Program and Processes

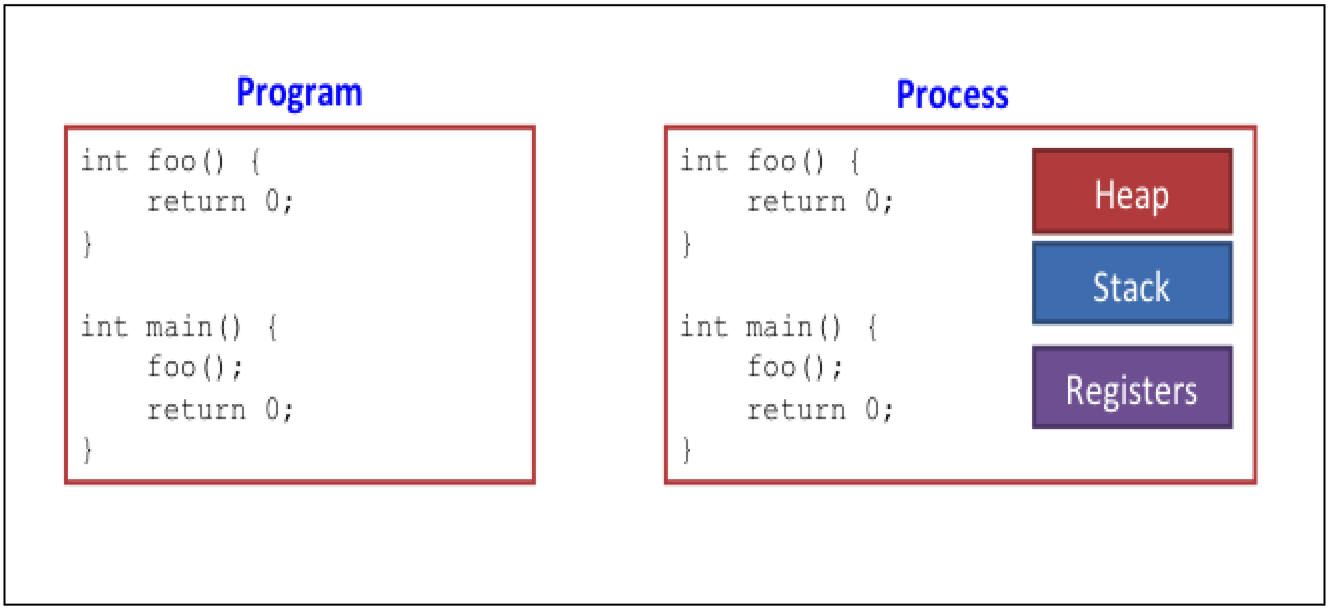
Program:

- A group of instructions to carry out a specified task
- Executable binary (code and static data)

Process:

- A program loaded into memory
- Program in execution.

  - Program (executable binary with data and text section)
  - Execution state (heap, stack, and processor registers)


Program
int foo() {
return 0;
}
int main() {
foo();
return 0;
}
Process
int foo() {
return 0;
}
int main() {
foo();
return 0;
}
Heap
Stack
Registers

Processes

- Any program ***running*** on the system is called a process
- UNIX **process** is an instance of a program in execution. It can be described by :

  - The executable code (stored in the **text segment** of the virtual memory image of the process)
  - The program data (stored in the data segment)
  - The state, including stack pointer and stack, program counter, etc. (usually collected in a process control block, or PCB)

- Linux is a **multi-process** system

- At any point there may be a **number of processes *active*** on the system

  - On Linux an active process can be in one of three states

    - **Running**: Executing on the CPU
    - **Ready**: Waiting to execute on CPU
    - **Blocked**: Waiting for I/O or synchronization with another thread

- The **OS is responsible** for determining **which process gets** control of the CPU
    - First come first serve , shortest job processing first , Round Robin (Time slicing) , priority based on process priority which is often difficult to do.

- **There are various ways to launch processes** that then communicate, and two ways dominate in the examples that follow:
    - A **terminal** is used to start one process, and perhaps a different terminal is used to start another.
    - The system function **fork** is called within one process (the parent) to spawn another process (the child).

Process Communication

- Inter Process Communication (IPC)
    - Pipe (named or unnamed)
    - Signal
    - Socket
        - Local communication
        - Network communication
        - Client and server
    - Shared memory (with semaphores)
    - Shared files
    - Message queues

- Thread
    - Intra-process

How are Processes Created?

- A process is **created** from another process
    - Parent-child relationship
    - When we type a **command at the shell prompt** , a process is created
        - **shell** is the **parent** process
        - The **command** is the **child** process

- We can also create processes from within other programs. What does the above imply?

    - Need **one process that invokes the initial processes**
        - **init**
        - It is the **only process** that **does not have a parent**

- Every process has certain properties and resources associated with them such as **Process ID (pid) and Parent Process ID (ppid)**

- **What happens** when the child dies before the parent ???
- **What happens** when the parent dies before the child ???

A Process Uses Certain Resources:

- Processor time on a CPU
- Memory, both virtual or real
- File descriptors
- . . .

• Subprocesses

UNIX is a **multi-process** operating systems

- **Many processes** run at the same time
- **Processes** can be created and can be terminated

Processes form a hierarchy

- All **processes** have a **unique parent**
- In the end, all (real) processes descent from the **init** process

Parent and child **share** a special relationship

- The parent has to retrieve the termination status of a process
- The child can get his parents process id
- If a parent dies, its special role is taken over by the **init** process

Process Properties

For each process, we can get various identifiers:

- The **process id** : **assigned by the kernel** and it usually between 2 and MAX_INT except for **init** which has pid 1 - once a process completes execution, pid goes back into the available pool The process id of the parent

- The **real user id** of the process (i.e. the user id of the owner)

- Parent Process ID **(ppid)**

- The **effective user id** of the process (i.e. the user id that is used to check access rights). It can differ e.g. for programs with the setuid bit set

- The **real group id**

- The **effective group id**

- **Address Space** : Memory locations addressable by the process - Has code and data segments - Normally, cannot access addresses outside the address space (**Segmentation Fault**)

- **Status**

- **CPU time**

- **Priority**

- **Open File Descriptors**

- **File Descriptor** table maintained by the kernel
- … more

- The kernel maintains a table called the ***process table*** where it ***stores information*** about all processes in the system

- This table is used in **scheduling** and **allocation** of processes

- The **ps** command displays information from this table

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

pid_t   getpid(void);      /* Get process id */
pid_t   getppid(void);     /* Get parent process id */
uid_t   getuid(void);      /* Get real user id */
uid_t   geteuid(void);     /* Get effective user id */
gid_t   getgid(void);      /* Get real group id id */
gid_t   getggid(void);     /* Get effective group id */
```

Starting a New Process

- Three ways to start a child process from within a C program
  - **system()**
  - **fork()**
  - **exec()**
- Depending on the method, the child may inherit different attributes from its parent

Running Other Programs: system()

- The **system()** function is defined by ANSI C
- **The function system() will invoke the command processor to execute a command.** **Once** the command execution is **terminated** the processor will give the control back to the program that has called the system command.
- **system()** hands the string pointed to by command to the systems **command processor** for execution
  - **system()** returns, when the command returns.
  - Return value of **system()** in this case is implementation defined
- If **command** is **NULL**, **system()** checks if the implementation **has** a command processor
  - It returns 0, if not
  - Anything else, otherwise

Example

```
#include <iostream>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main() {
	char command[50];

	strcpy( command, "ls -l");
	printf("Executing command ls\n\n");
	int i = system(command);
	printf("\nReturned value is: %d.\n",
	 i);

	return (0);
}
```

Sample Run

Executing command ls

drwxr-xr-x 36 husaingholoom staff 1224 Sep 15 16:23 ZF2020Examples

-rwxr-xr-x 1 husaingholoom staff 8920 Oct 29 22:19 a.out

drwxr-xr-x 3 husaingholoom staff 102 Sep 24 2019 a.out.dSYM

drwxr-xr-x 4 husaingholoom staff 136 Feb 10 2020 aaaaaa

-rwx------ 1 husaingholoom staff 106 Feb 17 2019 am.sh

-rwxr-xr-x 1 husaingholoom staff 2267 Mar 21 2019 ei

-rwxr-xr-x 1 husaingholoom staff 276 Apr 7 2020 example.c

-rwxr-xr-x 1 husaingholoom staff 375 Feb 12 2019 f1

.

.

Returned value is: 0.

Example - 2

```
#include <iostream>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main() {
	char command[50];

	strcpy( command, "lx -l");
	printf("Executing command ls\n\n");
	int i = system(command);
	printf("Returned value is: %d.\n", i);

	return (0);
}
```

Sample Run

Executing command ls

-bash: lx: command not found

Returned value is: 32512.

Note : the Return value is system dependant.

system() in UNIX

On UNIX, there **always** is a command processor

- The command is handed to the standard shell, **/bin/sh**
- It can make use of all shell facilities, including I/O redirection

The return value of the **system()** command normally is an encoding of the **exit status** of the executed command

- If the shell **cannot be executed**, it is treated as if the shell returned **127 (system dependent)**
- If for some reason no new process for the shell can be created, **-1** is returned (and **errno** is set to specify what went wrong)
- Otherwise, the return value is an encoding of the **exit status** of the shell

system()

Header

#include<stdlib.h>

Prototype

int system (const char *command);

- Starts the process denoted by command, within a shell

  /bin/sh -c command

- Returns
  - 127 if it fails to create the shell
  - - 1 if it fails for some other reason
  - exit status of the executed process if it succeeds

Problems with system()

Using **system()** is **not** the best way to launch a child process

- **Inefficient** because creates a new shell before executing the process
- **Less portable because commands might function differently under different shells**
- **Blocks** the current process until the child process terminates
- Provides **limited control** over running of the commands
- Need a named binary to execute

UNIX User Commands :

ps (short for "process status")

Usage: **ps** <complicated options>

– ps shows **information** about **currently executing processes**
– It is one of **the least** standardized UNIX tools

Our Linux ps can assume many different **personalities**

– Different personalities show different behavior
– . . . and accept different options.

Default behavior (ps without options):

– **Show information about all existing processes** of the current user controlled by the same terminal ps was run on

– For each process, list:

_ Process Id (PID)
_ Controlling terminal (TTY)
_ CPU time used by the process
_ Name of the executable program file

ps Example

```
[hag10@zeus ~]$ ps
 PID     TTY     TIME       CMD
 5789    pts/6   00:00:00   bash
 6316    pts/6   00:00:00   top
 27462   pts/6   00:00:00   ps
[hag10@zeus ~]$
```

Some ps Options

Some simple BSD style options for the default personality (note: BDS style options for ps are **not** preceeded by a dash!)

– **a**: Print information about **all processes** that are connected to any terminal

– **x**: Print information about processes **not connected** to a terminal

– **u** <username>: Print information about processes owned by the **named user u.**

It is user oriented output with more interesting information:

- Owner of a process (**USER**)
- Process Id (**PID**)
- Percentage of **available CPU** used by the process (**%CPU**)
- Percentage of **memory used** (%MEM) (note that this measures virtual memory usage, real memory usage may be lower because of shared pages)
- **Virtual memory size** of the process in KByte (VSZ)
- Size of the **resident set**, i.e. the recently referenced pages not swapped out (RSS)
- Controlling terminal (TTY)
- Ti**me or date when the process was started** (START)
- Seconds of CPU time used (**TIME**)
- Full command used to start the process (COMMAND)

ps u Example

```
[hag10@zeus ~]$ ps u
USER      PID  %CPU %MEM   VSZ     RSS   TTY     STAT START   TIME COMMAND
hag10     5789  0.0   0.1    121552 4352 pts/6   Ss    08:56   0:00  -bash
hag10     6316  0.0   0.0    25784  1300 pts/6   T     08:57   0:00 top
hag10    30295  0.0   0.0    121112 1260 pts/6   R+    09:51   0:00 ps u
[hag10@zeus ~]$
```

To print all running processes in system, use any one of the following commands.

