COMP3230A Tutorial 2: Process

Mr. HOU Weiying
Email: wyhou@cs.hku.hk

Content:

- Session 1: Programming Assignment 1: JCShell
- Session 2: How to implement a shell?
  - Process Creation: fork()
  - Process Synchronization: wait()
  - Process exec() family
  - Inter-Process Communication(IPC)
    - Pipe
    - Signal

Programming Assignment 1: Briefing

- Accept input [demo]:
  - Execute command under
    - Absolute path
    - Relative path
    - PATH, echo $PATH
    - path error
  - Using pipe:
    - Piped commands
  - Running Statistics
  - SigINT
- Submission:

What is a Shell?

- *Shell*: Shell program is a program that acts as the user interface to the Operating System and allows the system to understand your commands.

- Well-known shells on Linux
  - Ash, Bash, csh, ksh, ..., zsh, ..., ..., JCshell

(default on linux)

(default on macOS)

JCshell can do:

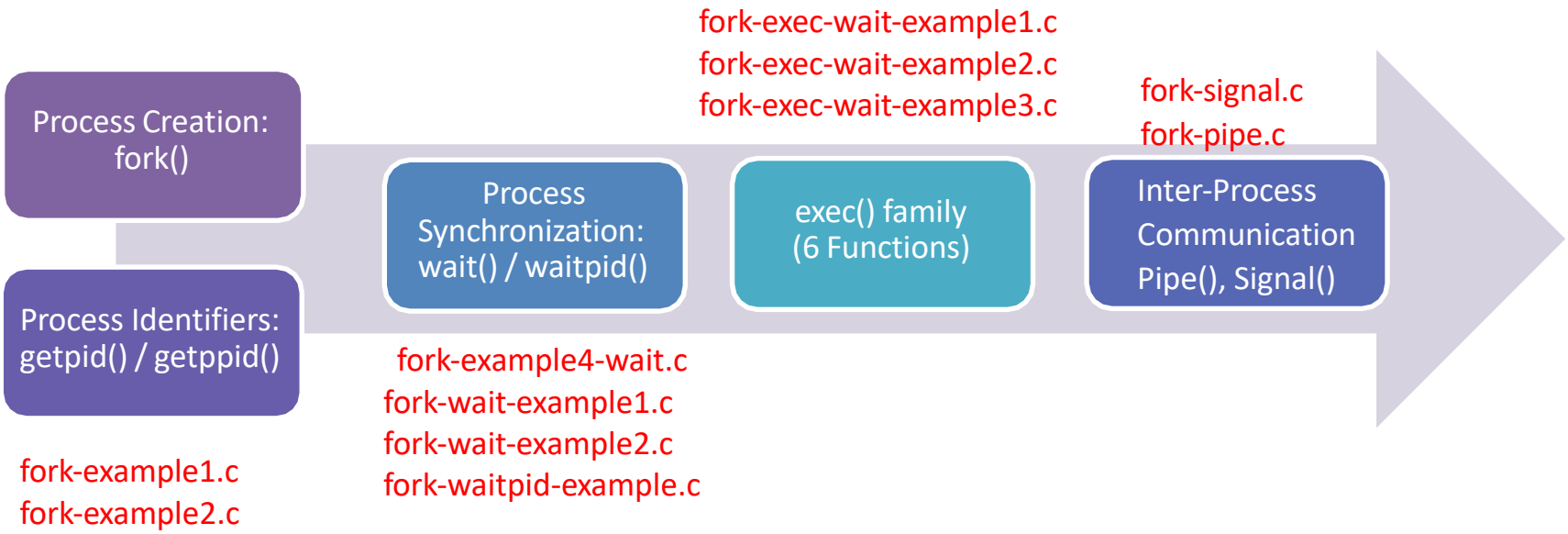
1. command execution: shell can parse and execute commands under $PATH, using absolute/relative path to command executables.
2. Use pipes (`|`) to connect the output of one command to the input of another command.
3. Print out process running statistics from /proc
4. Handle signals, e.g. SIGINT, SIGUSR1

Session 2: How to implement a shell?

… and get full mark in PA1?

1. command parse
2. Command execution: fork() wait() exec()
3. Create pipes for Inter-process communication
4. Read running statistics
5. Handle signals
6. …

Session 2: How to implement a shell



- Download/`git pull` from Github.
- Files under ./Tutorial2-Process

Process Creation – fork()

- An existing process can ***create a new process*** by calling ***fork()***.
  - Include Header File: <unistd.h> ←unix standard

```
pid_t fork(void);
```

  - The newly-created process is called ***the child process***.
  - This function is called once but returns twice.
  - How can we manage to know there are 2 processes running concurrently?

Return pid of the new child

Return Value	Description
< 0	Returns a value which is smaller than 0 to indicate the failure of creating a process.
0	Returns 0 in the ***child***.
> 0	Returns the process identifier (pid) of the child process in the ***parent***.

Process Creation – fork()

- Every process has ***a UNIQUE process ID***, ***a NON-NEGATIVE integer***.
  - With the process ID, you are able to know whether the current process is a ***parent*** process ***or*** a ***child*** process.

Useful Functions	Description
pid_t getpid(void);	Returns the process ID of calling process
pid_t getppid(void);	Returns the ***PARENT*** process ID of calling process

  - Process ID 1 is usually the `init` process and is invoked by the kernel at the end of the bootstrap procedure. It is responsible for starting or shutting down the system.

Process Creation – fork() (fork-example1.c)

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

int main(int argc, char *argv[]) {
        pid_t pid = fork();

        if (pid < 0) {
                // Error Occurred
                fprintf(stderr, "fork() Failed");
                exit(-1);
        } else if (pid == 0) {
                // Child Process
                printf("Child Process.\n");
                printf("The process id of Child Process is: %d\n", getpid());
                printf("The parent process id of Child Process is: %d\n", getppid());
                printf("\n");
        } else {
                // Parent Process
                printf("Parent Process.\n");
                printf("The process id of Parent Process is: %d\n", getpid());
                printf("The parent process id of Parent Process is: %d\n", getppid());
                printf("\n");
        }

