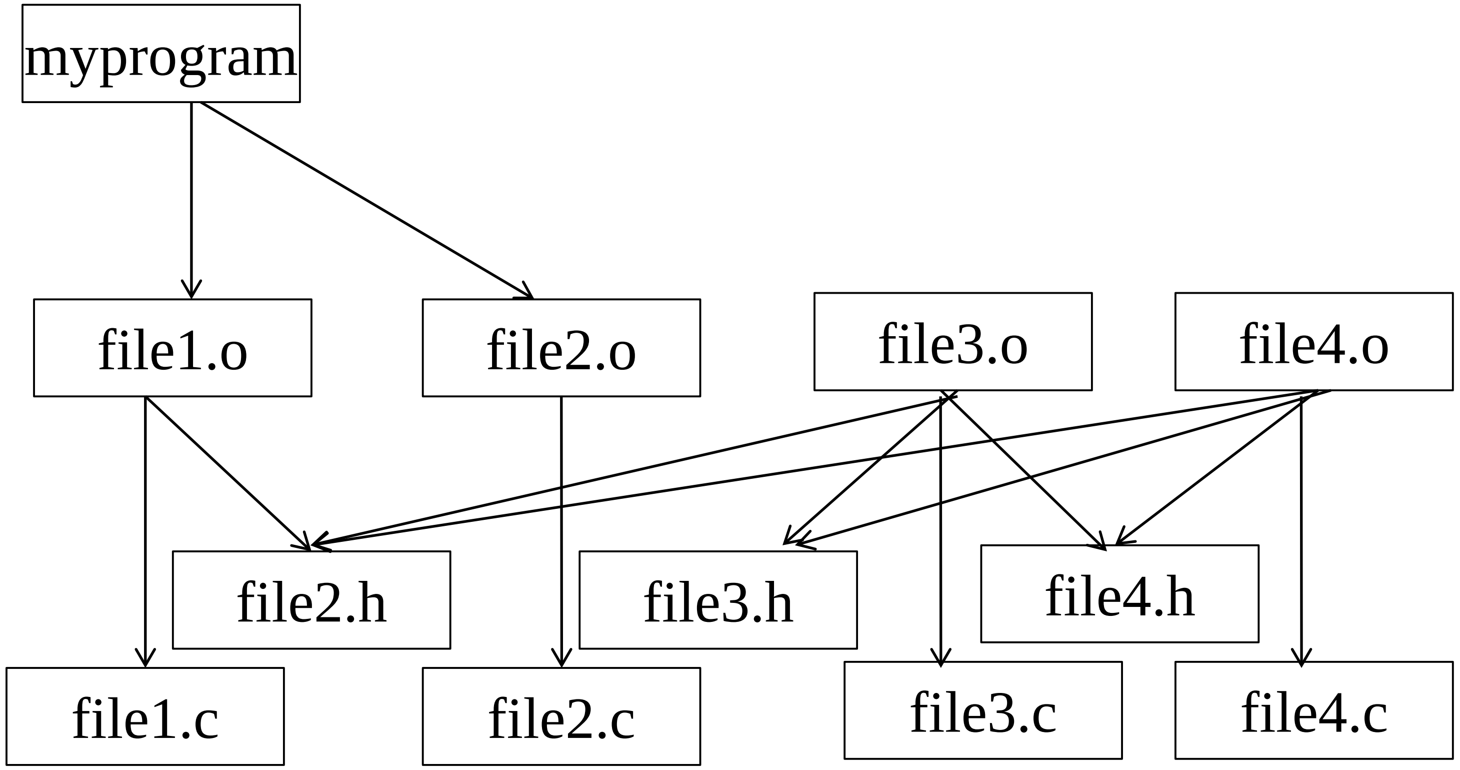
•Makefile consists of a series of rules

–target: requirements   target build instructions

•Makefile example:

```
myprogram: file1.o file2.o
     gcc –o myprogram file1.o file2.o
file1.o: file1.c
     gcc  -c file1.c
file2.o: file2.c
     gcc -c file2.c
clean:
     rm -f myprogram file1.o file2.o
```

Dependency Tree



If something changes what needs to be recompiled?