```
$ps -A
$ps -e
```

$ ps –e

```
  PID TTY        TIME CMD
    1 ??      0:22.00 /sbin/launchd
   10 ??      0:01.44 /usr/libexec/kextd
   11 ??      0:00.67 /usr/libexec/UserEventAgent -l System
   12 ??      0:03.17 /usr/sbin/notifyd
   13 ??      0:00.35 /usr/sbin/diskarbitrationd
   14 ??      0:03.61 /usr/libexec/configd
   15 ??      0:07.76
/System/Library/Frameworks/CoreServices.framework/Vers
……..
```

Interesting ps Example : ps aux

Print all the running process in system regardless from where they have been executed.

aux command options

a:- This option **prints the running processes** from all users.

u:- This option shows **user or owner column** in output.

x:- This option prints the **processes those have not been executed from the terminal.**

[hag10@zeus ~]$ ps aux

```
USER     PID   %CPU  %MEM  VSZ     RSS    TTY    STAT START  TIME  COMMAND
hag10    398   0.0   0.0   108092  900    pts/6  R+   09:57  0:00  ps aux
ns1254   3801  0.0   0.0   108108  1456   ?      S    Nov03  0:00  /bin/sh ./train
wg1075   3900  0.0   0.0   115892  2192   ?      S    Nov03  0:01  sshd: wg1075@pt
wg1075   3901  0.0   0.1   121556  4368   pts/4  Ss   Nov03  0:00  -bash
wg1075   3960  0.0   0.3   200800  15028  pts/4  S+   Nov03  0:00  sqlplus
hag10    5788  0.0   0.0   115896  1988   ?      S    08:56  0:00  sshd: hag10@pts
hag10    5789  0.0   0.1   121552  4352   pts/6  Ss   08:56  0:00  -bash
ns1254   5845  0.0   0.0   108108  1452   ?      S    Nov02  0:00  /bin/sh ./train
hag10    6316  0.0   0.0   25784   1300   pts/6  T    08:57  0:00  top
ns1254   30674 0.2   0.3   195868  14340  ?      S    09:52  0:00  python rr.py
ns1254   30675 0.0   0.0   100904  600    ?      S    09:52  0:00  sleep 360
jma105   31527 0.0   0.0   116440  2580   ?      S    Nov04  0:00  sshd: jma105@pt
jma105   31528 0.0   0.0   71344   2336   ?      Ss   Nov04  0:00  /usr/libexec/op
jma105   32535 0.0   0.1   121540  4400   pts/0  Ss+  Nov04  0:00  -bash
[hag10@zeus ~]$
```

An alternative set of options for viewing all the processes running on a system is

ps -ef

- The ***-e option*** generates a list of information about every process currently running.
- The ***-f option*** generates a listing that contains fewer items of information for each process than the -l option.

Example

[hag10@zeus ~]$ ps -ef

UID	PID	PPID	C	STIME	TTY	TIME	CMD
d_b123	3603	3508	0	10:04	?	00:00:00	sshd: d_b123@pts/8
d_b123	3604	3603	0	10:04	?	00:00:00	/usr/libexec/openssh/sftp-server
ns1254	3801	1	0	Nov03	?	00:00:00	/bin/sh ./training.sh
wg1075	3900	3898	0	Nov03	?	00:00:01	sshd: wg1075@pts/4
wg1075	3901	3900	0	Nov03	pts/4	00:00:00	-bash
wg1075	3960	3901	0	Nov03	pts/4	00:00:00	sqlplus
d_b123	4067	3603	0	10:05	pts/8	00:00:00	-bash
hag10	5788	5728	0	08:56	?	00:00:00	sshd: hag10@pts/6
hag10	5789	5788	0	08:56	pts/6	00:00:00	-bash
ns1254	5845	1	0	Nov02	?	00:00:00	/bin/sh ./training.sh
ns1254	6272	3801	0	10:10	?	00:00:00	python rr.py
ns1254	6273	5845	0	10:10	?	00:00:00	sleep 360
hag10	6316	5789	0	08:57	pts/6	00:00:00	top
hag10	8534	5789	0	10:15	pts/6	00:00:00	ps -ef
jma105	31527	31444	0	Nov04	?	00:00:00	sshd: jma105@pts/0
jma105	31528	31527	0	Nov04	?	00:00:00	/usr/libexec/openssh/sftp-server
jma105	32535	31527	0	Nov04	pts/0	00:00:00	-bash

[hag10@zeus ~]$

You can use system to write a program to run ps.

Example

```
#include <stdlib.h>
#include <stdio.h>
int main() {
	printf("Running ps with system\n");
	system("ps ax");
	printf("Done.\n");
	exit(0);
}
```

The system function runs the command passed to it as a string and **waits** for it to complete

Sample Run

./a.out

Running ps with system

PID	TTY	STAT	TIME	COMMAND
3801	?	S	0:01	/bin/sh ./training.sh
5845	?	S	0:01	/bin/sh ./training.sh
16470	?	S	0:01	sshd: jrs224@pts/0,pts/2
16578	pts/2	Ss	0:00	-bash
17066	pts/0	Ss	0:00	-bash
17097	pts/0	S+	0:00	nano linkedlistdraft.h
17438	pts/2	S+	0:00	nano bst.h
18053	?	S	0:00	sshd: marshallr@pts/5
18054	pts/5	Ss+	0:00	/home/LabStaff/marshallr/bin/zsh -l
18134	?	S	0:00	sshd: jrs224@pts/1,pts/3
18135	pts/1	Ss	0:00	-bash
18147	pts/3	Ss+	0:00	-bash
18241	?	S	0:00	sshd: hag10@pts/4
18242	pts/4	Ss	0:00	-bash
18295	pts/1	S+	0:00	nano bst.h
18336	?	S	0:00	python rr.py
18337	?	S	0:00	sleep 360
18347	pts/4	S+	0:00	./a.out
18348	pts/4	R+	0:00	ps ax

Done.

[hag10@zeus ~]$

In the above example , the program calls system with the string **“ps ax”,** which executes the **ps** program.

The program returns from the call to system **when the ps command has finished** and the function continues until the end. **(Done)**

The system function can be quite useful but is also **limited**. Because the **program has to wait** until the process started by the call to system finishes, you can’t get on with other tasks.

ps provides codes indicating the current status.

Common codes are given in the following table

STAT Code	Description
S	**Sleeping**. Usually waiting for an event to occur, such as a signal or input to become available.
R	**Running**. Strictly speaking, “runnable,” that is, on the run queue either executing or about to run.
D	**Uninterruptible Sleep** (**Waiting**). Usually waiting for input or output to complete.
T	**Stopped**. Usually stopped by shell job control or the process is under the control of a debugger.
Z	Defunct or “**zombie**” process.
N	**Low** priority task, “nice.”
W	**Paging**. (Not for Linux kernel 2.6 onwards.)
s	Process is a **session** leader.
+	Process is in the **foreground** process group.
L	Process is **multithreaded**.
<	**High** priority task

In general, using system is a *far* from ideal way to start other processes, because it invokes the desired program using a shell

Exiting

Normal ways of terminating a program :

Calling **return st**; from **main()** (ANSI C)

- In that case the **exit status** of the program is st
- Interpretation of the exit status is implementation-defined for ANSI C (but defined for UNIX)

Calling **exit**(st); **from anywhere** in the program (ANSI C)

– Exit status is st
– In **main(), exit()** and **return** are equivalent
– In both cases, some cleanup actions are performed
  - _ **Exit handlers** are called
  - _ All open files are flushed and closed

Calling _exit(st) (UNIX) or _Exit(st) (new in ANSI-C 99, may not be widely supported)

– Program is immediately terminated
– Exit status is **st**

Exit Formalities

#include <stdlib.h>

void   exit(int status);
void _Exit(int status);   /* New in C99 */

#include <unistd.h>

void _exit(int status);

ANSI C defines three different exit statuses:

– **EXIT_SUCCESS** (in stdlib.h)
– **EXIT_FAILURE** (in stdlib.h)
– **0**   (equivalent to EXIT SUCCESS)

In practice, **EXIT_SUCCESS** is nearly always just #defined as 0

Cleaning up: atexit()

The **atexit()** function registers the given function to be called **at normal program termination**, whether via **exit(3)** or via **return** from the program's main.

Functions so registered are **called in the reverse order** of their registration; no arguments are passed.

ANSI C allows us to register up to **32 functions** that will be called **whenever** the **program terminates normally**:

```
#include <stdlib.h>

int atexit(void (*func)(void));
```

- Argument is a pointer to a function that **neither takes an argument nor returns a value**
- Return value for **atexit()** is **0** on success, **-1** on error

Each call to **atexit()** results in a single call to the registered function

- Registered functions are called in reverse order of registration

- We can register the same function more than once

Note: Exit handlers should only access global variables

Example

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

int handler_counter=0;

void err_sys(char* message)
{
	perror(message);
	exit(EXIT_FAILURE);
}

void handler1(void)
{
	printf("Handler1, counter = %d\n", handler_counter);
	handler_counter++;
}

void handler2(void)
{
	printf("Handler2, counter = %d\n", handler_counter);
	handler_counter++;
}
```

Example - continue

```
int main(void)
{
	if(atexit(handler1) != 0)
	{
		err_sys("atexit");
	}

	if(atexit(handler2) != 0)
	{
		err_sys("atexit");
	}

	if(atexit(handler1) != 0)
	{
		err_sys("atexit");
	}

	if(atexit(handler1) != 0)
	{
		err_sys("atexit");
	}

printf("My PID is %d and my parents PID is %d\n", getpid(),
	getppid());

return EXIT_SUCCESS;

}
```

Example Output

My PID is 2012 and my parents PID is 746

Handler1, counter = 0

Handler1, counter = 1

Handler2, counter = 2

Handler1, counter = 3

Termination Status Interpretation

Termination status can come from multiple sources

– **system()** (which nicely packs up all the work for us)
– Functions that retrieve the exit status of a child process: **wait()** and **waitpid()** (more later)

Interpretation depends on the cause of the termination of the child process.

Assume that **status** is the termination status

– If **WIFEXITED(status)** is **true**, the process **terminated normally** (i.e. via **exit(), _exit() or return** from main)

_ **WEXITSTATUS(status)** returns the (lower 8 bit of) the value that was passed to **exit()**

– If **WIFSIGNALED(status)** is **true**, the process was terminated because of an uncaught **signal** with **default action abort**

_ **WTERMSIG(status)** gives the number of the signal

- **If WIFSTOPPED(status)** is **true**, the process is **currently stopped** (via **SIGSTOP or SIGSTP)**.

  - **WSTOPSIG(status)** returns the number of the stop signal

Continue - Program and Processes

Creating new Processes: fork()

fork() : System call to create a child process.

Fork system call : Is a system call that creates a new process **identical** to the calling one.