        return 0;
} // main
```

Child pid

Child ppid

Parent pid

Parent ppid

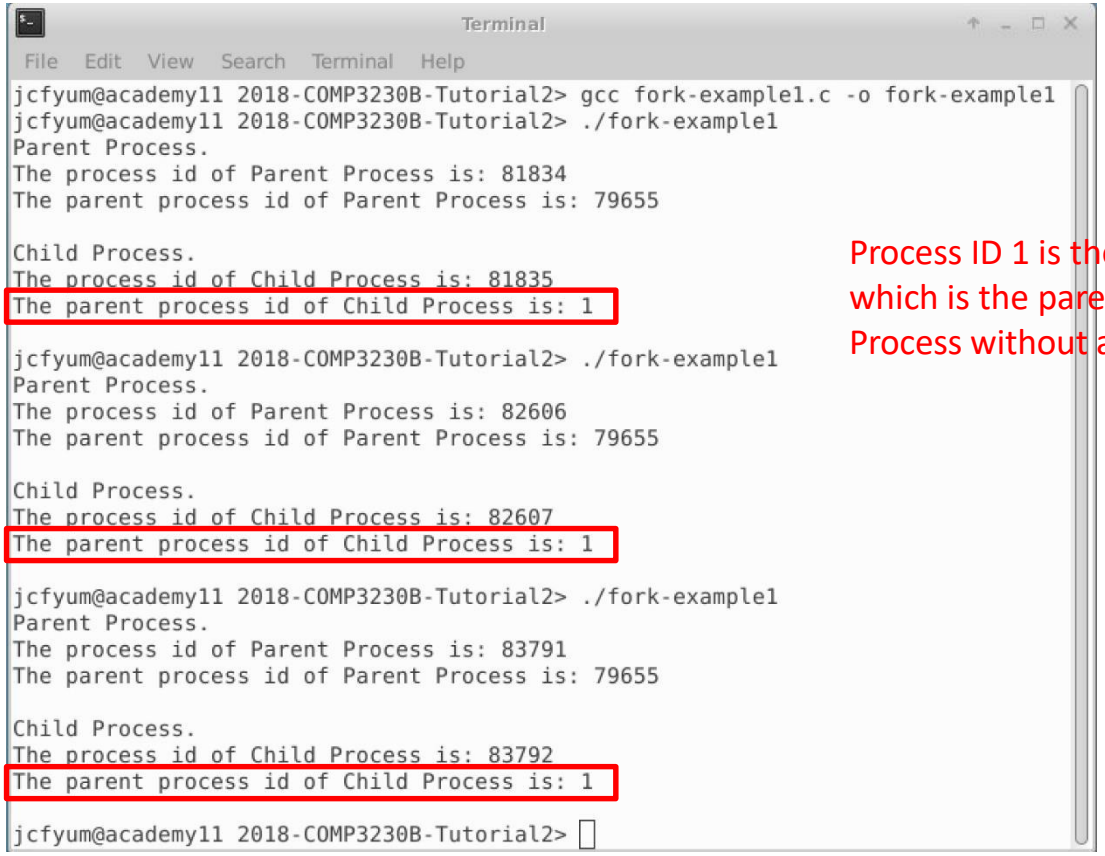
What's the output?
Relation of

Child pid
Child ppid
Parent pid
Parent ppid

<shell pid>
<init pid>

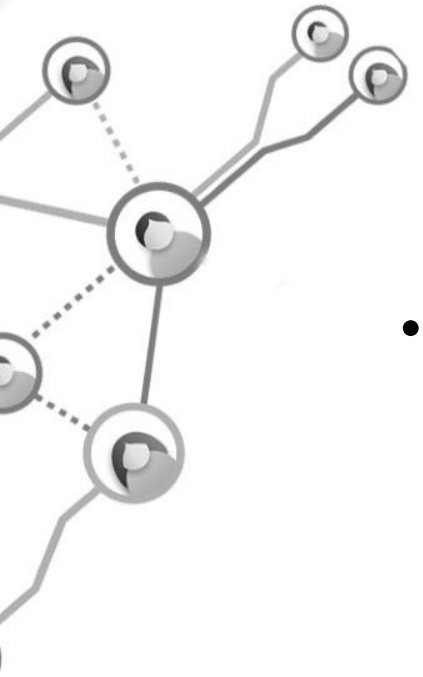
Process Creation – fork() (fork-example1.c)

- Sample Output:



Process ID 1 is the `init` process
which is the parent process of any orphaned child process.
Process without a parent(terminated).

Process Creation – fork() (fork-example1.c)



- Question: Can we get ***the SAME output*** when we execute the same program for many times?
  - **NO!!!**
    - ✓ In general, we never know whether the child starts executing before the parent, or vice versa.
    - ✓ The order depends on the scheduling algorithm used by the kernel.
- If we want to "schedule" the processes such that the child starts executing before the parent, how can we achieve this?
  - Using a delay function ***sleep()***
    - ✓ There is no guarantee that the length of this delay is adequate (***NOT a good idea!***).
  - A better solution: Using ***wait*** technique (talk about later)

Process Creation – Continuous fork()

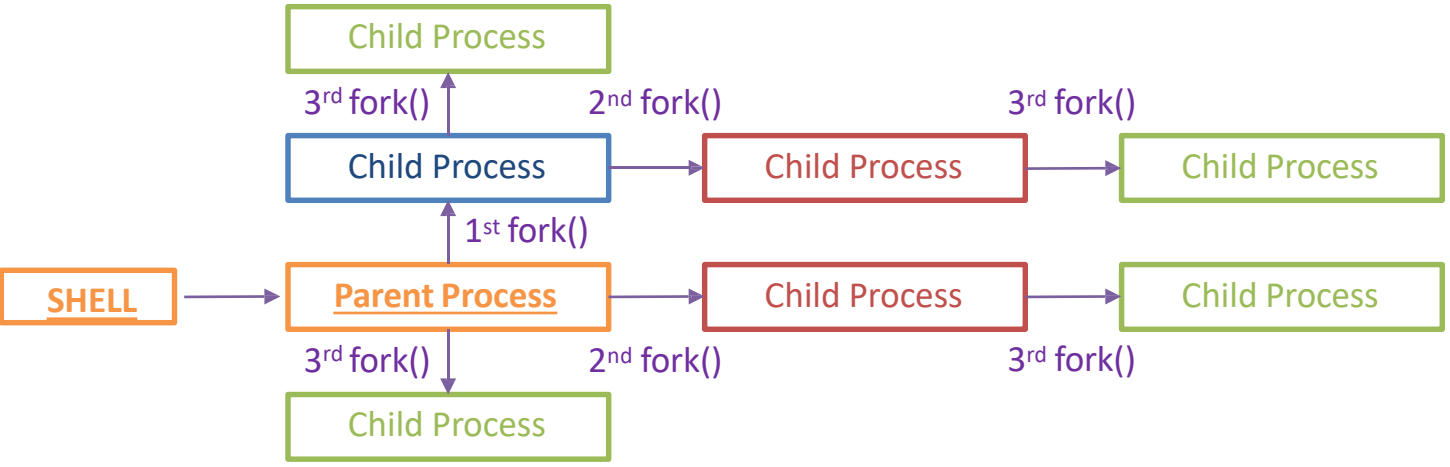
- Given ***fork-example2.c***, how many lines of "Hello World!" message displayed on the Console?

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


int main(int argc, char *argv[]) {

        fork();
        fork();
        fork();

        printf("Hello World!\n");
        return 0;
} // main
```

- There are 8 lines of "Hello World!" by counting.
- Can you explain why you get 8 lines of output with just ***3 fork() system calls***?

