PROGRAMMING LAB 1
WORKING WITH PROCESSES

BEFORE YOU START…

Download pLab1.zip from course's Moodle webpage to a local directory.

Unzip pLab1.zip with any compression tool in your system.

Goto the pLab1 directory

Assume you are using WSL or Docker, right click on pLab1 directory to open a Linux shell/Mac terminal

Then, execute the following command to compile all programs

- make

When you see 

- try these command(s) in the terminal

CONTENTS

fork()

exec family of functions

Zombie process

wait(), waitpid(), wait4(), and waited()

getrusage()

REFERENCES

Man pages

The GNU C Library

- http://www.gnu.org/software/libc/manual/html_node/index.html

Programming in C – UNIX System Calls and Subroutines using C by A. D. Marshall

- http://www.cs.cf.ac.uk/Dave/C/CE.html

FORK() – CREATE NEW PROCESS

```
#include <unistd.h>
pid_t  fork (void);
```

When a process calls fork()

- a new process (child process) is created (**by cloning** the calling process) and **runs concurrently** with the parent process

The child process executes the **same program code** as the parent process and **has the same contents** in most of the variables **initially**

- However, they have **different** process **ID**s and **private memory space**

EXAMPLE 1 – LAB1-01.C

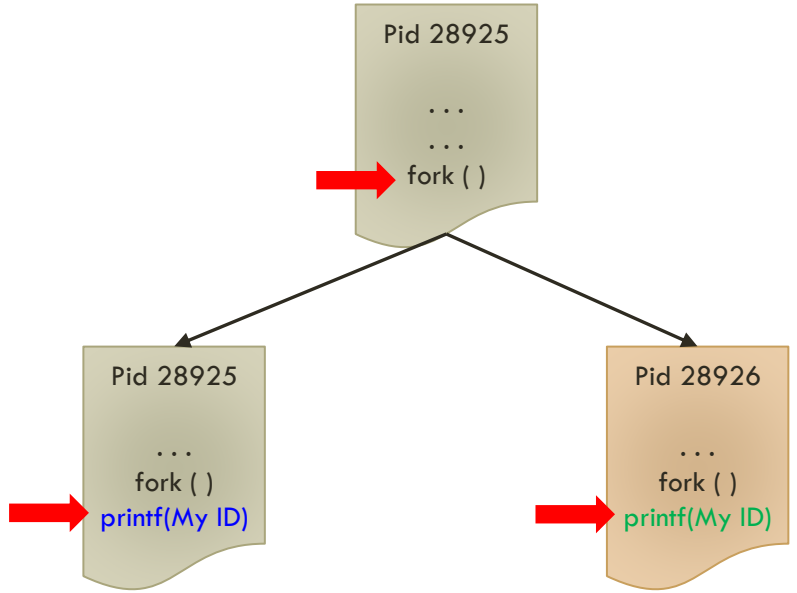
```c
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

int main() {
    printf("Process (%d) starts up\n", (int)getpid());

    fork();

    printf("My ID is %d\n", (int)getpid());

    return 0;
}
```



TRY! ./lab1-01

```
Process (28925) starts up
My ID is 28925
My ID is 28926
```

FORK() – CREATE A NEW PROCESS

Both processes continue the execution at the point immediately after the fork()

- After that, modifications of variables in one process will not be seen by the other process

child is 0

The return value of fork() is used to determine their identities

- who is the parent and who is the child
- which, in turn, usually determine which parts of the code fragment that each will execute

FORK() – TYPICAL USE CASE

```
pid_t pid = fork();


if (pid < 0) {
    //Error in creating new process
    //Report error
} else if (pid == 0) {  //only true in child process
    // only the child process enters this block
} else {  //pid > 0, only true in parent process
    // only the parent process enters this block
}
```

Actually, pid stores the process id of the **newly created** child process

EXAMPLE 2 — LAB1-02.C

```
int main() {
    pid_t who;
    int var = 999;

    printf("Process (%d) starts up\n", (int)getpid());
    who = fork();

    if (who == 0) {
        sleep(1);  //force the child to sleep first
        printf("The variable has value: %d\n", var);
        var -= 100;
        printf("The variable becomes: %d\n", var);
    } else {
        printf("The variable has value: %d\n", var);
        var += 100;
        sleep(3);  //force parent to sleep
        printf("The variable becomes: %d\n", var);
    }
    printf("My ID is %d\n", (int)getpid());

    return 0;
}
```