Whenever a fork() system call is called:

- **A copy of** all the pages(memory) related to the parent process is created and loaded into a separate memory location by the Operating System for the child process
- Makes a copy of text, data, stack, and heap
- Starts executing on that new copy

Simple Example :

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

int main() {

    // make two process which run same
    // program after this instruction

    fork();

    printf("Hello world!\n");
    return 0;
}
```

Sample Run

Hello world!
Hello world!

Uses of fork()

- To create a parallel program with multiple processes .

- It creates a new process, which becomes the child process of the caller

- It takes no arguments .

- It returns a process ID. Normally, the process ID is an integer.

- After a new child process is created, both processes will execute the next instruction following the fork() system call.

- **A** child process differs from its parent process **only** in **pid(process ID)** and **ppid(parent process ID)**.

- In order to distinguish the parent process from the child process, You must test the returned value of fork() system call

    - If fork() **returns a negative value**, the creation of a child process was **unsuccessful**.
    - fork() **returns a zero** to the newly created **child process**.
    - fork() **returns a positive value**, for the **parent** process.

- A process can use function getpid() to retrieve the process ID assigned to this process.

Must include the following libraries :

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

pid_t   fork(void);
```

Example

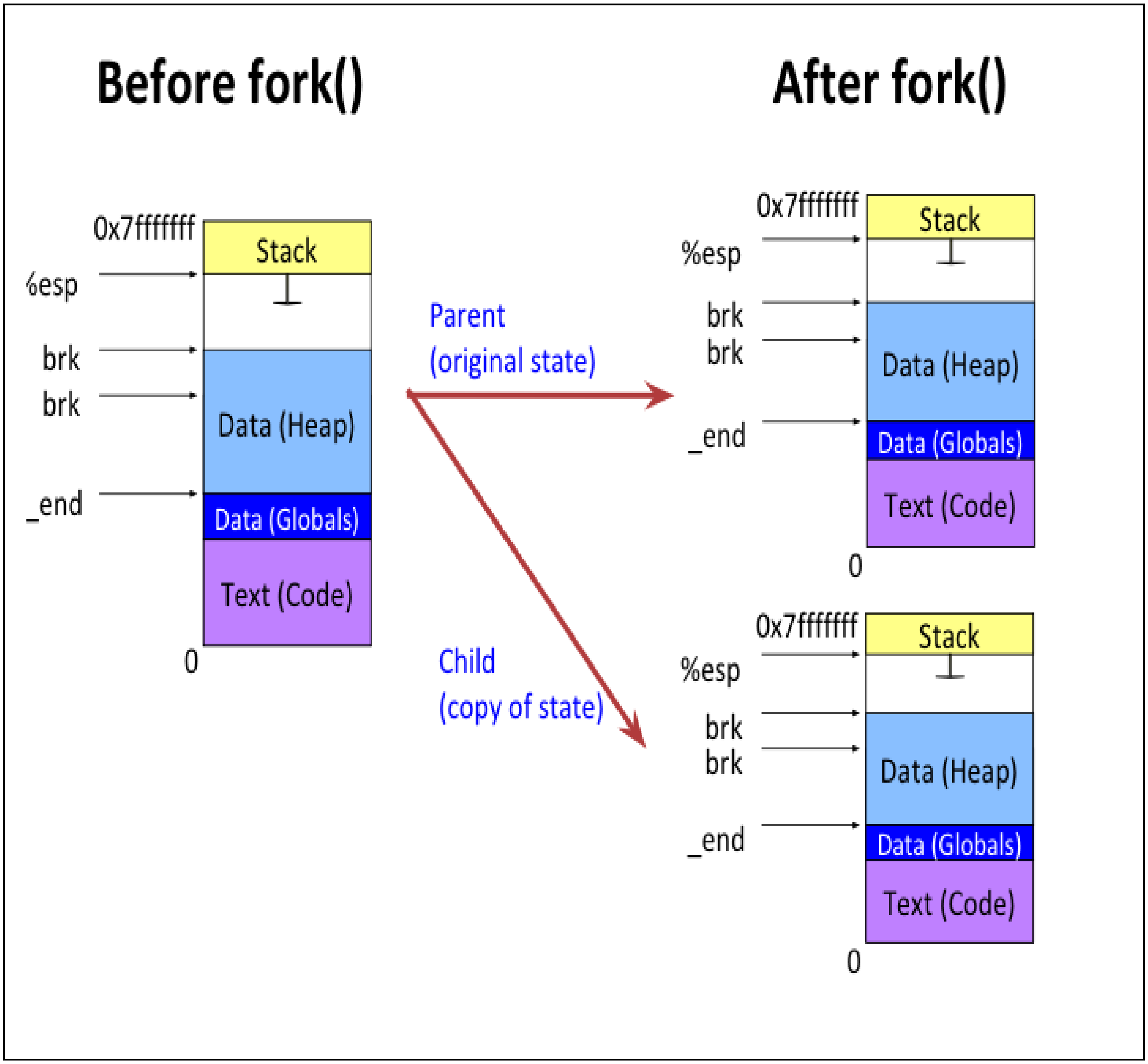
```c
#include <stdio.h>
#include <unistd.h>

int main() {

	int seq = 0;

	if (fork() == 0) {
		printf("Child! Seq=%d\n", ++seq);
	} else {
		printf("Parent! Seq=%d\n", ++seq);
	}
	printf("Both! Seq=%d\n", ++seq);
	return 0;
}
```

Sample run

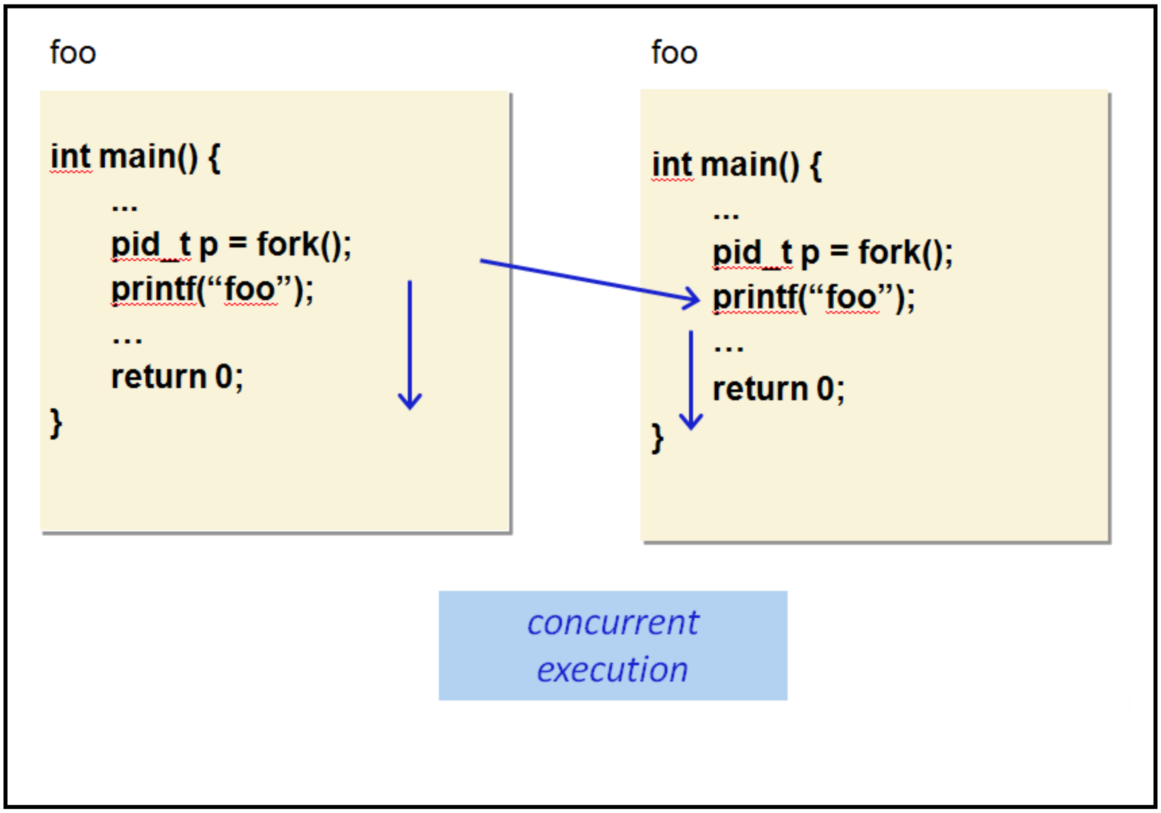
[hag10@zeus ~]$ ./a.out

Parent! Seq=1
Both! Seq=2
Child! Seq=1
Both! Seq=2

Since fork() creates a new **child** process that is in nearly all ways an exact copy of the parent , a new **entry in the process table** is created for the new process and the execution continues in both parent and child



fork() Control



- **fork()** returns the **pid of the child process** to the parent
- **fork()** returns 0 to the child process
- Allows parent and child to distinguish themselves
- The current process continues running from the instruction past the **fork()** call
- The child process starts executing from the instruction after the **fork()** call

Example To show the return value.

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

main()
{
	int i;

	printf("Ready to fork...\n");
	i=fork();
	printf("Fork returned %d\n",i);
	while (1);
}
```

Sample Run

Ready to fork...
Fork returned 663 // Parent PID
Fork returned 0 // Child PID

Fork() and getpid()

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
int main() {
	int i;

	i = getpid();
	printf("\n\nBefore the Fork  pid is   =   %d \n", i);
	int s = fork();
	if (s == 0) {
		int child_pid = getpid();
		printf("\nI am the Child Process  ? %d \n", s);
		printf("\nChild Process ID =    %d\n", child_pid);
	} else {
		int i = getpid();
		printf("\nI am the Parent Process  ? %d \n", i);
		printf("\nparent Process ID  =   %d\n", i);
	}
	return EXIT_SUCCESS;
}
```

Sample Run

Before the Fork pid is = 963

I am the Parent Process ? 963

Parent Process ID = 963

I am the Child Process ? 0

Child Process ID = 964

What is the output of the following Program

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

int main()
{
        int i;

        i=getpid();
        printf("Parent= %d \n",i);
        fork();
        fork();
        i=getpid();
        printf("Who am I ?  %d\n",i);
        return EXIT_SUCCESS;
}
```

Sample Run

Parent = 1091
Who am I ?   1092
Who am I ?   1093
Who am I ?   1091
Who am I ?   1094

How do Parent / Child Run in Parallel?

- **Obvious answer**: by running on different processors on a multiprocessor system
- What if it's a uniprocessor system?
- What if other processors are already busy?
- **Answer**: by **context switching** (running each process in short time slots)

Comments on fork()

- On modern UNIX versions, fork() is implemented with **copy on write**
    - Both processes actually **share the same pages** in memory
    - Only when a process actually **tries to change a value in memory is a private copy created**

    - **Consequence**: Forking is very cheap – it only has to copy basic process structures

- Forked processes **behave** as if an actual copy has been made

- All of the processes memory is accessible in both parent and child
    - Changing them in one does not affect the other

- *** **Order of execution for parent and child is unpredictable!** ***

- In **many** situations, when a parent **forks()** a child process, the parent needs to **wait** for the child to die before moving on
    - Child **produces** some data that the parent needs to use
    - Child has **some effect on the system** that is critical for the parent to function properly
- **UNIX programmers use forking a lot!**
    - **Servers** may fork one process for each connection!
    - Shells fork for executing commands

Example

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

void err_sys(char* message) {
        perror(message);
        exit(EXIT_FAILURE);
}

int main(int argc, char* argv[]) {
        pid_t pid, ppid, child_pid;
        int some_var = 42;
        pid = getpid();
        printf("Parent. My PID is %d and I am about to
                procreate\n", pid);
        child_pid = fork();
        if (child_pid < 0) {
                err_sys("fork");
        }


        if (child_pid == 0) {
                pid = getpid();
                ppid = getppid();
                printf("Child. My PID is %d, my parent is   %d\n", pid, ppid);
                printf("Child: some_var=%d - Changing it  now!\n", some_var);
                some_var = 7;
                printf("Child: some_var=%d\n", some_var);
        }

        else {
```

```
        printf("Parent. My PID is %d, my child is %d\n", pid, child_pid);
        printf("Parent: some_var=%d\n", some_var);
        printf("Going to sleep now, waiting for the  child process to die...\n");
        sleep(5);
        printf("I'm awake again. some_var is still   %d\n", some_var);
    }
    return EXIT_SUCCESS;
}
```

Example Output

192:fork husaingholoom$ ./a.out

Parent. My PID is 1183 and I am about to procreate

Parent. My PID is 1183, my child is 1184
Parent: some_var=42
Going to sleep now, waiting for my child to die...