Process Creation – Continuous fork()

Each process prints “Hello World!” once.
We have **8** processes in total, including the parent process.
Therefore, we have **8** lines of “Hello World!”.



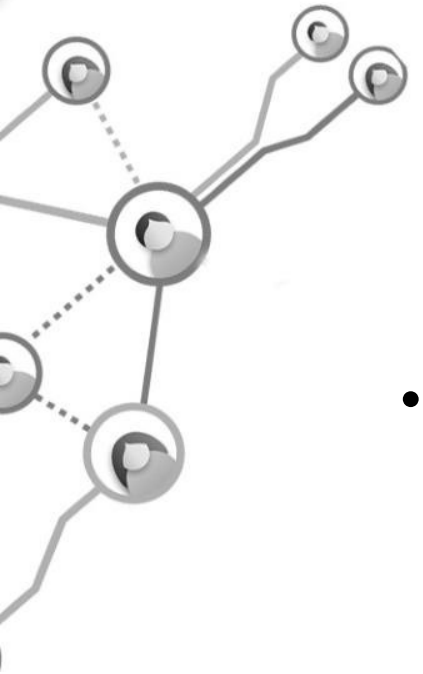
Any Question?

- How to debug a fork() creation?
  - Use commandline
  - Use VScode DEBUG CONSOLE

Process Synchronization – wait()/waitpid()

- When a process terminates, either normally or abnormally, the kernel notifies the parent by sending the ***SIGCHLD signal*** to the parent.
  - Because the termination of a child is an ***asynchronous event*** — it can happen at any time while the parent is running — this signal is the asynchronous notification from the kernel to the parent.
  - The parent can choose to ignore this signal (By default), or it can provide a function that is called when the signal occurs: ***a signal handler***.

Process Synchronization – wait()/waitpid()



- Calling ***wait()*** or ***waitpid()*** can:
  - Block, if all of its children are still running
  - Return immediately with the termination status of a child, if a child has terminated and is waiting for its termination status to be fetched
  - Return immediately with an error, if it doesn't have any child processes
- If the process is calling ***wait()*** because it received the SIGCHLD signal, we expect wait to return immediately.
  - But if we call it at any random point in time, it can block.

Process Synchronization – wait()/waitpid()

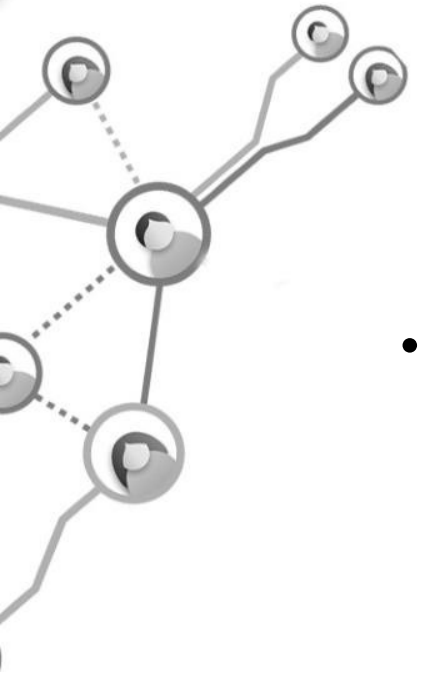
- Definition of ***wait()*** and ***waitpid()***

Function Prototype	Description
Wait for any child pid_t wait(int *statloc);	The wait function can block the caller until a child process terminates.
pid_t waitpid(pid_t pid, int *statloc, int options); ***Wait for specific child***	• The waitpid function has an option that prevents it from blocking. • The waitpid function doesn't wait for the child that terminates first; it has a number of options that control which process it waits for.

- Header File: ***<sys/wait.h>***
- Return value of both functions

Return Value	Description
-1	Error!
Process ID (> 1)	Fine!

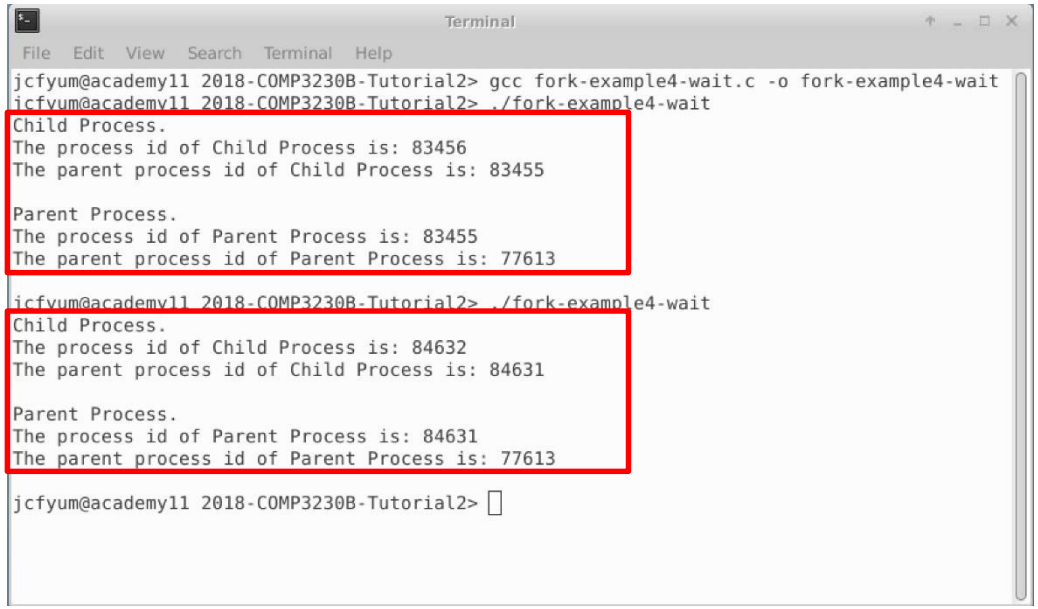
Process Synchronization – wait()/waitpid()



- If a child has already terminated and is a zombie, wait returns immediately with that child's status.
  - Otherwise, it ***blocks the caller UNTIL a child terminates***.
  - If the caller blocks and has multiple children, ***wait()*** returns ***when any ONE of them terminates***.
    - ✓ We can always tell which child terminated, because the process ID is returned by the function.
- For both functions, the argument ***statloc*** is a pointer to an integer.
  - If this argument is ***NOT a null pointer***, ***the termination status*** of the terminated process ***is stored*** in the location pointed to by the argument.
  - If we don't care about the termination status, we simply pass a null pointer as this argument.