TRY! ./lab1-02

```
Process (29019) starts up
The variable has value: 999
The variable has value: 999
The variable becomes: 899
My ID is 29020
The variable becomes: 1099
My ID is 29019
```

所以父进程和子进程谁先启动不知道

EXEC…() - EXECUTE A PROGRAM

The purpose of this set of exec functions is for “transforming” the calling **process** to execute another program

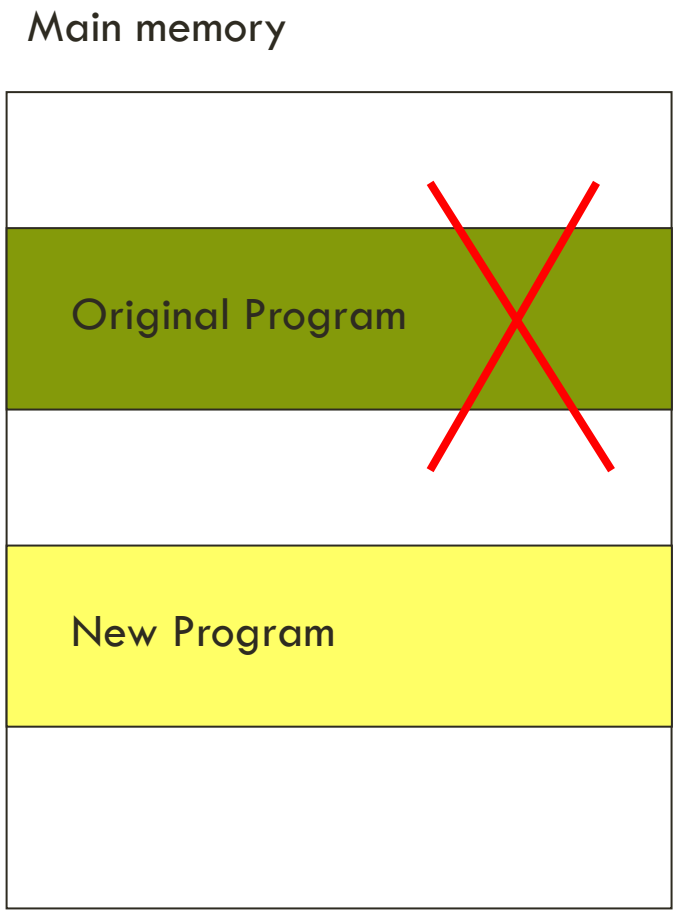
- **The effect of this system call is to replace current process’s program code with another program**

- Example usage:
  - when the user enters a command, the shell process creates a new child process by cloning itself and then the child process “transforms” itself to run the user command

EXEC…() - EXECUTE A PROGRAM

Step 1:

The process executes exec(…) system call (with necessary arguments)



Step 2:

If the call is successful, OS frees the original program of the process from memory

Step 3:

New program is allocated with new memory

Step 4:

In the new program, main() is called, and the program starts execution

EXEC...() FAMILY

There is a family of functions working on this

- execl(), execv(), execle(), execvpe(), execlp(), execvp()
- The functions are more-or-less the same, they just differ in how you specify the arguments
- In Linux, just one system call is implemented – execve(); the rest are wrapper functions that call execve() ultimately

Declared in the header file **unistd.h**

	Variadic Arguments (in list)	Arguments in Vector
Given specific pathname	int execl(const char *path, const char *arg, ...);	int execv(const char *path, char *const argv[]);
Search executable from $PATH	int execlp(const char *file, const char *arg, ...);	int execvp(const char *file, char *const argv[]);
Pass custom environmental variables to new program image	int execle(const char *path, const char *arg, ..., char * const envp[]);	int execvpe(const char *file, char *const argv[], char *const envp[]);

EXAMPLES: EXECLP() & EVECVP()

```
int execlp(const char *filename, const char *arg0, const char *arg1, …)
```

For execlp()

- 1st parameter: program name
  - The system searches the program through directories listed in PATH environment variable
- 2nd, 3rd, … parameters: corresponding to the individual argument string of the argv[] in main() of the program
  - The first one is program name
  - The last one must be NULL
- For example, to execute "ps -e -f"
  - execlp("ps", "ps", "-e", "-f", NULL);

```
int execvp(const char *filename, char *const argv[])
```

For execvp()

- Similar to execlp(), except arguments are specified as an array of char *
- execvp("ps", arg_list);
  - where arg_list is a char *argv[ ] with contents {"ps", "-e", "-f", NULL}