Child. My PID is 1184, my parent is 1183
Child: some_var=42 - Changing it now!
Child: some_var=7

Parent -- I'm awake again. some_var is still 42

192:fork husaingholoom$

Unix Linux

```
[hag10@zeus ~]$
[hag10@zeus ~]$
[hag10@zeus ~]$ ./a.out
Parent. My PID is 27880 and I am about to  procreate
Parent. My PID is 27880, my child is 27881
Parent: some_var=42
Going to sleep now, waiting for the  child process to die...
Child. My PID is 27881, my parent is   27880
Child: some_var=42 - Changing it  now!
Child: some_var=7
I'm awake again. some_var is still   42
[hag10@zeus ~]$
```

What is the exact output of the following program

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

int main()
{
        int i,j;
        j=0;

        printf("Ready to fork...\n");
        i=fork();
        if (i == 0)
        {
                printf("The child executes this code.\n");
                for (i=0; i<5; i++)
                        j=j+i;
                printf("Child j=%d\n",j);
        }
        else
        {
                printf("The parent executes this code.\n");
                for (i=0; i<3; i++)
                        j=j+i;
                printf("Parent j=%d\n",j);
        }
}
```

Don't Do This!

```
#include <unistd.h>
int main(int argc, char* argv[])
{
    while(1)
    {
        fork();
    }
}
```

It is the simplest version of a **fork bomb**

– Will create an exponentially growing number of Processes

– Quickly consumes all system resources

– Makes system essentially unusable

Buffered I/O

- Buffered I/O refers to the technique of temporarily **storing** the results of an **I/O** operation in user-space before transmitting it to the kernel or **before providing it to your process**
- Buffering the data **can minimize the number of system calls** and can block-align I/O operations, which **may improve the performance** of your application.
- For example, consider a process that writes one character at a time to a file. This is obviously inefficient: Each write operation corresponds to a `write()` system call, which means a trip into the kernel, a memory copy (of a single byte!), and a return to user-space, only to repeat the whole ordeal.
- Buffered I/O avoids this inefficiency by buffering the writes in a data buffer in user-space until a certain threshold is reached.
- In the previous example, copy each character into the buffer and call `write()` only when the block size is reached.

Example: Buffered I/O and Forking

/* Usual includes and stuff omitted */

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

void err_sys(char* message) {
	perror(message);
	exit(EXIT_FAILURE);
}

int main(int argc, char* argv[]) {
	pid_t child_pid;
	printf("Hihello   "); // <--- Note: No Newline!
	child_pid = fork();
	if (child_pid < 0) {
		err_sys("fork");
	}

	if (child_pid == 0) {
		printf("from the child!\n");
	} else {
		printf("from the parent!\n");
		sleep(1);
	}

	return EXIT_SUCCESS;
}
```

Example Output

Hihello   from the parent!
Hihello   from the child!

What is the output of the following Program:

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

void forkexample() {
	if (fork() == 0)
		printf("Hello from Child!\n");
	else
		printf("Hello from Parent!\n");
}
int main() {
	forkexample();
	return 0;
}
```

What is the output of the following Program:

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

int main() {
	printf("L0\n");
	fork();
	printf("L1\n");
	fork();
	printf("Bye\n");

	return 0;
}
```

Sample Run

L0
L1
L1
Bye
Bye
Bye
Bye

In Zeus :

```
[hag10@zeus F2024]$
[hag10@zeus F2024]$
[hag10@zeus F2024]$ ./a.out
L0
L1
L1
Bye
Bye
Bye
Bye
[hag10@zeus F2024]$
```

What is the output of the following Program:

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

#define   MAX_COUNT  10
#define   BUF_SIZE   100

int main(void) {
        pid_t pid;
        int i;
        char buf[BUF_SIZE];

        fork();
        pid = getpid();

        for (i = 1; i <= MAX_COUNT; i++) {
                sprintf(buf, "This line is from pid %d, value =  %d\n", pid, i);
                write(1, buf, strlen(buf));
        }

        return 0;
}
```

Sample Run

This line is from pid 1385, value = 1
This line is from pid 1385, value = 2
This line is from pid 1385, value = 3
This line is from pid 1385, value = 4
This line is from pid 1385, value = 5
This line is from pid 1385, value = 6
This line is from pid 1385, value = 7
This line is from pid 1385, value = 8
This line is from pid 1385, value = 9
This line is from pid 1385, value = 10
This line is from pid 1386, value = 1
This line is from pid 1386, value = 2
This line is from pid 1386, value = 3
This line is from pid 1386, value = 4
This line is from pid 1386, value = 5
This line is from pid 1386, value = 6
This line is from pid 1386, value = 7
This line is from pid 1386, value = 8
This line is from pid 1386, value = 9
This line is from pid 1386, value = 10

In Zeus

```
[hag10@zeus F2024]$
[hag10@zeus F2024]$
[hag10@zeus F2024]$ ./a.out
This line is from pid 2757650, value =  1
This line is from pid 2757650, value =  2
This line is from pid 2757650, value =  3
This line is from pid 2757650, value =  4
This line is from pid 2757650, value =  5
This line is from pid 2757650, value =  6
This line is from pid 2757650, value =  7
This line is from pid 2757650, value =  8
This line is from pid 2757650, value =  9
This line is from pid 2757650, value =  10
This line is from pid 2757651, value =  1
This line is from pid 2757651, value =  2
This line is from pid 2757651, value =  3
This line is from pid 2757651, value =  4
This line is from pid 2757651, value =  5
This line is from pid 2757651, value =  6
This line is from pid 2757651, value =  7
This line is from pid 2757651, value =  8
This line is from pid 2757651, value =  9
This line is from pid 2757651, value =  10
[hag10@zeus F2024]$
[hag10@zeus F2024]$
```

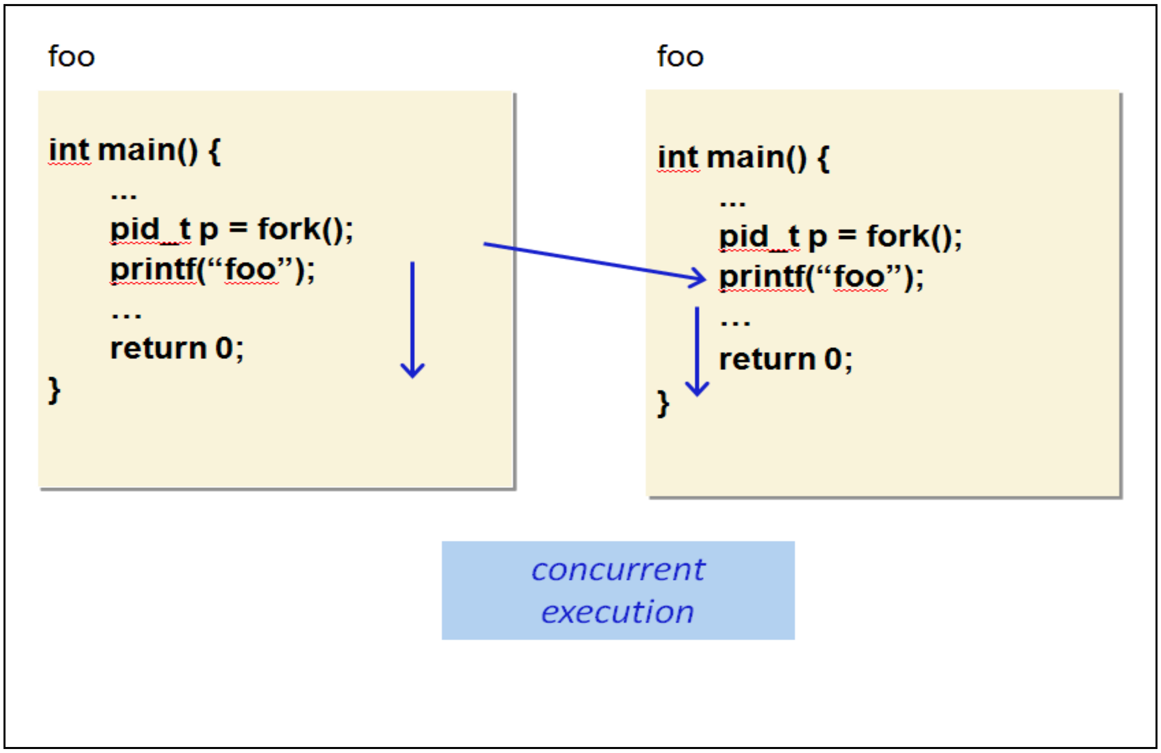
exec() Functions

– The exec() family of functions ***replaces*** the current (invoking) process image with a new (invoked) process image. **It replaces the program in the current process with a brand new program.**

  the created **child process does not have to run the same program as the parent** process does. The **exec** type system calls allow a process to run any program files, which include **a binary executable or a shell script**

– The process image includes
  - text
  - data
  - heap
  - stack

– By default the parents environment is passed on to the child process


foo
int main() {
...
pid_t p = fork();
printf("foo");
...
return 0;
}
foo
int main() {
...
pid_t p = fork();
printf("foo");
...
return 0;
}
concurrent execution

exec() Functions

Inherited Values

- Process ID and the parent process ID
- Real user ID and group ID
- Current working directory
- File descriptors if FD_CLOEXEC is not set
- Streams are closed but their descriptors are not changed and can be reopened by fdopen()
- File mode creation mask
- Elapsed processor time
- . . .

exec

#include<unistd.h>

int **execl** (const char ***path**, const char *arg, ...)

- **Execute the executable file named by path.**
- Only return -1 if any error and set the global **errno**.
- Arguments are provided by arg.
- The last element of this array must be a null pointer.

exec family functions:

#include<unistd.h>

int execl(const char ***path**, const char *arg, ...);
int execlp(const char ***file**, const char *arg, ...);
int execv(const char ***path**, char *const argv[]);
int execvp(const char ***file**, char *const argv[]);
int execle(const char ***path**, const char *arg, ..., char * const envp[]);

e: It is an array of pointers that points to environment variables and is passed explicitly to the newly loaded process.

l: l is for the command line arguments passed a list to the function

p: p is the path environment variable which helps to find the file passed as an argument to be loaded into process.

v: v is for the command line arguments. These are passed as an array of pointers to the function.

It should be noted here that these functions have the same base exec followed by one or more letters.

In General

- If **path** is used, the program must be locatable in the **path or the current directory.**

- If **file** is used, the program must be **locatable in the file or the PATH**.

- The program must be **executable**.

- The **first argument** is normally the **name** of the program.

- The exec function **does not (cannot) return** on success.

path is used to specify the **full path name of the file** which is to be executes.

arg is the argument passed. **It is the name of the file which will be executed** in the process. Most of the times the value of arg and path is same.

const char* arg in functions execl(), execlp() and execle() is considered as arg0, arg1, arg2, …, argn**. It is basically a list of pointers to null terminated strings**. Here the first argument points to the filename which will be executed.

envp is an array which contains pointers that point to the environment variables.

file is used to specify the path name which will identify the path of new process image file

Unchanged values

- The process ID and the parent process ID.
- Real user ID and group ID.
- Current working directory and root directory.
- File descriptors if FD_CLOEXEC is not set.
- Streams are closed but their descriptors are not changed and can be reopened by fdopen().
- File mode creation mask.

Difference between fork() and exec() system calls:

The fork() system call is used to **create** an exact copy of a running process and the created copy is the child process and the running process is the parent process.

Whereas, exec() system call is used to **replace** a process image with a new process image. Hence there is no concept of parent and child processes in exec() system call.

In fork() system call the parent and child processes are executed at the same time. But in exec() system call, if the replacement of process image is successful, the **control does not return** to where the exec function was called rather it will execute the new process. The control will only be transferred back if there is any error.

Example : Using execlp()

```
#include <unistd.h> // execlp()
#include <stdio.h>  // perror()
#include <stdlib.h> // EXIT_SUCCESS, EXIT_FAILURE

int main(void) {
  execlp("ls", "ls", "-l", NULL);
  perror("Return from execlp() not expected");
  exit(EXIT_FAILURE);
}
```

Sample Run

```
.
.