Process Synchronization – wait()/waitpid()

- Do you still remember ***fork-example1.c ?***
- Now, we can ***ensure*** that ***the child starts executing BEFORE the parent by wait()***.
  - fork-example4-wait.c
  - Sample Output:



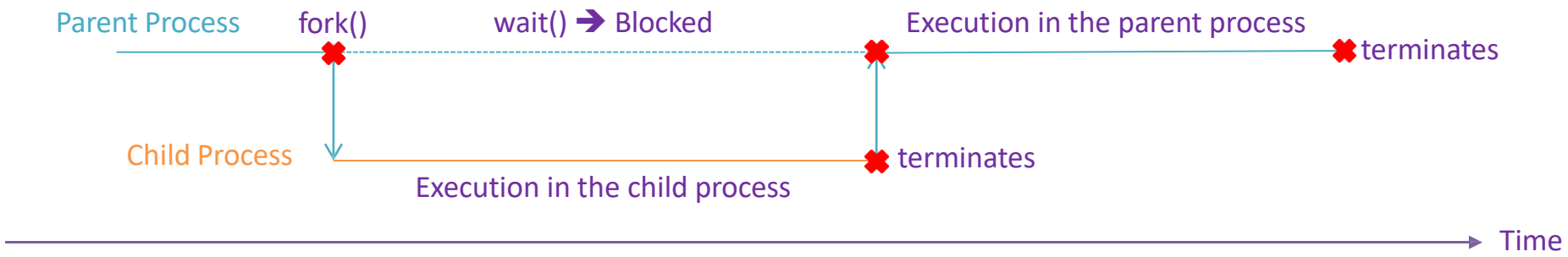
Process Synchronization – wait()/waitpid()

- fork-example4-wait.c

```
…… (header files and constant definition omitted)
int main(int argc, char *argv[]) {
        pid_t pid = fork();

        if (pid < 0) {
                // Error Occurred
                fprintf(stderr, "fork() Failed");
                exit(-1);
        } else if (pid == 0) {
                // Child Process
                printf("Child Process.\n");
                printf("The process id of Child Process is: %d\n", getpid());
                printf("The parent process id of Child Process is: %d\n", getppid());
                printf("\n");
        } else {
                // Parent Process
                wait(NULL);
                printf("Parent Process.\n");
                printf("The process id of Parent Process is: %d\n", getpid());
                printf("The parent process id of Parent Process is: %d\n", getppid());
                printf("\n");
        }
        return 0;
} // main
```

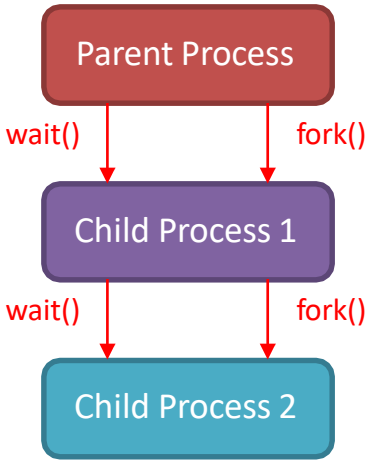
Process Synchronization – wait()/waitpid()

- fork-example4-wait.c



Process Synchronization – wait()/waitpid()

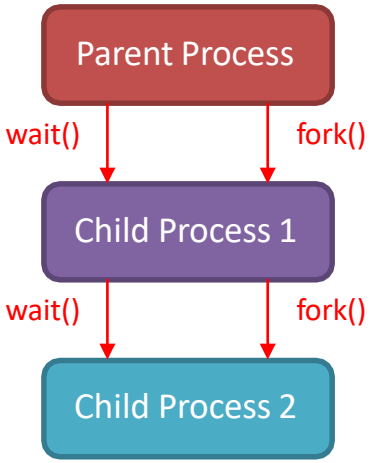
- fork-wait-example1.c




```
…… (header files and constant definition omitted)
int main(int argc, char *argv[]) {
        pid_t pid1, pid2;

        pid1 = fork();
        if (pid1 < 0) {
                // Error Occurred
                fprintf(stderr, "fork() Failed");
                exit(-1);
        } else if (pid1 == 0) {
                // Child Process 1
                pid2 = fork();
                if (pid2 < 0) {
                        // Error Occurred
                        fprintf(stderr, "fork() Failed");
                        exit(-1);
                } else if (pid2 == 0) {
                        // Child Process 2
                        printf("Child Process 2\n");
                } else {
                        // Child Process 1
                        wait(NULL);
                        printf("Child Process 1\n");
                }
        } else {
                // Parent Process
                wait(NULL);
                printf("Parent Process\n");
        }

        return 0;
} // main
```

Process Synchronization – wait()/waitpid()

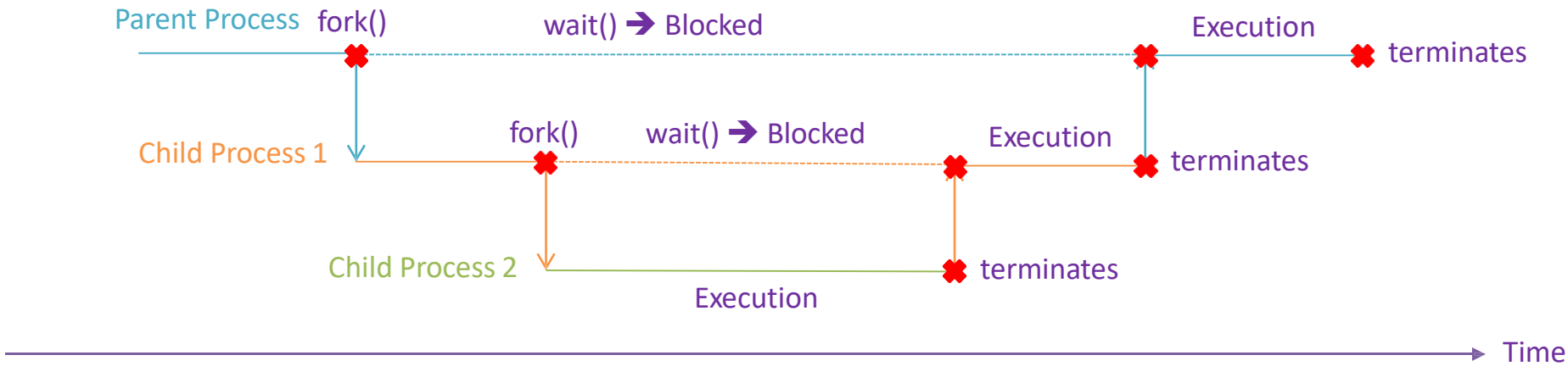
- fork-wait-example1.c (Sample Output)




```
Terminal
File  Edit  View  Search  Terminal  Help
jcfyum@academy11 2018-COMP3230B-Tutorial2> gcc fork-wait-example1.c -o fork-wait-example1
jcfyum@academy11 2018-COMP3230B-Tutorial2> ./fork-wait-example1
Child Process 2
Child Process 1
Parent Process
jcfyum@academy11 2018-COMP3230B-Tutorial2>
```