Processes

Process

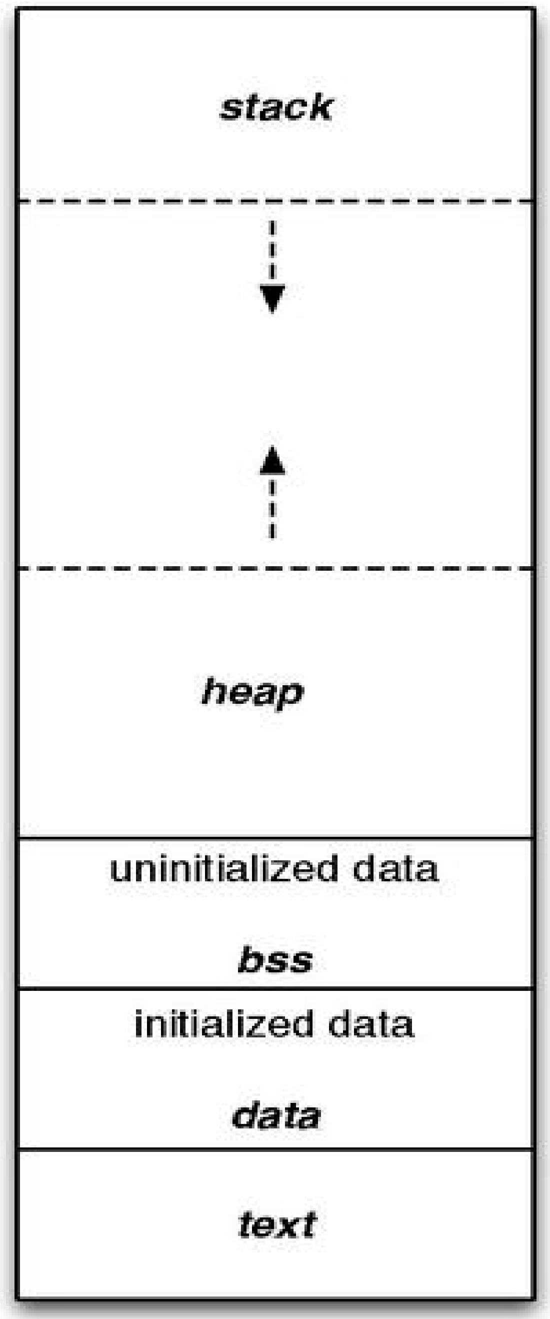
- A program in execution.
- An abstraction of a running program
- The logical unit of work scheduled by operating system.

Processes are independent, carry considerable state information, have separate address spaces and interact through system-provided inter-process communication mechanism.

Program in Virtual Memory

- Stack: Used to store function arguments and local variables, and the return address of functions that called the current function.

- Heap: Memory is dynamically located by system calls
  - new(), malloc(), calloc(), …
- BSS segment: uninitialized data
- Data segment: initialized data
  - global variables and static variables
- Text: Read-only region containing program instructions (or program code)



Stack v.s. Heap

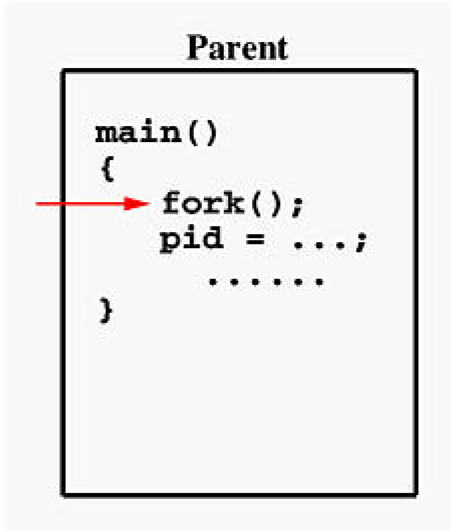
	Stack	**Heap**
Creation of an object	Member m;	Member* m = new Member();
Lifetime	Function runs to completion	delete, free is called
Grow in size?	Fixed size	More memory can be added by the operating system
Common error	Stack overflow	Heap fragmentation
Which one to use?	Know the size of memory to be used, or when data size is small	When you need a large scale of dynamic memory

Multiple Processes

- We could divide a program into multiple tasks by creating multiple processes using the fork() command.