-rwxr-xr-x    1 husaingholoom  staff    307 Nov 17 16:32 ztest2.c
-rwxr-xr-x    1 husaingholoom  staff   1296 Nov 17 16:49 ztest3.c
-rwxr-xr-x    1 husaingholoom  staff    763 Nov 17 17:05 ztest4.c
-rwxr-xr-x    1 husaingholoom  staff   1402 Nov 17 17:39 ztest5.c
-rwxr-xr-x    1 husaingholoom  staff    648 Nov 17 19:26 ztest6.c
-rwxr-xr-x    1 husaingholoom  staff    871 Nov 17 18:36 ztest7.c.
.
.
```

Example : Using execvp()

```
#include <unistd.h> // execvp()
#include <stdio.h>  // perror()
#include <stdlib.h> // EXIT_SUCCESS, EXIT_FAILURE

int main(void) {
  char *const cmd[] = {"ls", "-l", NULL};
  execvp(cmd[0], cmd);
  perror("Return from execvp() not expected");
  exit(EXIT_FAILURE);
}
```

Sample Run

.
.

```
-rwxr-xr-x   1 husaingholoom  staff    307 Nov 17 16:32 ztest2.c
-rwxr-xr-x   1 husaingholoom  staff   1296 Nov 17 16:49 ztest3.c
-rwxr-xr-x   1 husaingholoom  staff    763 Nov 17 17:05 ztest4.c
-rwxr-xr-x   1 husaingholoom  staff   1402 Nov 17 17:39 ztest5.c
-rwxr-xr-x   1 husaingholoom  staff    648 Nov 17 19:26 ztest6.c
-rwxr-xr-x   1 husaingholoom  staff    871 Nov 17 18:36 ztest7.c.
```

.
.

Example : Using execl()

The following example executes the *ls* command, specifying the pathname of the executable (**/bin/ls**) and using arguments supplied directly to the command to produce single-column output.

```
#include <unistd.h> // execvp()
#include <stdio.h>  // perror()
#include <stdlib.h> // EXIT_SUCCESS, EXIT_FAILURE
int main(void) {

        int ret;
        execl("/bin/ls", "ls", "-l", (char *) 0);
        perror("Return from execl() not expected");
        exit(EXIT_FAILURE);


}
```

Sample Run

.

.

.

```
-rwxr-xr-x    1 husaingholoom  staff    871 Nov 17 18:36 ztest7.c
-rwxr-xr-x    1 husaingholoom  staff    934 Nov 17 18:44 ztest8.c
-rwxr-xr-x    1 husaingholoom  staff   1814 Nov 19 18:58 ztest9.c
-rw-r--r--    1 husaingholoom  staff    321 Nov  5 17:31 zwait0.c
-rw-r--r--    1 husaingholoom  staff    966 Nov  5 17:50 zwait1.c
```

.

.

.

Example : Using execle()

The following example is similar to Using execl(). In addition, it specifies the environment for the new process image using the *env* argument.

#include <unistd.h>

```
#include <unistd.h> // execle()
#include <stdio.h>  // perror()
#include <stdlib.h> // EXIT_SUCCESS, EXIT_FAILURE
int main(void) {

        char *env[] = { "HOME=/usr/home",
                                "LOGNAME=home", (char *) 0 };

        execle("/bin/ls", "ls", "-l", (char *) 0, env);

        perror("Return from execle() not expected");
        exit(EXIT_FAILURE);

}
```

Sample Run

.

.

.

```
-rwxr-xr-x    1 husaingholoom  staff    871 Nov 17 18:36 ztest7.c
-rwxr-xr-x    1 husaingholoom  staff    934 Nov 17 18:44 ztest8.c
-rwxr-xr-x    1 husaingholoom  staff   1814 Nov 19 18:58 ztest9.c
-rw-r--r--    1 husaingholoom  staff    321 Nov  5 17:31 zwait0.c
-rw-r--r--    1 husaingholoom  staff    966 Nov  5 17:50 zwait1.c
```

.

.

.

Example : Using execv()

The following example passes arguments to the *ls* command in the *cmd* array.

#include <unistd.h>

```
#include <unistd.h> // execv()
#include <stdio.h>  // perror()
#include <stdlib.h> // EXIT_SUCCESS, EXIT_FAILURE
int main(void) {
        char *cmd[] = { "ls", "-l", (char *)0 };
        execv ("/bin/ls", cmd);
        perror("Return from execv() not expected");
        exit(EXIT_FAILURE);




}
```

Sample Run

.

.

.

```
-rwxr-xr-x    1 husaingholoom  staff    871 Nov 17 18:36 ztest7.c
-rwxr-xr-x    1 husaingholoom  staff    934 Nov 17 18:44 ztest8.c
-rwxr-xr-x    1 husaingholoom  staff   1814 Nov 19 18:58 ztest9.c
-rw-r--r--    1 husaingholoom  staff    321 Nov  5 17:31 zwait0.c
-rw-r--r--    1 husaingholoom  staff    966 Nov  5 17:50 zwait1.c
```

.

.

.

Example : Using execvp to Lunch Another Program

```
//EXEC.c

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

int main() {
	int i;

	printf("I am EXEC.c called by execvp() ");
	printf("\n");

	return 0;
}
```

//execDemo.c

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
int main() {
	//A null terminated array of character
	//pointers
	char *args[] = { "./EXEC", NULL };
	execvp(args[0], args);

	/* All statements are ignored after execvp() call as
	this whole process(execDemo.c) is replaced by
	another process (EXEC.c)     */

	printf("Ending-----");

	return 0;
}
```

Sample run

Create EXEC.c file , Save , chmod , and compile using

gcc EXEC.c -o EXEC

Create an executable file of execDemo.c

[hag10@zeus ~]$ gcc execDemo.c

Running the executable file of execDemo.c

[hag10@zeus ~]$ ./a.out

The following output is produced:

```
I AM EXEC.c called by execvp()
```

Example – 2 : using execv to Lunch Another Program

// example.c

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
int main(int argc, char *argv[]) {
        printf("PID of example.c = %d\n", getpid());
        char *args[] = { "Hello", "C", "Programming", NULL };
        execv("./hello", args);
        printf("Back to example.c");
        return 0;
}
```

// hello.c compile with gcc hello.c –o hello

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
int main(int argc, char *argv[]) {
        printf("We are in Hello.c\n");
        printf("PID of hello.c = %d\n", getpid());
        return 0;
}
```

Sample Run

PID of example.c = 2478

We are in Hello.c

PID of hello.c = 2478

Waiting for Children to Terminate

As stated above, parents need to get the termination status of their children (otherwise those **children** become **zombies**)

They can do so by calling **wait()**

One of the **main purposes** of **wait()** is to **wait** for completion of child processes.

wait() system call **suspends** execution of current process **until a child has exited** or until a **signal** has delivered whose action is to terminate the current process or call signal handler.

wait() takes the address of an integer variable and returns the process ID of the completed process.

The execution of **wait()** could have two possible situations.

1. If there are at least one child processes running when the call to **wait()** is made, the caller will be blocked until one of its child processes exits. At that moment, the caller resumes its execution.

2. If there is no child process running when the call to **wait()** is made, then this **wait()** has no effect at all. That is, it is as if no **wait()** is there.

#include <sys/types.h>
#include <sys/wait.h>

pid_t wait(int *status);

- **wait()** system call **suspends** execution of current process until a child has exited **or** until a signal has delivered whose action is to terminate the current process or call signal handler.

- **waitpid():** Suspends execution of current process until a child as **specified by pid arguments** has exited or until a signal is delivered.

pid_t waitpid (pid_t pid, int *status, int options);

Example - 1 : fork & without wait

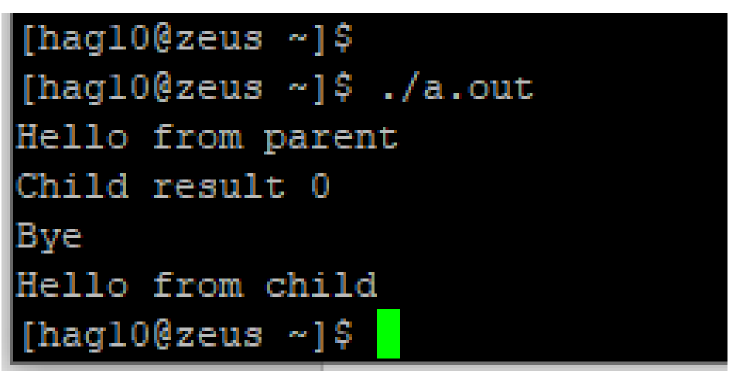
```
#include<stdlib.h>
#include<unistd.h>
#include<sys/types.h>
#include<sys/wait.h>
#include<stdio.h>


int main(){
   pid_t pid = fork();
   if (pid == 0) {
      printf("Hello from child\n");
      exit(17);
   } else {
      int child_status;
      printf("Hello from parent\n");
      printf("Child result %d\n", WEXITSTATUS(child_status));
   }
   printf("Bye\n");
   return 0;
}
```

Sample run

192:fork husaingholoom$ ./a.out

Hello from parent
Child result 0
Bye
Hello from child

192:fork husaingholoom$



Example - 2 : fork & wait

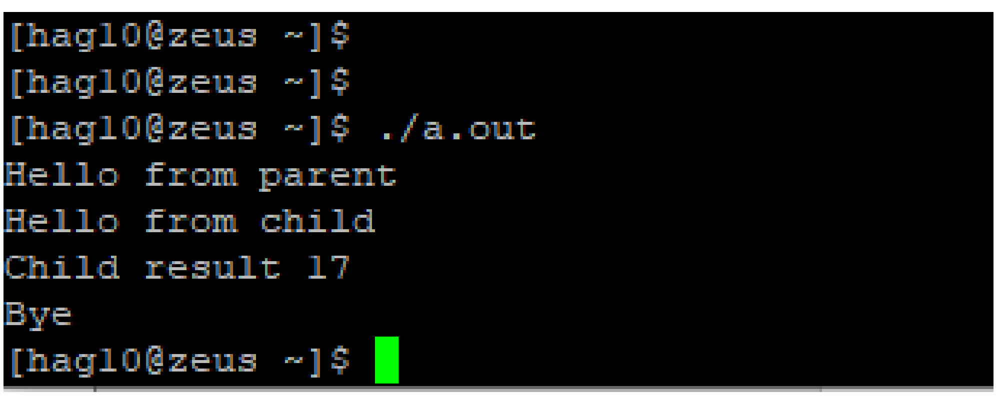
```
#include<stdlib.h>
#include<unistd.h>
#include<sys/types.h>
#include<sys/wait.h>
#include<stdio.h>


int main(){
   pid_t pid = fork();
   if (pid == 0) {
      printf("Hello from child\n");
      exit(17);
   } else {
      int child_status;
      printf("Hello from parent\n");
      waitpid(pid, &child_status, 0); // Waits for child to end
      printf("Child result %d\n", WEXITSTATUS(child_status));
   }
   printf("Bye\n");
   return 0;
}
```

Sample run

192:fork husaingholoom$ ./a.out

Hello from parent
Hello from child
Child result 17
Bye

192:fork husaingholoom$



Example - 3 : wait & fork

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

int main(int argc, char** argv) {
	pid_t pid;
	int sta;
	pid = fork();
	if (pid < 0) {
		printf("fail\n");
	} else if (pid == 0) {
		printf("child\n");
		return (10);
	} else if (pid > 0) {
		if (waitpid(pid, &sta, 0) == pid)
			printf("child status: %d\n", sta);

		printf("%d\n", WEXITSTATUS(sta));
		return (1);
	}
}
```

Sample Run

192:fork husaingholoom$ ./a.out
child
child status: 2560
10
192:fork husaingholoom$

Example - 4 : wait & fork

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

#include <sys/types.h>
#include <sys/wait.h>

int main(){

 pid_t   c_pid, pid;
 int status;

 c_pid = fork(); //duplicate

 if( c_pid == 0 ){
   //child
   pid = getpid();

   printf("Child: %d: I'm the child\n", pid, c_pid);
   printf("Child: sleeping for 2-seconds, then exiting with status
            12\n");

   //sleep for 2 seconds
   sleep(2);

  //exit with status 12
   exit(12);

 }else if (c_pid > 0){
```

```
    //parent

   //waiting for child to terminate
   pid = wait(&status);

   if ( WIFEXITED(status) ){
       printf("Parent: Child exited with status: %d\n",
       WEXITSTATUS(status));
   }

  }else{
   //error: The return of fork() is negative
   perror("fork failed");
   _exit(2); //exit failure, hard
  }

  return 0; //success
}
```