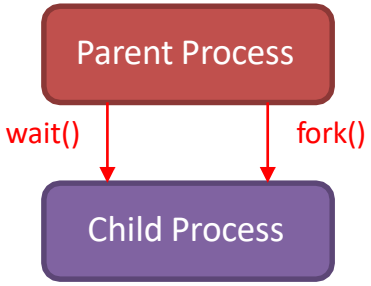
Process Synchronization – wait()/waitpid()

- fork-wait-example1.c



Process Synchronization – wait()/waitpid()

- fork-wait-example2.c

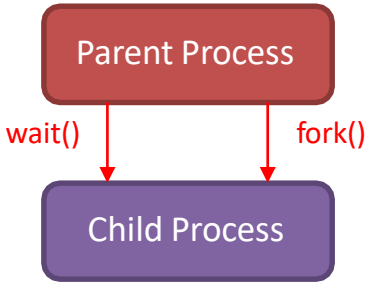


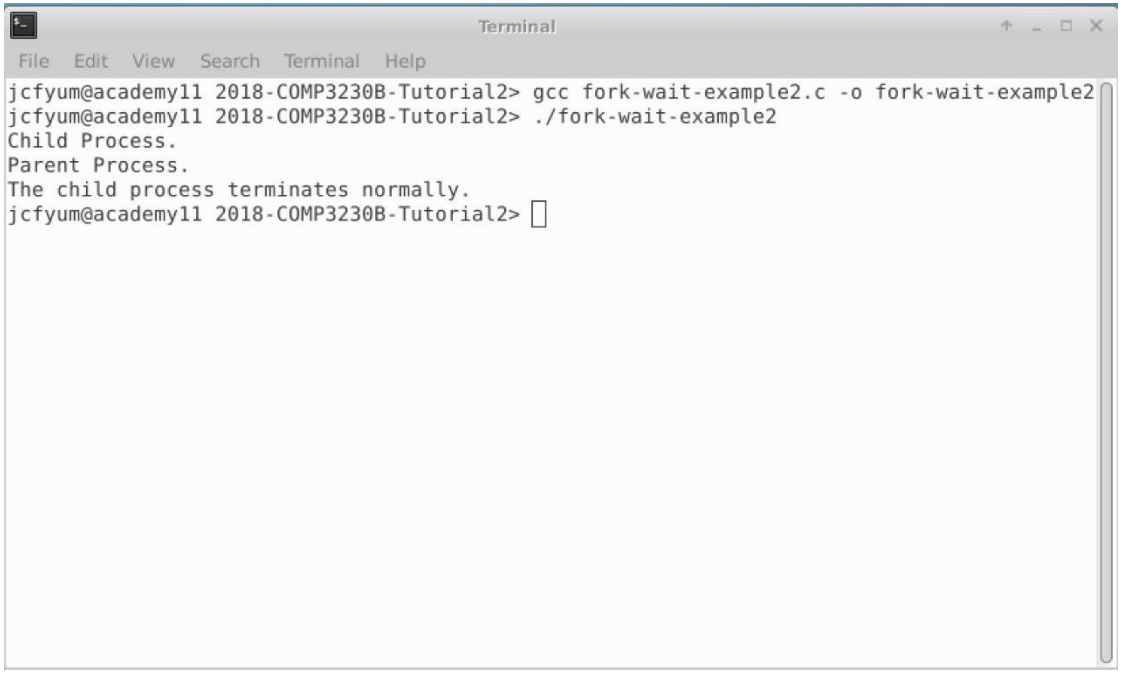
The exit() function with the argument 0 causes ***NORMAL process termination*** and the value of status (0) is returned to the parent.

```
..... (header files and constant definition omitted)
int main(int argc, char *argv[]) {
        int status;
        pid_t pid = fork();

        if (pid < 0) {
                // Error Occurred
                fprintf(stderr, "fork() Failed");
                exit(-1);
        } else if (pid == 0) {
                // Child Process
                printf("Child Process.\n");
                exit(0);
        } else {
                // Parent Process
                wait(&status);
                printf("Parent Process.\n");
                if (status == 0) {
                        printf("The child process terminates normally.\n");
                } else {
                        printf("The child process terminates abnormally.\n");
                }
        }

        return 0;
} // main
```

Process Synchronization – wait()/waitpid()

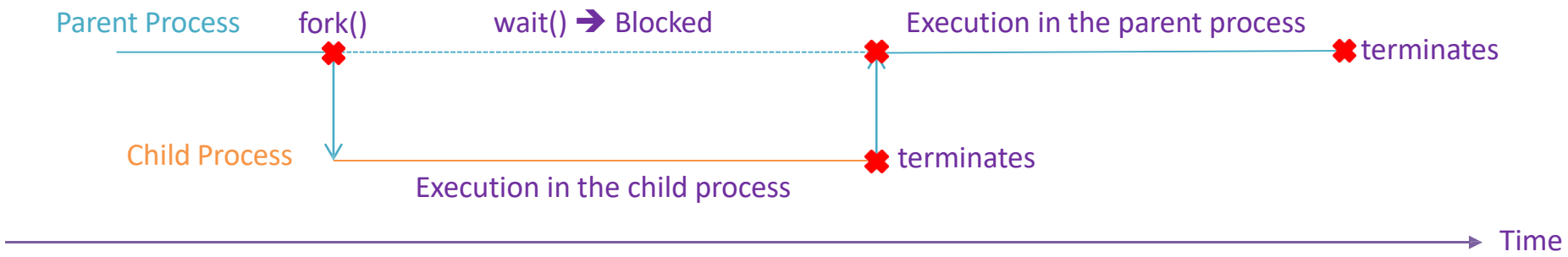
- fork-wait-example2.c (Sample Output)





Process Synchronization – wait()/waitpid()

- fork-wait-example2.c



Process Synchronization – wait()/waitpid()

```
pid_t waitpid(pid_t pid, int *statloc, int options);
```

- A process can wait for ***a SPECIFIC process*** by waitpid().
- 1st argument ***pid***

The Value of pid	Description
< -1	Waits for ***ANY child*** whose process group ID equals the absolute value of pid.
-1	Waits for ***ANY child*** process. In this respect, waitpid is equivalent to wait.
0	Waits for ***ANY child*** whose process group ID equals that of the calling process.
> 0	Waits for ***THE child*** whose process ID equals pid.