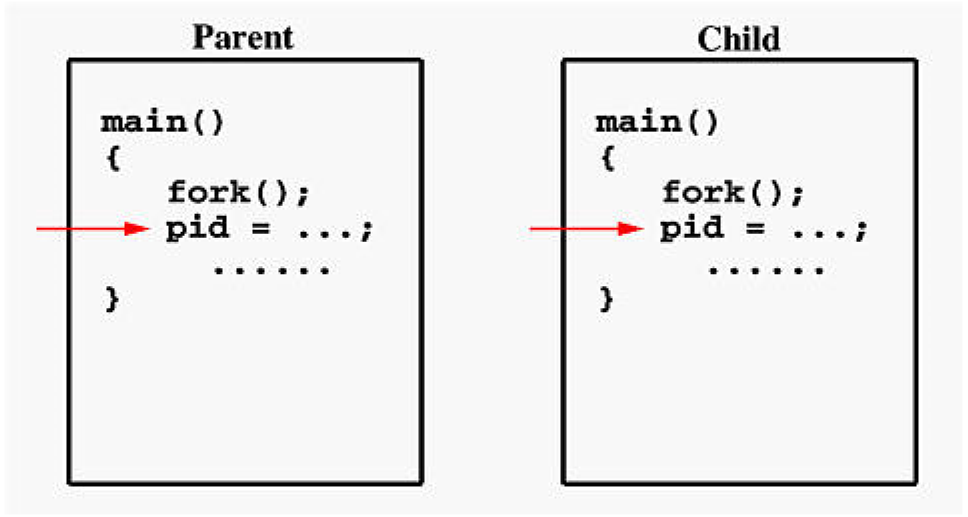
- Fork() creates a child process identical to its parent.

- fork() returns a **value of 0** to the child process and returns the **process ID** of the child process to the parent process.



```
main()
{
    fork();
    pid = ...;
      ......
}
```