Sample Run

Child: 2334: I'm the child
Child: sleeping for 2-seconds, then exiting with status 12
Parent: Child exited with status: 12

```
[hag10@zeus ~]$
[hag10@zeus ~]$
[hag10@zeus ~]$ ./a.out
Child: 4788: I'm the child
Child: sleeping for 2-seconds, then exiting with status  12
Parent: Child exited with status: 12
[hag10@zeus ~]$
```

Example - 5 : wait and fork

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

void err_sys(char* message)
{
	perror(message);
	exit(EXIT_FAILURE);
}

int main(int argc, char* argv[])
{
	pid_t pid, ppid, child_pid;
	int i, status;

	pid = getpid();
	printf("Parent. My PID is %d and I am about to procreate\n", pid);
	fflush(stdout);

	for(i=0; i<3; i++)
	{
		child_pid = fork();
		if(child_pid<0)
		{
			err_sys("fork");
		}
		if(child_pid == 0)
		{
			break; /* Only the parent forks! */
		}
	}
```

```
    if(child_pid == 0)
    {
        pid = getpid();
        ppid = getppid();
        printf("Child. My PID is %d, my parent is %d\n", pid, ppid);
        sleep(1);
        exit(i);
   }

    printf("Parent: Waiting for my children\n");

    while((child_pid = wait(&status))!=-1)
    {
        printf("Child %d terminated with termination status %d\n",
        child_pid, status);
        if(WIFEXITED(status))
        {
            printf("Termination normal, exit status %d\n",
            WEXITSTATUS(status));
        }
    }

return EXIT_SUCCESS;

}
```

Example Output

192:fork husaingholoom$ ./a.out

Parent. My PID is 507 and I am about to procreate

Child. My PID is 508, my parent is 507

Child. My PID is 509, my parent is 507

Parent: Waiting for my children

Child. My PID is 510, my parent is 507

Child 509 terminated with termination status 256

Termination normal, exit status 1

Child 508 terminated with termination status 0

Termination normal, exit status 0

Child 510 terminated with termination status 512

Termination normal, exit status 2

192:fork husaingholoom$

```
[hagl0@zeus ~]$
[hagl0@zeus ~]$ ./a.out
Parent. My PID is 5574 and I am about to procreate
Child. My PID is 5575, my parent is 5574
Parent: Waiting for my children
Child. My PID is 5576, my parent is 5574
Child. My PID is 5577, my parent is 5574
Child 5575 terminated with termination status 0
Termination normal, exit status 0
Child 5576 terminated with termination status 256
Termination normal, exit status 1
Child 5577 terminated with termination status 512
Termination normal, exit status 2
[hagl0@zeus ~]$
```

Fork , exec and wait

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

#include <sys/types.h>
#include <sys/wait.h>

int main(int argc, char * argv[]) {
	//arguments for ls, will run: ls  -l
	char * ls_args[3] = { "ls", "-l", NULL };
	pid_t c_pid, pid;
	int status;

	c_pid = fork();

	if (c_pid == 0) {
		/* CHILD */

		printf("Child: executing ls\n");

		//execute ls
		execvp(ls_args[0], ls_args);
		//only get here if exec failed
		perror("execve failed");
	} else if (c_pid > 0) {
		/* PARENT */

		if ((pid = wait(&status)) < 0) {
			perror("wait");
			_exit(1);
		}

		printf("Parent: finished\n");

	} else {
		perror("fork failed");
		_exit(1);     }

	return 0; //return success       }
```

Sample run

husain-gholooms-macbook:~ husaingholoom$ ./a.out
Child: executing ls
total 2912
-rw-r--r--   1 husaingholoom staff   56185 Dec 18 2009 5211_108701388007_674443007_2339434_6002540_n[1].jpg
drwxr-xr-x   4 husaingholoom staff   136 Mar 30 2014 Alice3
-rwxrwxrwx@   1 husaingholoom staff 150528 Feb 16 2012 Automated Debugging Methodologies.doc
drwx------+   56 husaingholoom staff   1904 Apr 6 21:33 Desktop
drwx------+ 282 husaingholoom staff   9588 Apr 5 22:13 Documents
drwx------+ 3723 husaingholoom staff 126582 Apr 9 09:46 Downloads
.
.
.
-rwxr-xr-x   1 husaingholoom staff   121 Oct 15 17:45 zyy.txt
-rwxr-xr-x   1 husaingholoom staff   436 Oct 15 17:50 zz.c
-rwxr-xr-x   1 husaingholoom staff   109 Oct 15 17:54 zzz.txt
Parent: finished
husain-gholooms-macbook:~ husaingholoom$

Sample Run 2 – Rplace the ls_arg to be

```
char * ls_args[3] = { "cat", "foo.txt", NULL };
```

```
[hag10@zeus ~]$
[hag10@zeus ~]$ ./a.out
Child: executing ls
1234 txt
Parent: finished
[hag10@zeus ~]$
```

What is the output of the following

```
#include <unistd.h>

#include <stdio.h>

#include <stdlib.h>

#include <sys/types.h>

#include <sys/wait.h>

int main(void) {
	char *argv[3] = { "Command-line", ".", NULL };

	int pid = fork();

	if (pid == 0) {
		execvp("find", argv);
	}

	sleep(2);

	printf("Finished executing the parent process\n");

	return 0;
}
```

Process Groups

UNIX processes are organized in **process groups**

– A process group has a **group leader**
– All processes in the group have the same **process group id** (which is the **process id** of the group leader)

Some operations can be done not just for single processes.

For a whole group:
– Delivering signals with kill
– Waiting for process termination with waitpid() (later)

By default, a process inherits the process group id from its parent

– Processes can change their own process group id

_ . . . to become process group leaders in a new process group, or

_ . . . to join an existing process group

– **Parents can change the process group id of their children** (unless the children already called exec())

Note: Don't confuse the pgid (process group) with the gid (user/owner group)

Getting and Changing Process Groups

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

pid_t   getpgrp(void);

int   setpgid(pid_t  pid, pid_t  pgrp);
```

getpgrp() always returns the **process group id** of the current process
– No error condition!

setpgid(pid_t pid, pid_t pgrp) sets the process group id of the process with the PID pid to **pgrp**

- Return value: 0 on success, -1 on error (errno set)
- Special values:
  - If pid is 0, the PID of the calling process is assumed
  - If pgrp is 0, the process id denoted by the first argument is assumed (i.e. that process is made into a process group **leader** of a new process group)

– Note that this means that setpgid(0,0) makes the current process into a process **group leader**

Example

```
#include <stdlib.h>
#include <stdio.h>
#include <sys/wait.h>
#include <unistd.h>
void err_sys(char* message) {
	perror(message);
	exit(EXIT_FAILURE);
}

int main(int argc, char* argv[]) {
	pid_t pid, pgid, child_pid;
	int i, res;
	pid  = getpid();
	pgid = getpgrp();
	printf("Parent. My PID is %d and my process group is %d\n",   pid, pgid);
	res = setpgid(0, 0);
	if (res == -1) {
		err_sys("setpgid");
	}
	printf("Parent. I’m now the process group leader.\n");
	for (i = 0; i < 3; i++) {
		child_pid = fork();
		if (child_pid < 0) {
			err_sys("fork");
		}
		if (child_pid == 0) {
			break; /* Only the parent forks! */
		}
	}
```

```
    if (child_pid == 0) {
        pid  = getpid();
        pgid = getpgrp();
        printf("Child %d. My PID is %d, my process group is
                %d.\n", i, pid,  pgid);
        sleep(1);
        res = setpgid(0, 0);
        if (res == -1) {
            err_sys("setpgid");
        }
        pid  = getpid();
        pgid = getpgrp();
        printf("Child %d. I'm now independent, pid %d and pgid
                %d\n", i, pid, pgid);
        printf("Child %d exiting\n", i);
        exit(EXIT_SUCCESS);
    }
    printf("Parent, sleeping.\n");
    sleep(3);
    printf("Parent, exiting.\n");
    return EXIT_SUCCESS;
}
```

Sample Run

$ ./pg example

Parent. My PID is 1946 and my process group is 1946
Parent. I'm now the process group leader.
Parent, sleeping.
Child 0. My PID is 1947, my process group is 1946.
Child 1. My PID is 1948, my process group is 1946.
Child 2. My PID is 1949, my process group is 1946.
Child 0. I'm now independent, pid 1947 and pgid 1947
Child 0 exiting
Child 1. I'm now independent, pid 1948 and pgid 1948
Child 1 exiting
Child 2. I'm now independent, pid 1949 and pgid 1949
Child 2 exiting
Parent, exiting.

- Note that the parent **starts out** as a process group leader!
- Most shells with build-in job control will always execute commands in their own process group

Sample Run

```
[hag10@zeus ~]$
[hag10@zeus ~]$ ./a.out
Parent. My PID is 8546 and my process group is 8546
Parent. I'm now the process group leader.
Child 0. My PID is 8547, my process group is  8546.
Parent, sleeping.
Child 1. My PID is 8548, my process group is  8546.
Child 2. My PID is 8549, my process group is  8546.
Child 0. I'm now independent, pid 8547 and pgid   8547
Child 0 exiting
Child 1. I'm now independent, pid 8548 and pgid   8548
Child 1 exiting
Child 2. I'm now independent, pid 8549 and pgid   8549
Child 2 exiting
Parent, exiting.
[hag10@zeus ~]$
```

Redirection

In Unix, you can redirect standard output to go to a file. This output redirection is very useful if you want to save the output of a command to a file rather than just letting it flash across the screen and eventually scroll away from view.

For example, you can redirect the output of the ls –l *.c command to a file called foo2.txt.

```
ls –ls *.c > foo2.txt
```

You can also create a process that performs such task by using system()

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

int main()
{
   int pid;
   pid = fork();
   if ( pid == 0 )
      system("ls -l *.c >  foo2.txt");

   return 0;
}
```

Sample Run

```
husain-gholooms-macbook:~ husaingholoom$ ./a.out
husain-gholooms-macbook:~ husaingholoom$ cat foo2.txt
-rwxr-xr-x  1 husaingholoom  staff   276 Apr  7  2020 example.c
-rwx------  1 husaingholoom  staff   274 Mar 31  2019 file.c
-rwxr-xr-x  1 husaingholoom  staff   192 Apr  7  2020 hello.c
-rwx------  1 husaingholoom  staff    25 Oct  1  2013 myfile.c
-rwxr-xr-x  1 husaingholoom  staff   147 Mar 29  2020 sys.c
.
.
.

husain-gholooms-macbook:~ husaingholoom$
```

Redirection via dup (C System Call)

The system call dup(int fd) **duplicates a file descriptor** fd.

What this does is return a second file descriptor that points to the same file table entry as fd does.

So now you can treat the two file descriptors as identical.