EXAMPLES: LAB1-EXECVP & LAB1-EXECLP

./lab1-execvp

The result is

- The initiating program is freed from memory
- The new program is executed instead
- Just like the initiating program is "replaced" by the new program
- process ID has not been changed

./lab1-execlp

lab1-execvp.c

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

int main() {
    char *a[3] = {(char *)"ls", (char *)"-l", (char *)NULL};

    printf("execvp: Press <enter> to execute 'ls -l'\n");

    getchar();

    if (execvp(a[0], a) == -1) {
        printf("execvp: error no = %s\n", strerror(errno));
    }

    printf("These lines should not be printed\n");
    return 0;
}
```

运行后直接替换掉了

If execvp() is executed successfully, the program will be "replaced" by the *ls* program

HAVE SOME PRACTICE AT HOME

Modify lab1-exec**l**p.c to execute this command:

- `ls –l –t –r`

ZOMBIE PROCESS

What happens if the parent process **does not** execute wait()/waitpid()?

- The child process becomes a Zombie process
- This “dead” process still appears in the system
- With ps command, zombie processes can be identified and displayed as

  [program_name] <defunct>

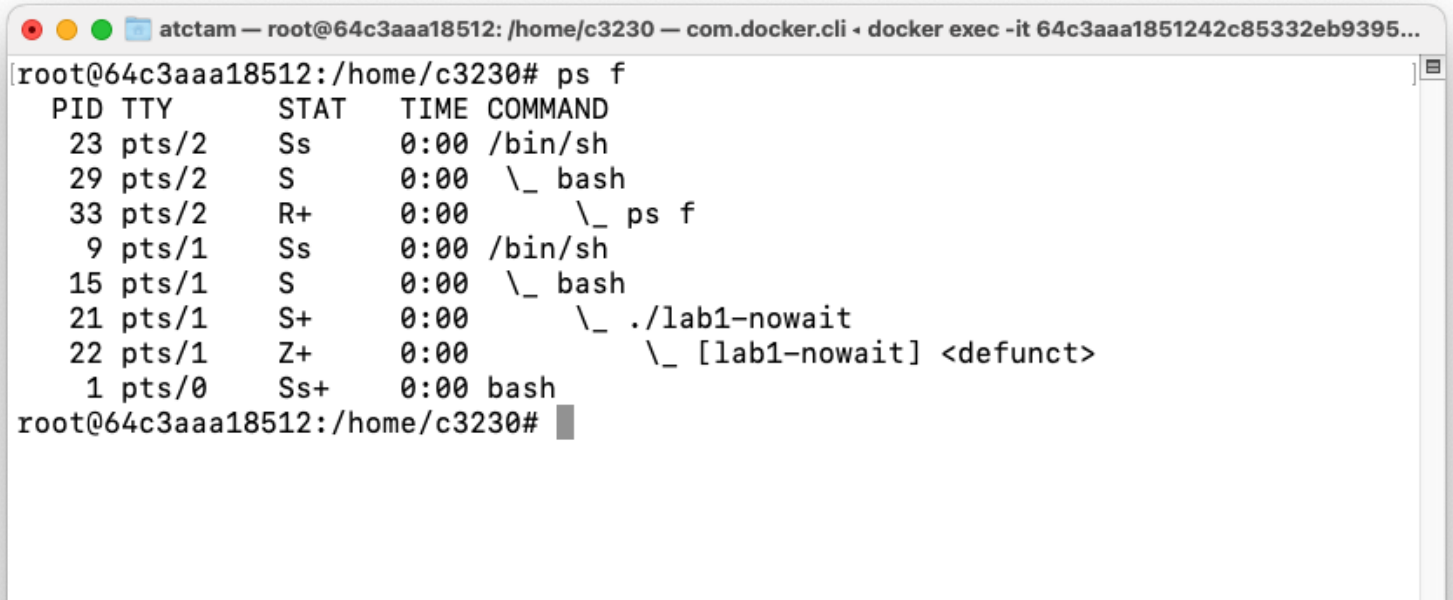
Open 2 terminals,

Run this on one terminal

TRY! ./lab1-nowait

On another terminal

TRY! ps f



WAITPID() / WAIT() / WAIT4() / WAITID() - WAIT FOR A CHILD TO TERMINATE

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t  wait(int * status);
pid_t  waitpid(pid_t pid, int *status, int options);


int waitid(idtype_t idtype, id_t id, siginfo_t *infop, int options);  //Linux only



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

pid_t wait4(pid_t pid, int *wstatus, int options, struct rusage *rusage);  //BSD style
```