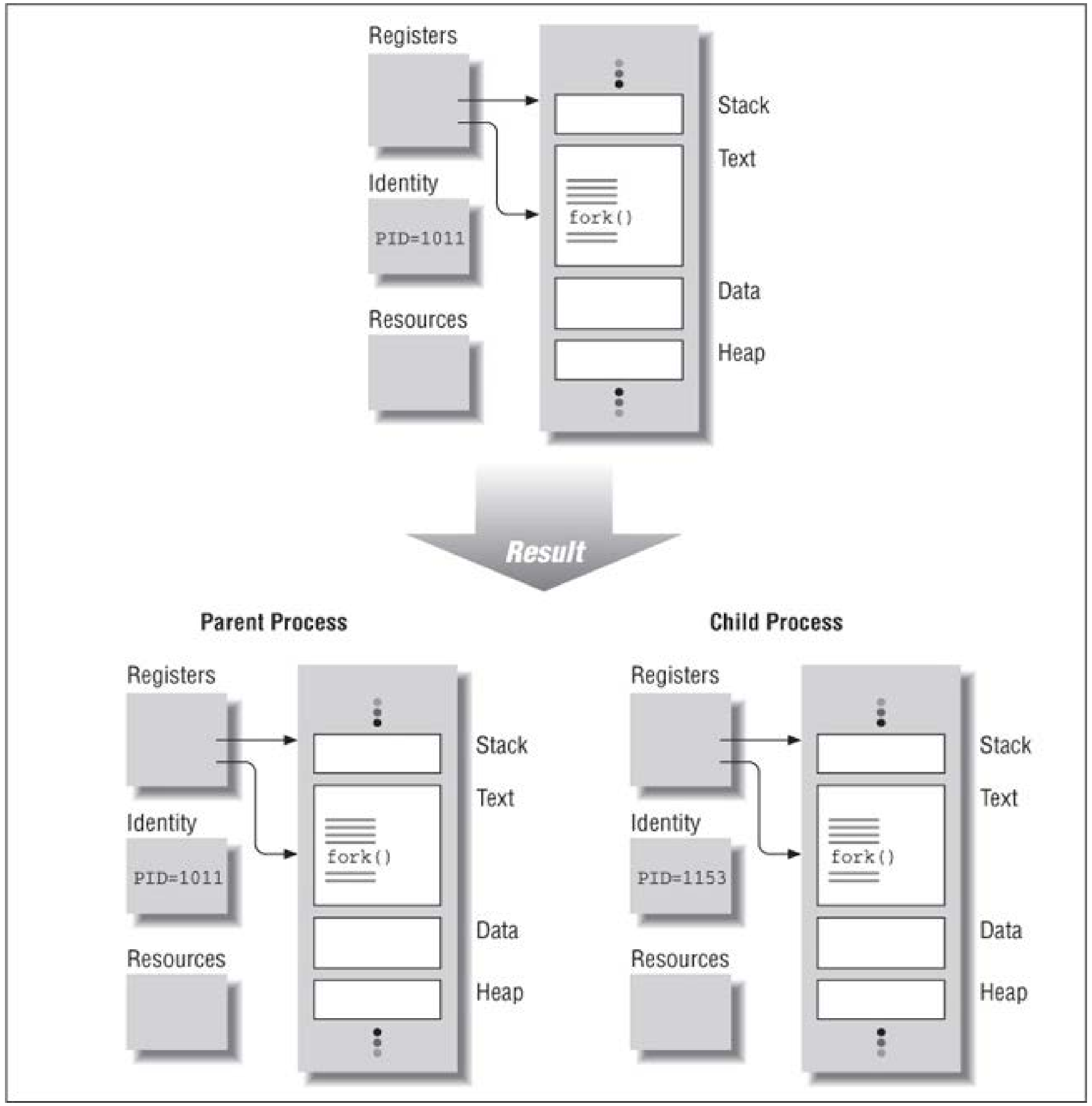
If the call to **fork()** is executed successfully, Unix will
(1) Make two identical copies of address spaces, one for the parent and the other for the child.
(2) Both processes will start their execution at the next statement following the **fork()** call.



```
main()
{
    fork();
    pid = ...;
      ......
}
```

```
main()
{
    fork();
    pid = ...;
      ......
}
```

A program before and after a fork



Fork() Call

- Lot of overhead to create process as everything duplicated. Also, data space isn't shared, so harder to communicate.
- Variables initiated before fork() will be duplicated in both parent and child process. After fork() branch is needed to separate the parent and the child.

```
void main(void)
 {
     pid_t pid;
     pid = fork();

     if (pid == 0)
          ChildProcess();
     else
          ParentProcess();
 }
```

Example

```
#include<stdio.h>

main(int argc, char ** argv)
{
    int child = fork();
    int c = 0;
    if(child)
        c += 5;
    else  {
        child = fork();
        c += 5;
        if(child)
            c += 5;
    }
    printf("%d ",c);
}
```

Example

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>

int main() {
   pid_t pid;
   char * str;
   str = malloc( 100 );
   strcpy( str, "Hello" );
   pid = fork();
if( pid == 0 ) {
      printf( "In Child, the string is: %s\n", str );
      strcpy( str, "Goodbye" );
      printf( "In Child, the string is: %s\n", str );
      free( str );
  }
else if( pid>0) {
      printf( "In Parent, the string is: %s\n", str );
      sleep(1);
      printf( "In Parent, the string is: %s\n", str );
      free( str );
   }
Else
      printf((Error with fork()\n);
      return 0;
}
```