Example

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

main()
{
  int fd1, fd2;

  fd1 = open("file2", O_WRONLY | O_CREAT | O_TRUNC, 0644);
  fd2 = dup(fd1);

 write(fd1, "Unix_System_Programming\n",
           strlen("Unix_System_Programming\n"));

  write(fd2, "CS_Department_TX-State\n",
           strlen("CS_Department_TX_State\n"));

   write(fd2, "Fall 2024\n",
            strlen("Fall 2024\n"));

    write(fd1, "November 2024\n",
             strlen("November  2024\n"));

  close(fd1);
  close(fd2);
}
```

Sample Run

husain-gholooms-macbook:~ husaingholoom$ ./a.out

husain-gholooms-macbook:~ husaingholoom$ cat file2

Unix_System_Programming

CS_Department_TX-State

Fall 2024

November 2024

husain-gholooms-macbook:~ husaingholoom$

Sample Run

```
[hag10@zeus F2024]$
[hag10@zeus F2024]$ cat file2
Unix_System_Programming
CS_Department_TX-State
Fall 2024
November 2024
[hag10@zeus F2024]$
[hag10@zeus F2024]$
```

Redirection via dup2 (C System Call)

dup2 is a **system call** similar to **dup** in that it **duplicates one file descriptor.**

Dup2() is most often used so that you can **redirect standard input or output.**

When you call **dup2(fd, 0)** and the dup2() call is successful, then whenever your program **reads** from standard input, it will read from fd.

Similarly, when you call **dup2(fd, 1)** and the dup2() call is successful, then whenever your program **writes** to standard output, it will write to fd.

For example, if you wanted to redirect standard output to a file, then you would simply call **dup2**, providing **the open file descriptor** for the file as the **first** command **and 1** (standard output) as the **second command.**

Example

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

int main() {
        int fd;
        char *s;

        fd = open("file4", O_WRONLY | O_CREAT | O_TRUNC, 0666);

        if (dup2(fd, 1) < 0) {
                perror("dup2");
                exit(1);
        }

        printf("Standard output now goes to file4\n");

        close(fd);

        printf("It goes even after we closed file descriptor %d\n",
                    fd);
        putchar('p');
        putchar('u');
        putchar('t');
        putchar('c');
        putchar('h');
        putchar('a');
        putchar('r');
        putchar(' ');
   putchar('w');
        putchar('o');
        putchar('r');
        putchar('k');
        putchar('s');
        putchar('\n');

        s = "And fwrite\n";

        fwrite(s, sizeof(char), strlen(s), stdout);

        fflush(stdout);
```

```
        s = "And write\n";
        write(1, s, strlen(s));       }
```

Sample Run

husain-gholooms-macbook:~ husaingholoom$ ./a.out
husain-gholooms-macbook:~ husaingholoom$

husain-gholooms-macbook:~ husaingholoom$ cat file4

Standard output now goes to file4
It goes even after we closed file descriptor 3
putchar works
And fwrite
And write
husain-gholooms-macbook:~ husaingholoom$

Sample Run

```
[hag10@zeus ~]$
[hag10@zeus ~]$
[hag10@zeus ~]$ cat file4
Standard output now goes to file4
It goes even after we closed file descriptor 3
putchar works
And fwrite
And write
[hag10@zeus ~]$
```

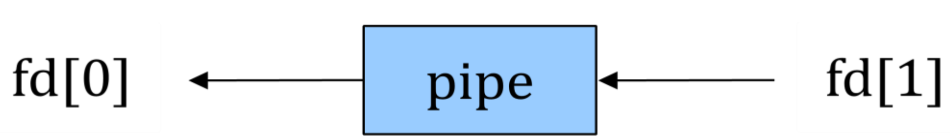
Pipe

Pipe is used to **combine two or more commands.**

It is a **form of redirection** (transfer of standard output to some other destination) that is used in Linux and other Unix-like operating systems **to send the output of one** command/program/process **to another** command/program/process for further processing.

This direct connection between commands/ programs/ processes allows them to **operate** simultaneously and permits data to be transferred between them continuously **rather than** having to pass it through temporary text files or through the display screen.

Pipes are unidirectional i.e **data flows from left to right through the pipeline.**



Example

```
husain-gholooms-macbook:~ husaingholoom$ ls | more

Alice3
Automated Debugging Methodologies.doc
Desktop
Documents
Downloads
HelloWord
Library
NUL
Retrieved Contents
SWT-Research-Interst.doc
Sites
Temp
Travel
USA Expenses.xlsx
Znew.txt
Zold.c
a.out
file4
:
```

To create a simple pipe with C, we make use of the **pipe() system call.**

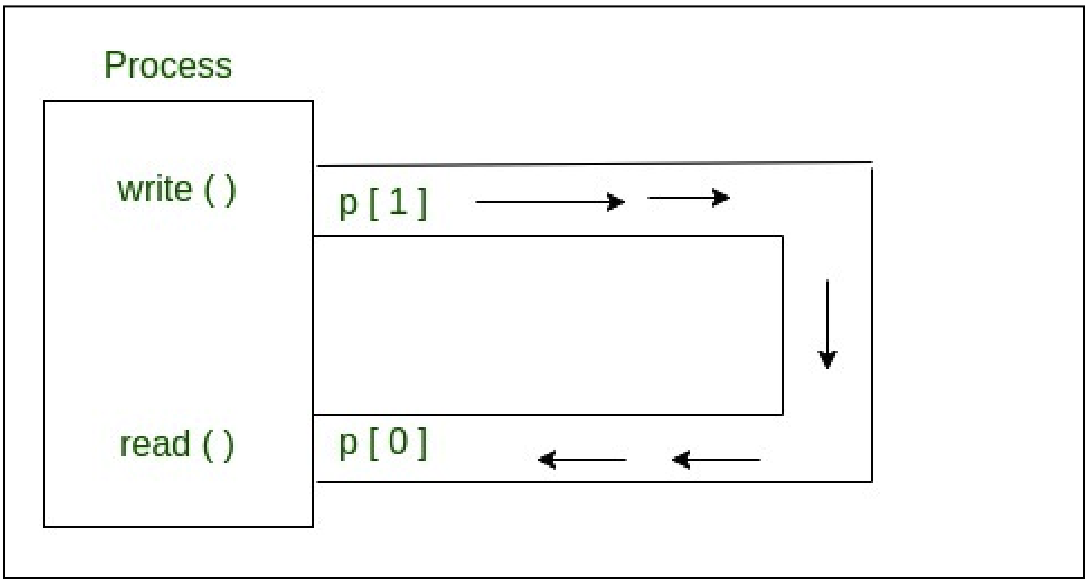
It takes a single argument, **which is an array of two integers**, and if successful, the array will contain two new file descriptors to be used for the pipeline.

After creating a pipe, the process typically spawns a new process (remember the child inherits open file descriptors). It opens a pipe, which is an area of main memory that is treated as a ***"virtual file"*.**

One process can **write to this "virtual file"** or pipe and **another** related process can **read** from it.

If a process **tries to read before something is written** to the pipe, the process is suspended until something is written.

The pipe system call finds the first two available positions in the process's open file table and allocates them for the read and write ends of the pipe.



#include<unistd.h>

int pipe (int fd[2])

Parameters :

fd[0] will be the fd(file descriptor) for the **read** end of pipe.
fd[1] will be the fd for the **write** end of pipe.

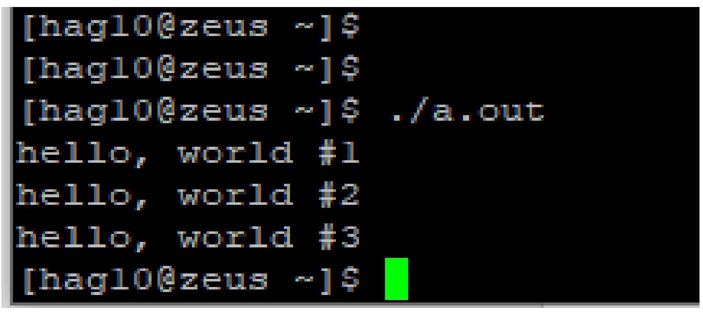
Returns : 0 on Success. -1 on error.

Example

```
#include <stdio.h>
#include <unistd.h>
#define MSGSIZE 16
char* msg1 = "hello, world #1";
char* msg2 = "hello, world #2";
char* msg3 = "hello, world #3";

int main()
{
   char inbuf[MSGSIZE];
   int p[2], i;

   if (pipe(p) < 0)
      exit(1);

   /* continued */
   /* write pipe */

   write(p[1], msg1, MSGSIZE);
   write(p[1], msg2, MSGSIZE);
   write(p[1], msg3, MSGSIZE);

   for (i = 0; i < 3; i++) {
      /* read pipe */
      read(p[0], inbuf, MSGSIZE);
      printf("% s\n", inbuf);
   }
   return 0;   }
```

Sample run

husain-gholooms-macbook:~ husaingholoom$ ./a.out
hello, world #1
hello, world #2
hello, world #3
husain-gholooms-macbook:~ husaingholoom$



Note

Pipes behave **FIFO**(First in First out), Pipe behave like a **queue** data structure. Size of read and write don't have to match here.

Size of read and write don't have to match.

In a pipe , 512 bytes can be **written** at a time , but **only 1 byte** at a time can be **read** .

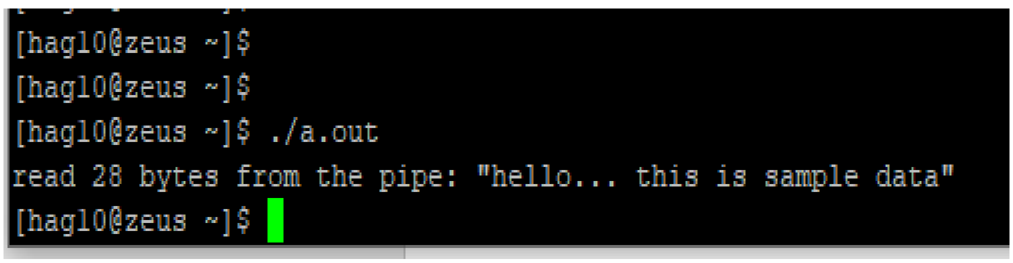
Example

```
#include <stdlib.h>
#include <stdio.h> /* for printf */
#include <string.h> /* for strlen */
int main(int argc, char **argv)
{
        int n;
        int fd[2];
        char buf[1025];
        char *data = "hello... this is sample data";
        pipe(fd);
        write(fd[1], data, strlen(data));
        if ((n = read(fd[0], buf, 1024)) >= 0)
        {
            buf[n] = 0; /* terminate the string */
            printf("read %d bytes from the pipe: \"%s\"\n", n, buf);
        }
        else
            perror("read");

    exit(0);
}
```

Sample Run

192:c_programs husaingholoom$ ./a.out
read 28 bytes from the pipe: "hello... this is sample data"
192:c_programs husaingholoom$

Sample Run



A regular pipe can only connect two related processes. It is created by a process and will vanish when the last process closes it.

Fork () and pipe .