Process Synchronization – wait()/waitpid()

```
pid_t waitpid(pid_t pid, int *statloc, int options);
```

- 2nd argument ***statloc***
  - The ***waitpid()*** function returns the process ID of the child that terminated and ***stores the child's termination status*** in the memory location pointed ***to*** by ***statloc***.
  - It is same as what we have in ***wait()***.

status obtained from wait()/waitpid()

- You can **retrieve** information **from the status value** returned by wait()/waitpid() using the following macro functions:
  - WIFEXITED(status)
    - returns true **if** the child **terminated normally**
    - that is, by calling exit(), or by returning from main()
  - WEXITSTATUS(status)
    - returns the **exit code** of the child, which is stored in the **least significant 8 bits** of the status argument
    - this macro should only be called ***if WIFEXITED returned true***
  - WIFSIGNALED(status)
    - returns true **if** the child process was **terminated by a signal**
  - WTERMSIG(status)
    - returns the signal number that caused the child process to terminate
    - this macro should only be employed ***if WIFSIGNALED returned true***

Process Synchronization – wait()/waitpid()

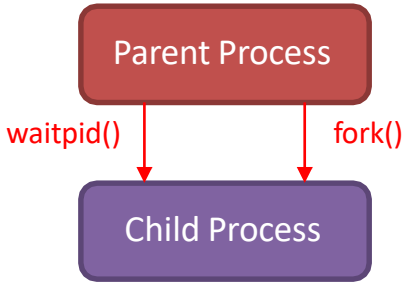
```
pid_t waitpid(pid_t pid, int *statloc, int options);
```

- 3rd argument ***options***
  - lets us further control the operation of ***waitpid()***
  - Either is 0 or is constructed from the bitwise OR of the following constants (For your own interest!)

Constant	Description
WCONTINUED	If the implementation supports job control, the status of any child specified by pid that has been continued after being stopped, but whose status has not yet been reported, is returned (XSI option).
WNOHANG	The waitpid function will not block if a child specified by pid is not immediately available. In this case, the return value is 0.
WUNTRACED	If the implementation supports job control, the status of any child specified by pid that has stopped, and whose status has not been reported since it has stopped, is returned. The WIFSTOPPED macro determines whether the return value corresponds to a stopped child process.

Process Synchronization – wait()/waitpid()

- fork-waitpid-example.c

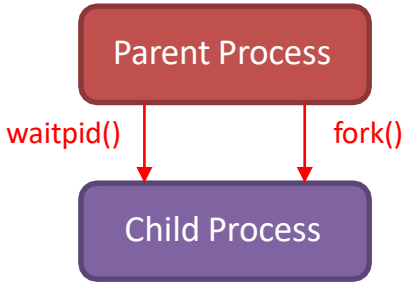


Although we have not included any exit() function in the child process block, the value of status ***zero*** is returned to the parent if the child process terminates normally without any errors.

```
..... (header files and constant definition omitted)
int main(int argc, char *argv[]) {
        int status;
        pid_t pid = fork();

        if (pid < 0) {
                // Error Occurred
                fprintf(stderr, "fork() Failed");
                exit(-1);
        } else if (pid == 0) {
                // Child Process
                printf("Child Process.\n");
        } else {
                // Parent Process
                waitpid(pid, &status, 0);
                printf("Parent Process.\n");
                if (status == 0) {
                        printf("The child process terminates normally.\n");
                } else {
                        printf("The child process terminates abnormally.\n");
                }
        }
        return 0;
} // main
```

Process Synchronization – wait()/waitpid()

- fork-waitpid-example.c (Sample Output)




```
Terminal
File  Edit  View  Search  Terminal  Help
jcfyum@academy11 2018-COMP3230B-Tutorial2> gcc fork-waitpid-example.c -o fork-waitpid-example
jcfyum@academy11 2018-COMP3230B-Tutorial2> ./fork-waitpid-example
Child Process.
Parent Process.
The child process terminates normally.
jcfyum@academy11 2018-COMP3230B-Tutorial2>
```

RECAP

Process – exec() Family

- A common programming pattern is to ***call fork followed by exec and wait (fork-exec-wait pattern)***.
  - The original process calls fork, which creates a child process.
  - The child process then uses exec to start execution of a new program.
  - Meanwhile the parent uses wait (or waitpid) to wait for the child process to finish.
- When a process calls one of the exec functions, that process is ***COMPLETELY replaced by the NEW program***, and ***the new program starts executing*** at its main function.
- The process ID does not change across an exec, because a new process is not created; exec merely replaces the current process — its text, data, heap, and stack segments — with a brand-new program from disk.

Process – exec() Family

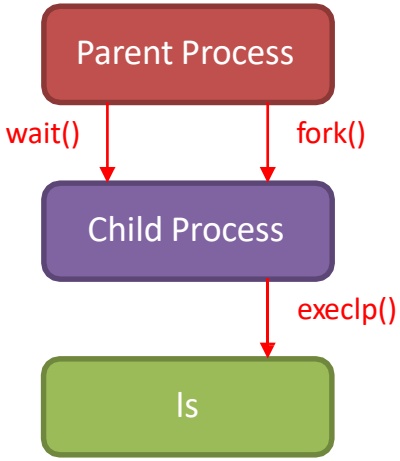
exec() functions
int execl(const char *pathname, const char *arg0, ... /* *(char *)0* */);
int execle(const char *pathname, const char *arg0, ... /* *(char *)0*, char *const envp[] */);
int execlp(const char *filename, const char *arg0, ... /* *(char *)0* */);
int execv(const char *pathname, char *const argv[]);
int execve(const char *pathname, char *const argv[], char *const envp[]);
int execvp(const char *filename, char *const argv[]);

Letter	Meaning
p	Means that the function takes a filename argument and uses the PATH environment variable to find the executable file (“Relative Path”)
l	Means that the function takes a list of arguments which is mutually exclusive with the letter v
v	Means that it takes an argv[] vector
e	Means that the function takes an envp[] environment array instead of using the current environment

- Return Value
  - –1 on error
  - no return on success

Process – exec() Family

- fork-exec-wait-example1.c

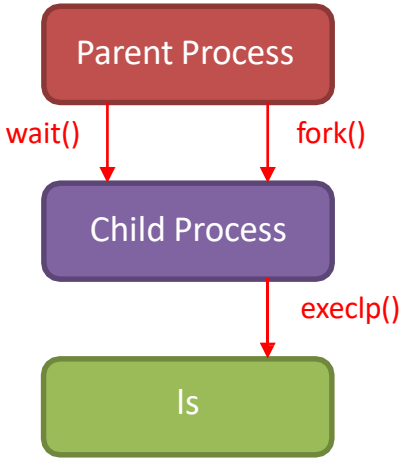



```
..... (header files and constant definition omitted)
int main(int argc, char *argv[]) {
        pid_t pid = fork();

        if (pid < 0) {
                // Error Occurred
                fprintf(stderr, "fork() Failed");
                exit(-1);
        } else if (pid == 0) {
                // Child Process
                printf("Child Process.\n");
                execlp("/bin/ls", "ls", (char *) 0);
                printf("execlp() Failed");
                exit(-1);
        } else {
                // Parent Process
                wait(NULL);
                printf("Parent Process.\n");
                printf("\n");
        }

        return 0;

} // main
```

These two lines will not be executed if execlp() executes successfully without any errors.