WAITPID() / WAIT() / WAIT4() / WAITID() - WAIT FOR A CHILD TO TERMINATE

All of these system calls can be used by the parent process to **wait for** state change in one of its child processes that

- **the child has terminated**; or
- the child was stopped by a signal (except wait()); or
- the child was resumed by a signal (except wait())

To obtain information about the child, e.g the **exit status**

To **free up** the system resources used by a child process by "releasing" the zombie process

- waitid() may have a slightly different behavior

If a child has already changed state, then these calls return immediately

Otherwise, the caller **is blocked** **(i.e., put into blocked state)** until these calls completed or failed

- Either a child has terminated or stopped or resumed
- Or the caller is interrupted by signal

WAITPID() / WAIT() / WAIT4() / WAITID() - WAIT FOR A CHILD TO TERMINATE

wait()	waitpid()	wait4()	waitid()
• Blocking system calls • Wait for any child process to terminate • Can retrieve termination exit status (exit code or signal code) • Free the terminated child zombie process			
	• Can set to run in non-blocking mode • Can just wait for any child process in a specific process group • Can also wait for a child that has terminated or stopped or continued		
		• Can retrieve terminated child's resource usage information	• Can detect a child has terminated but without removing its zombie process

WAITPID() & WAIT()

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

"**status**" stores the exit status of a terminated child process

- **Can pass a "NULL" pointer** if the parent does not want to get the exit status

To wait for **any** terminated child process without the need of obtaining exit status

- wait(NULL); OR
- waitpid(-1, NULL, 0);

To wait for a specific child with PID = 3241

- waitpid(3241, NULL, 0);

WAITPID()

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

Parameters:

- 1st Parameter: Process ID of the child process that parent wants to wait for (type: pid_t)
  - If set to **-1**, wait any child process
  - There are other usages which can set it to < -1 or 0, see man page
- 2nd Parameter: status (pass by pointer), stores the exit status of child process
  - Can use "**NULL**" if we don't need any exit status
  - However, cannot use the returned data directly
  - To obtain the exit code of the child process, use **WEXITSTATUS(status)**
- 3rd Parameter: options, normally put a zero here, see man page for detailed usage

Return Value:

- On success, the **process ID** of the exited child is returned
- If WNOHANG option was specified, but child process(es) has/have not yet changed state, then a zero is returned (i.e., non-blocking)
- On error, **-1** is returned

RETRIEVE EXIT STATUS

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***

lab1-waitpid.c

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

    if (pid < 0) {
        printf("fork: error no = %s\n", strerror(errno));
        exit(-1);
    } else if (pid == 0) {
        printf("child: I am a child process, with pid: %d\n", (int) getpid());
        printf("child: exited\n");
        exit(0);
    } else {
        printf("I am a parent process, with pid: %d and my child's pid is %d\n", (int) getpid(), (int) pid);

        pid_t child_pid;
        int status;
        child_pid = waitpid(pid, &status, 0);

        printf("Child process (%d) exited, with exit status %d\n", (int) child_pid, WEXITSTATUS(status));

        printf("Press <enter> to continue\n");
        getchar();
    }

    return 0;
}
```

TRY! ./lab1-waitpid

TRY THIS AT HOME

Modify lab1-waitpid.c and change the statement **exit(0)** to

- exit(111), then compile the program and check the result
- exit(555), then compile the program and check the result

GETRUSAGE() - RESOURCES USAGE

```
int getrusage (int who, struct rusage *rusage)
```

Measure the resources (e.g., CPU time) used by the calling process or its child processes

- if ***who*** equals **RUSAGE_SELF**
  - information shall be returned about resources used by the calling process itself
- if ***who*** equals **RUSAGE_CHILDREN**
  - information shall be returned about resources used by those **terminated and been waited-for** children of the current process
- The returned resource usages information is stored in the provided object of type struct rusage
  - Check the man page to find out the types of resource info

TRY! ./lab1-getrusage

In another terminal, use kill command to terminate the child process

WAIT4()