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

int main(void)
{
    int   fd[2], nbytes;
    pid_t  childpid;
    char   string[] = "Hello, world!\n";
    char   readbuffer[80];

    pipe(fd);

    if((childpid = fork()) == -1)
    {
        perror("fork");
        return (1);
    }


  if(childpid == 0)
    {
        /* Child process closes up input side of pipe */
        close(fd[0]);

        /* Send "string" through the output side of pipe */
        write(fd[1], string, (strlen(string)+1));
        return (0);
    }
```

```
    else
    {
        /* Parent process closes up output side of pipe */
        close(fd[1]);

        /* Read in a string from the pipe */
        nbytes = read(fd[0], readbuffer, sizeof(readbuffer));
        printf("Received string: %s", readbuffer);
    }

    return(0);
}
```

Sample Run -1

husain-gholooms-macbook:~ husaingholoom$ ./a.out
Received string: Hello, world!
husain-gholooms-macbook:~ husaingholoom$

Sample Run -2

```
[hag10@zeus ~]$
[hag10@zeus ~]$ ./a.out
Received string: Hello, world!
[hag10@zeus ~]$
[hag10@zeus ~]$
[hag10@zeus ~]$
```

Signals

Signals are a way to signal unusual events to a process

– Run time errors
– User requests
– Pending communication

In general, signals can arrive asynchronously, i.e. at any time

Signals can have many different values.

Depending on the value, the process can :

- – Ignore a signal
- – Perform **a default action** (defined by the implementation)
- – Invoke an explicit signal handler

Each signal has a name and an integer value associated with them

- When a signal is emitted, the corresponding integer value is sent to the process
- Message encoded in integer value

All signal names start with SIG

- SIGKILL
- SIGABORT

Associated integer values may vary across systems

Should refer to signals by their name rather than their numeric value

Standard C Signals

Standard C defines a small number of signals, UNIX defines many more. They begin with SIG

Signal	Meaning	Default Action (UNIX)
SIGABRT	Abort the process	Terminate
SIGFPE	Floating point exception	Terminate with core
SIGILL	Illegal instruction	Terminate with core
SIGINT	Interactive interrupt	Terminate
SIGSEGV	Illegal memory access	Terminate with core
SIGTERM	Termination request	Terminate

Note: **SIGINT** is generated when you press [CTRL-C]!

– The signal is delivered to the process
– The default action is to terminate the process

Signal Generation

From the terminal

- typing ^C at the terminal sends the SIGINT to the process running in the foreground

H/W exceptions

- invalid memory reference results in SIGSEGV

kill() function

- send named signals to other processes (restrictions apply)

kill command

- send named signals to other processes from the shell (restrictions apply)

Sending Signals From The Terminal

^C : SIGINT

^\ : SIGQUIT

^Z : SIGTSTP

Signal only sent to foreground process :

- **Mostly used for terminating the current process**

Some UNIX Signals

UNIX defines about 60 different signals, including all Standard C signals

Some important UNIX signals:

Signal	Meaning	Default Action (UNIX)
SIGHUP	Terminal connection lost (or controlling process dies)	Terminate
SIGKILL	Kill process, cannot be caught or ignored	Terminate
SIGBUS	Bus error	Terminate with core
SIGSTOP	Stop a process	Suspends process
	(does not terminate, cannot be caught or ignored)	
SIGCONT	Continue suspended	process Ignored (*)
SIGURG	Out of band data arrived on a socket	Ignore
SIGXCPU	CPU time limit reached	Terminate with core

(*) OS will still wake process up

[CTRL-Z] generates SIGSTOP

UNIX User Command: kill

Note: kill is often implemented as a shell built-in

- – Syntax may differ slightly from the kill program
- – Allows use of kill in job control

Usage for our kill: kill [-<SIG>] <pad> ...

If no signal is specified, SIGTERM is sent

- Signals can be specified symbolically (for a list of names run kill -l)
- or numerically (**man 7 signal** gives a list of signals and their numeric values)

kill accepts a list of <pad> arguments

– Most common case: **<pad>** is a normal **process id** (a positive integer). The signal is sent to the corresponding process

– If <pad> is **-1**, **the signal is sent to all processes of the user** (kill –KILL -1 is a surefire way to log yourself out)

– Finally, if <pad> **is any other negative number**, the signal is sent to the **corresponding process group**

For example, to send a “hang-up” signal to a shell running on a different terminal with PID 512, you would use the command

$ kill –HUP 512

The kill() Function

Header

#include<signal.h>

Prototype

kill (pid_t pid, int sig);

- Sends signal **sig** to process **pid**
- Fails if
    - Invalid signal
    - Insufficient permission
    - Process does not exist

- **Returns -1 on failure**

Note: kill() is the function used to implement the kill command

Examples

Terminate the processes with pids 1412 and 1157:

```
$ kill 1412 1157
```

Send the hangup signal (SIGHUP) to the process with pid 5071:

```
$  kill -s HUP 5071
```

Terminate the process group with pgid 12117:

```
$ kill -- -12117
```

To send -9 (KILL) singal to the processo with pid 1234, enter:

```
$ kill -9 1234
```

Example

```
#include <stdlib.h>
#include <stdio.h>
#include <sys/wait.h>
#include <signal.h>
#include <unistd.h>
void err_sys(char* message)
{
perror(message);
exit(EXIT_FAILURE);
}

int main(int argc, char* argv[])
{
        pid_t pid, pgid, child_pid;
        int i, res;
        res = setpgid(0,0);
        if(res==-1)
        {
                err_sys("setpgid");
        }
        pid = getpid();
        pgid = getpgrp();
        printf("Queen bee:PID is %d process group is %d\n",pid,pgid);
```

```
        for(i=0; i<3; i++)
        {
                child_pid = fork();
                if(child_pid<0)
                {
                        err_sys("fork");
                }
                if(child_pid == 0)
                {
                        break; /* Only the parent forks! */
                }
        }

        if(child_pid == 0)
        {
                while(1)
                {
                        printf("Worker bee %d gathering honey\n", i);
                        sleep(1);
                }
        }
        for(i=0; i<3; i++)
        {
                printf("Queen bee sleeping\n");
                sleep(1);
        }
        printf("Queen bee terminates\n");
        kill(-getpgrp(), SIGTERM); /* Commented out for version 2 */

return EXIT_SUCCESS;
}
```

Example Output with kill

```
192:c_programs husaingholoom$ ./a.out
Queen bee:PID is 1078 process group is 1078
Worker bee 0 gathering honey
Worker bee 1 gathering honey
Queen bee sleeping
Worker bee 2 gathering honey
Worker bee 1 gathering honey
Queen bee sleeping
Worker bee 0 gathering honey
Worker bee 2 gathering honey
Queen bee sleeping
Worker bee 0 gathering honey
Worker bee 1 gathering honey
Worker bee 2 gathering honey
Queen bee terminates
Worker bee 0 gathering honey
Terminated: 15
192:c_programs husaingholoom$
```

Example Output without kill

./pgkill_example

Queen bee:PID is 2460 process group is 2460
Queen bee sleeping
Worker bee 0 gathering honey
Worker bee 1 gathering honey
Worker bee 2 gathering honey
Queen bee sleeping
Worker bee 0 gathering honey
Worker bee 1 gathering honey
Worker bee 2 gathering honey
Queen bee sleeping
Worker bee 0 gathering honey
Worker bee 1 gathering honey
Worker bee 2 gathering honey
Queen bee terminates

```
192:c_programs husaingholoom$ Worker bee 0 gathering honey

Worker bee 1 gathering honey
Worker bee 2 gathering honey
Worker bee 0 gathering honey
Worker bee 1 gathering honey
Worker bee 2 gathering honey
Worker bee 0 gathering honey
Worker bee 1 gathering honey
Worker bee 2 gathering honey
Worker bee 1 gathering honey
Worker bee 2 gathering honey
Worker bee 0 gathering honey
Worker bee 1 gathering honey
Worker bee 2 gathering honey
Worker bee 0 gathering honey

.
.
.
```

Signal Handling

- Since signals can be generated automatically in response to some condition, processes should provide some mechanism for handling signals

- Often not necessary for trivial programs
  - There is usually a default handler

Three things we can tell the kernel to do when a signal is received

- **Ignore** the signal
  - SIGKILL and a few others cannot be ignored

- **Catch** the signal
  - Ask the kernel to execute a user-defined function whenever a particular signal is raised

- Apply the **default** action
  - There is a default action associated with every signal
  - Default action usually terminates the program

Catching Signals

User programs can set up a signal handler to catch signals

– A signal handler is a normal function

– It has to be explicitly set up for each signal type

– It will be called asynchronously when a signal of the correct type has been caught

– When the signal handler returns, the program will resume execution at the old spot

UNIX implements several different ways of handling signals, we will concentrate on the **ANSI C** signal handling

All use the same signal: Signals are small integers

Signal handling stuff is defined in **<signal.h>**

ANSI C Signal Handling with signal.h

signal.h defines the **signal()** function for establishing signal handlers as follows :

void (*signal(int sig, void (*handler)(int)))(int)

Predefined (pseudo) signal handlers - possible arguments to **signal():**

– **SIG_DFL**: Revert to the default behavior for that signal
– **SIG_IGN**: Ignore the signal from now on

int raise(int sig) raises a signal to the program

– Return value: 0 on success, something else otherwise

There are several limitations on signal handler:

- Since signals can arrive asynchronously, the state of the program is not well-defined!
- Signals may be handled even within a single C statement
- Once a signal has been **caught**, the signal handler for that signal is reset to default behavior

Example

```
#include<stdio.h>
#include<unistd.h>
#include<signal.h>  // need this

int main() {

  int i;

  /* declare struct sigaction */
  struct sigaction newaction;

  /* SIG_IGN for ignore  */
  newaction.sa_handler = SIG_IGN;

  /* associate SIGINT with ignore action */
  sigaction(SIGINT, &newaction, 0);

  /* associate default action with SIGQUIT */
  sigaction(SIGQUIT, &newaction, 0);

  while(1) {
    printf("iteration count %d\n", i++);
    /* switch to default action for SIGINT when i > 10 */
    if (i > 10) {
      newaction.sa_handler = SIG_DFL;
      sigaction(SIGINT, &newaction, 0);
    }
    sleep(1);    }

  return 0;
}
```

Sample Run

```
./a.out
iteration count 0
iteration count 1
iteration count 2
iteration count 3
^Citeration count 4                // ignored
iteration count 5
iteration count 6
iteration count 7
^Citeration count 8                // ignored
iteration count 9
iteration count 10
iteration count 11
iteration count 12
iteration count 13
iteration count 14
^C
192:C_Programs husaingholoom$
```

UNIX User Commands: top

top is an interactive version of **ps**

- It shows various information about the **top** processed currently running
- Also shows general system information
- All information is periodically updates
- top seems to be more consistent between different UNIX dialects, and is often preferred for interactive use (or even for scripting)

Snapshot

.
.
Tasks: 7 total, 1 running, 6 sleeping, 0 stopped, 0 **zombie**
Mem: 3922928k total, 516360k used, 3406568k free, 105876k buffers
Swap: 4194296k total, 84872k used, 4109424k free, 205080k cached

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
3801	ns1254	20	0	105m	688	536	S	0.0	0.0	0:01.20	training.sh
5845	ns1254	20	0	105m	688	536	S	0.0	0.0	0:01.24	training.sh
23965	hag10	20	0	113m	1988	1032	S	0.0	0.1	0:00.08	sshd
23966	hag10	20	0	118m	4352	1588	S	0.0	0.1	0:00.11	bash
23997	hag10	20	0	25784	1284	1052	R	0.0	0.0	0:00.36	top
24021	ns1254	20	0	191m	13m	3556	S	0.0	0.4	0:00.44	python
24022	ns1254	20	0	98.5m	600	516	S	0.0	0.0	0:00.00	sleep

Zombie vs Orphan

- **Orphan**: parent finishes first
  - Taken over by **init** process, (1)
- **Zombie**: child finishes first, but parent didn't call wait()
  - Released after the parent calls wait().
  - Released if the parent terminates.

Common UNIX functions: sleep()

Often, a program only has to perform task only occasionally, or it has to **wait** for a certain event to happen.

ANSI C has no way of delaying a program

– Old-style home computer programmers use **busy delay loop**
– However, those are unacceptable on multi-user systems
– Moreover, they can usually be optimized away by a good compiler

All UNIX versions address this problem with the **sleep()** function (normally defined in **<unistd.h>):**

unsigned int sleep(unsigned int seconds);

sleep() makes the current process sleep (do nothing ;-) until either

- (At least) seconds have elapsed or
- A non-ignored signal arrives

Return value:

- 0 if sleep terminated because of elapsed time
- Number of seconds left when the process was awakened by a signal

pause()

- **Suspend** calling process until a signal is received
- Similar to using **sleep()**
- Only difference is, **not all signals are guaranteed to wake** up a sleeping process

    - Solaris : SIGCHILD doesn't wake up parent

Example :

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

void handler(int sig) {
  printf("received SIGINT, ignoring\n");
  fflush(stdout);
}

int main() {

        int i = 1;

        struct  sigaction   newaction;
        newaction.sa_handler = handler;
        sigaction(SIGINT, &newaction, 0);

        printf("started execution\n");
        printf("going to sleep ...\n");

        pause();

        printf("woke up!\n");
       return 0;
}
```

Sample Run

started execution
going to sleep ...
^Creceived SIGINT, ignoring
woke up!
192:C_Programs husaingholoom$