Process – exec() Family

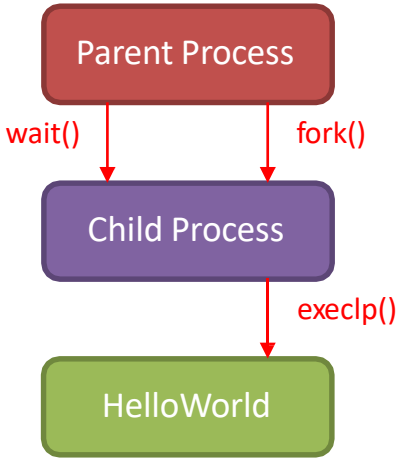
- fork-exec-wait-example1.c (Sample Output)




```
Terminal
File  Edit  View  Search  Terminal  Help
jcfyum@academy11 2018-COMP3230B-Tutorial2> gcc fork-exec-wait-example1.c -o fork-exec-wait-example1
jcfyum@academy11 2018-COMP3230B-Tutorial2> ./fork-exec-wait-example1
Child Process.
fork-example1        fork-example4-wait         fork-wait-example2
fork-example1.c      fork-example4-wait.c       fork-wait-example2.c
fork-example2        fork-exec-wait-example1    fork-waitpid-example
fork-example2.c      fork-exec-wait-example1.c  fork-waitpid-example.c
fork-example3-COW    fork-wait-example1
fork-example3-COW.c  fork-wait-example1.c
Parent Process.

jcfyum@academy11 2018-COMP3230B-Tutorial2>
```

Process – exec() Family

- fork-exec-wait-example2.c

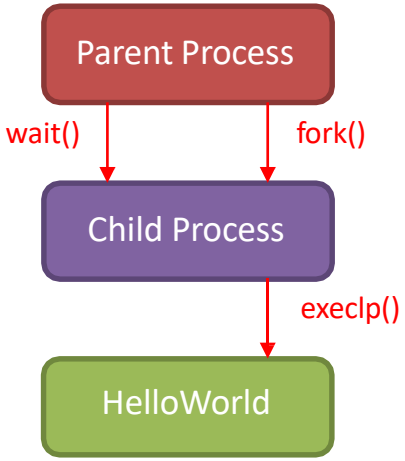


We can ask the child process to execute our own program by execlp(), e.g., HelloWorld.

```
..... (header files and constant definition omitted)
int main(int argc, char *argv[]) {
        pid_t pid = fork();

        if (pid < 0) {
                // Error Occurred
                fprintf(stderr, "fork() Failed");
                exit(-1);
        } else if (pid == 0) {
                // Child Process
                printf("Child Process.\n");
                execlp("./HelloWorld", "HelloWorld", (char *)
                0);
                printf("execlp() Failed");
                exit(-1);
        } else {
                // Parent Process
                wait(NULL);
                printf("Parent Process.\n");
                printf("\n");
        }
        return 0;

} // main
```

Process – exec() Family

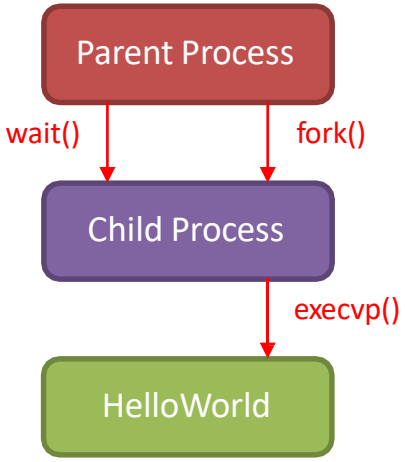
- fork-exec-wait-example2.c(Sample Output)




```
Terminal
File  Edit  View  Search  Terminal  Help
jcfyum@academy11 2018-COMP3230B-Tutorial2> gcc HelloWorld.c -o HelloWorld
jcfyum@academy11 2018-COMP3230B-Tutorial2> gcc fork-exec-wait-example2.c -o fork-exec-wait-example2
jcfyum@academy11 2018-COMP3230B-Tutorial2> ./fork-exec-wait-example2
Child Process.
HelloWorld!
Parent Process.

jcfyum@academy11 2018-COMP3230B-Tutorial2>
```

Process – exec() Family

- fork-exec-wait-example3.c

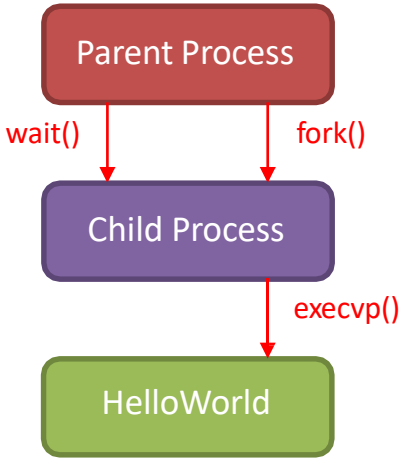


fork-exec-wait-example3.c is same as fork-exec-wait-example2.c except using ***execvp()*** in this example.

```
..... (header files and constant definition omitted)
int main(int argc, char *argv[]) {
        pid_t pid = fork();
        char * argVector[] = { "./HelloWorld", "HelloWorld", (char *) 0 };

        if (pid < 0) {
                // Error Occurred
                fprintf(stderr, "fork() Failed");
                exit(-1);
        } else if (pid == 0) {
                // Child Process
                printf("Child Process.\n");
                execvp("./HelloWorld", argVector);
                printf("execvp() Failed");
                exit(-1);
        } else {
                // Parent Process
                wait(NULL);
                printf("Parent Process.\n");
                printf("\n");
        }

        return 0;

} // main
```

Process – exec() Family

- fork-exec-wait-example3.c (Sample Output)




```
Terminal
File  Edit  View  Search  Terminal  Help
jcfyum@academy11 2018-COMP3230B-Tutorial2> gcc fork-exec-wait-example3.c -o fork-exec-wait-example3
jcfyum@academy11 2018-COMP3230B-Tutorial2> ./fork-exec-wait-example3
Child Process.
HelloWorld!
Parent Process.

jcfyum@academy11 2018-COMP3230B-Tutorial2>
```

Inter process communication

In computer science, inter-process communication (IPC) refers specifically to the mechanisms an operating system provides to allow the processes to manage shared data.

(1) Signals
(2) Pipes

https://en.wikipedia.org/wiki/Inter-process_communication

1.6 Inter process communication

(1) Signals

Signals are a limited form of inter-process communication (IPC), which are software interrupts sent to a program to indicate that an event has occurred.