```
pid_t wait4(pid_t pid, int *wstatus, int options,
            struct rusage *rusage);
```

Similar to waitpid() **except** that it can return resource usage information about the child in the structure pointed to by rusage.

Parameters:

- 1st Parameter: Process ID of the child process that parent wants to wait for
  - If set to **-1**, wait any child process
- 2nd Parameter: wstatus (pass by pointer), stores the exit status of child process
- 3rd Parameter: options, similar to waitpid
- 4th Parameter: rusage (pass by pointer), filled with accounting information about the child

Return Value:

- On success, the **process ID** of the exited child is returned
- If WNOHANG option was specified, but child process(es) has/have not yet changed state, then a zero is returned (i.e., non-blocking)
- On error, **-1** is returned

WAIT4()

We had demonstrated this before

TRY! ./lab1-getrusage

In another terminal, use kill command to terminate the child process

TRY! ./lab1-waitrusage

In another terminal, use kill command to terminate the child process

WAITID() //LINUX ONLY

Similar to waitpid() **except** that we can instruct the system **not to clear the zombie process** and still **leave it in waitable state**

waitid(idtype_t idtype, id_t id, siginfo_t * infop, int options);

- First & second argument: idtype, id
  - P_PID: wait for child with process ID of id
  - P_GPID: Wait for any child whose process group ID matches id
  - P_ALL: ignore id, wait for any child
- Third argument: siginfo_t type pointed by infop
  - Contains some info about the terminated child process, e.g., process ID, exit status, exit code, etc.
  - Read the man page to get more details
- Fourth argument: options (can have more than one option)
  - WNOWAIT: Leave the child in a waitable state
  - WNOHANG: If no child has been terminated or stopped, returns immediately
  - … (See man page)

WAITID()

Return Value:

- On success, a zero is returned
- if WNOHANG was specified and no child process(es) specified by id has yet changed state, a zero is returned.
- On error, -1 is returned

WAITID()

```c
/* Lab 1 - Process
*  lab1-waitid.c
*/

int main() {
  pid_t pid = fork();

  if (pid < 0) {
    printf("fork: error no = %s\n", strerror(errno));
    exit(-1);
  } else if (pid == 0) {
    printf("child: I am a child process, with pid %d\n", (int)getpid());
    printf("Use the kill system command to terminate me !!!\n");
    while (1);  // use the kill system command and send the SIGTERM signal to kill it!!!
  } else {
    siginfo_t info;
    int status;
    int ret = waitid(P_ALL, 0, &info, WNOWAIT | WEXITED);  // wait for child to terminate
    if (!ret) {
      printf("Child with process id: %d has exited\n", (int) info.si_pid);
      waitpid(info.si_pid, &status, 0);
      printf("Child process (%d) exited, with signal status %d\n",
                  (int) info.si_pid, WTERMSIG(status));
    } else {
      perror("waitid");
    }
  }
  return 0;
}
```

/PROC FILE SYSTEM

The proc file system is an interface to kernel data structures

- /proc/cpuinfo : collection of CPU and system architecture information
- /proc/meminfo : statistics information about the memory usage
- Each process (running/zombie) has a set of information stored under /proc/<pid>/
  - /proc/<pid>/stat : Status information about the process
  - /proc/<pid>/status : Status information about the process (in human reading form)

…

- For more, read http://linux.die.net/man/5/proc

Note that the zombie process as well as it's information stored in will be cleared when the parent process call waitpid()

/PROC/<PID>/STAT

(1) Process ID

(2) File name

(3) Status: R – Running, S – Sleeping, Z – Zombie

(4) PID of parent

(5) Group ID

…

(14) utime : Amount of time that this process has been scheduled in **user mode** (CPU resource usage count) in units of clock ticks

(15) stime : Amount of time that this process has been scheduled in **kernel mode** (CPU resource usage count) in units of clock ticks

- You can obtain (clock ticks/second) using *sysconf(_SC_CLK_TCK)*

LAB1-READPROC.C

In the terminal

TRY! ./lab1-readproc

- The process works on some computation
- Then, pause and ask you to access its /proc/..../stat with another terminal
- Finally, it accesses its own stat info at /proc/..../stat and prints the user time and system time

```
My process ID is: 108
Please use another terminal to run this command to access the
path: /proc/108/stat -- 'cat /proc/108/stat'
Press enter to continue after you obtained the information

The number of clock ticks per second is: 100
User time: 1.080000 s
In system time: 0.000000 s
The process is in state: R
```

```
108 (lab1-readproc) S 16 108 9 34817 108 4194304 80 0 1 0 108
0 0 0 20 0 1 0 461282 2834432 234 18446744073709551615
94110558928896 94110558930289 140722357105344 0 0 0 0 0 0 0 0
0 17 4 0 0 0 0 0 94110558940520 94110558941200 94110564966400
140722357111086 140722357111102 140722357111102
140722357112808 0
```