#include <signal.h>

- **Sending Signals :**

  int kill(int pid, int signal)

  If pid is *greater than zero*, the signal is sent to the process whose process ID is equal to pid. If *pid is 0*, the signal is sent to all processes, except system processes.

- **Signal Handling:**

  typedef void (*sighandler_t)(int);

  sighandler_t signal(int signum, sighandler_t handler);

  signal() will call the function *handler* if the process receives a signal sig. Signal returns a pointer to function *handler* if successful or it returns an error to errno and -1 otherwise.

SIGHUP 1 /* hangup */	SIGINT 2 /* interrupt */
SIGQUIT 3 /* quit */	SIGILL 4 /* illegal instruction */
SIGABRT 6 /* used by abort */	SIGKILL 9 /* hard kill */

https://users.cs.cf.ac.uk/Dave.Marshall/C/node24.html

https://www.geeksforgeeks.org/signals-c-set-2/

https://man7.org/linux/man-pages/man3/errno.3.html

Example 1.6.1: *Signals*

- **Code:** example_1.6.1_signals.c

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

void sighup(int sig);  // function declaration
void sigint(int sig);
void sigquit(int sig);

int main()
{
    int pid;
    if ((pid = fork()) < 0) {
        exit(1);
    }

    if (pid == 0) { /* child */
        signal(SIGHUP, sighup);
        signal(SIGINT, sigint);
        signal(SIGQUIT, sigquit);
        while(1);
    }
    else { /* parent */
        printf("\nPARENT: sending SIGHUP\n");
        kill(pid, SIGHUP);

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

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

    printf("\nThis is the END of program.\n\n");
    return 0;
}
```

```c
// sighup() function definition
void sighup(int sig){
    printf("CHILD: I have received a SIGHUP\n");
    signal(SIGHUP, SIG_DFL); /* reset signal */
}

// sigint() function definition
void sigint(int sig){
    printf("CHILD: I have received a SIGINT\n");
    signal(SIGINT, SIG_DFL); /* reset signal */
}

// sigquit() function definition
void sigquit(int sig){
    printf("My DADDY has Killed me!!!\n");
    exit(0);
}
```

Child process registers 3 signal handlers

Signal handlers to be executed upon Child receives the corresponding signal sent from Parent.

Parent process sending signal to child by kill()

SIG_DFL specifies the default action for the particular signal

Example 1.6.1: *Signals*

- **Example output**

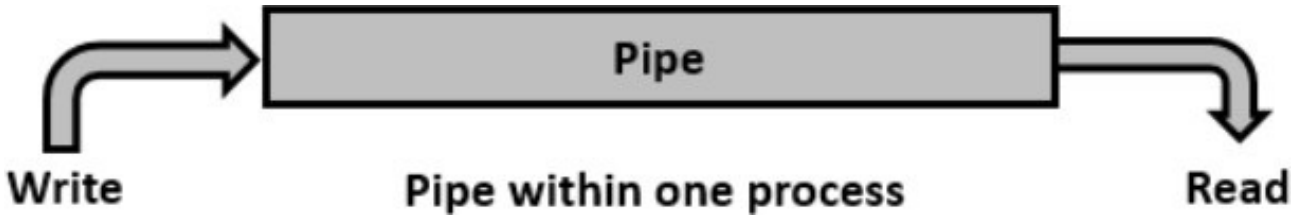
jpwang@workbench.cs.hku.hk:22 - Bitvise xterm - jpwang@workbench: ~/git_tutorial/tutorial2

```
jpwang@workbench:~/git_tutorial/tutorial2$ gcc example_1.6.1_signals.c -o example_signal
jpwang@workbench:~/git_tutorial/tutorial2$ ./example_signal

PARENT: sending SIGHUP
CHILD: I have received a SIGHUP

PARENT: sending SIGINT
CHILD: I have received a SIGINT

PARENT: sending SIGQUIT
My DADDY has Killed me!!!

This is the END of program.

jpwang@workbench:~/git_tutorial/tutorial2$
```

1.6 Inter process communication

(2) Pipes

Pipe is a communication medium between two or more related or interrelated processes. Communication is achieved by one process writing into the pipe and other reading from the pipe:



https://www.tutorialspoint.com/inter_process_communication/inter_process_communication_pipes.htm

1.6 Inter process communication

(2) Pipes

#include <unistd.h>

int pipe(int fd[2]);

- **int fd[2]:** fd[0] is set up for reading, fd[1] is set up for writing;
- **return:** 0 on success, -1 on error;

ssize_t write(int fd, void *buf, size_t count)

- **int fd**: file descriptor;
- **void *buf:** proper buffer with allocated memory;
- **size_t count:** the size of buffer;
- **Return:** the number of bytes written on success, -1 on failure;

ssize_t read(int fd, void *buf, size_t count)

- **Return:** the number of bytes read on success, -1 on failure;

Read message from pipes using stream:
https://www.gnu.org/software/libc/manual/html_node/Creating-a-Pipe.html

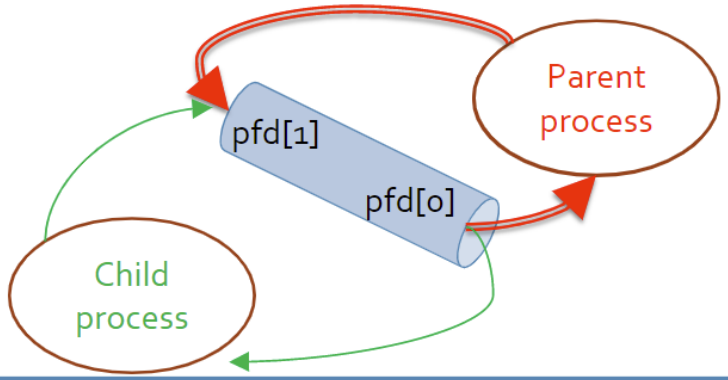
pipe( ) - Creating a Pipe

```
#include <unistd.h>
int pipe(int pfd[2]);
```

- This function creates both the reading and writing ends of the pipe
- Puts the file descriptors for the **reading** and **writing** ends of the pipe into **pfd[0]** and **pfd[1]** respectively
- Returns
  - 0 if successful
  - -1 on failure

Pipe & fork

- Typically, parent process creates a pipe just before it forks one or more child processes
- When we execute fork()
  - **All opened files and memory variables are copied**
    - So as the file descriptors created by pipe()



dup2(*newfd, oldfd*): Duplicate a (pipe) FILE descriptor

The **dup2**() system call uses the file descriptor number specified in *newfd*. In other words, the file descriptor *newfd* is adjusted so that it now refers to the same open file description as *oldfd*.

It's useful when connect or modify pipe end.

Example: pipe-dup-fork.c

The End

Materials from
Mr. Justin Yum's & Mr. Jiepeng Wang's version and lab of the other class are reused.