Operating System (OS)
CS232

Process: API and Implementation

Dr. Muhammad Mobeen Movania

Outlines

- What is an API?
- Dual mode operation and transitions
- POSIX
- Types of system calls
- Process management system calls
- fork(), wait(), exec() with examples
- Summary

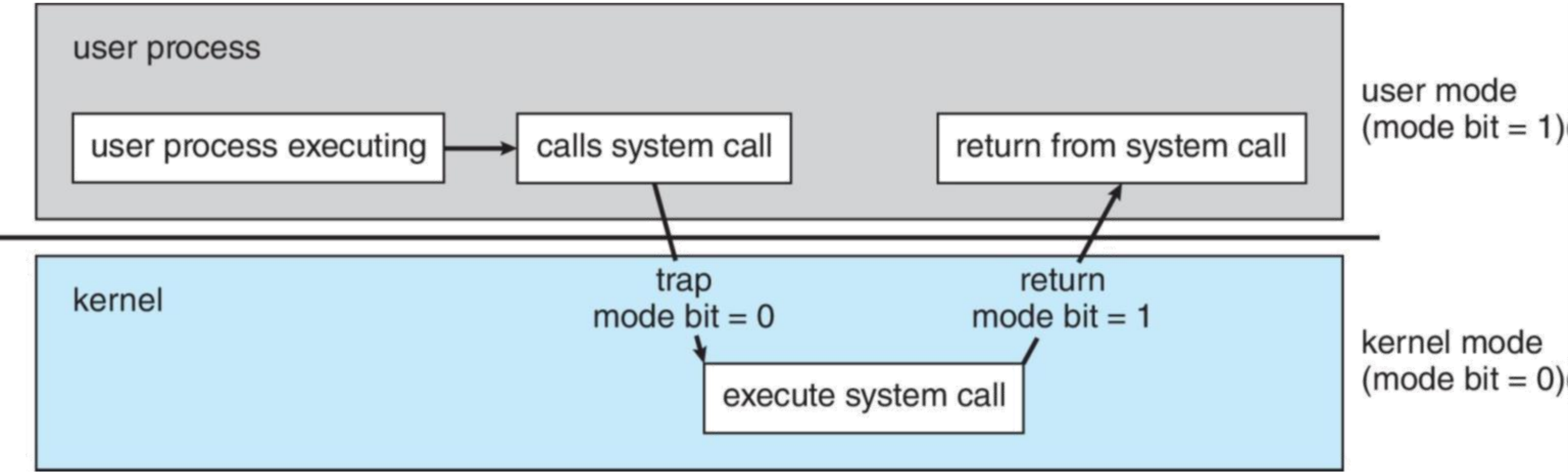
What is an API?

- API (Application programming interface) is a collection of functions that are provided to users to control any system or program
- All OS provide functions called **system calls**
- System calls
  - Provide access to hardware and other privileged accesses to user processes
  - Are always run in kernel mode (privilege mode)

Dual mode operation

- OS provide two modes of operation
  - User mode (non-privileged)
  - Kernel mode (privileged)
- Why
  - Protection of hardware and other system components
- How
  - A special mode bit is provided in hardware
  - Changed through a system call
- When running a system call, the process must transition from user mode to kernel mode

Transition from user to kernel mode

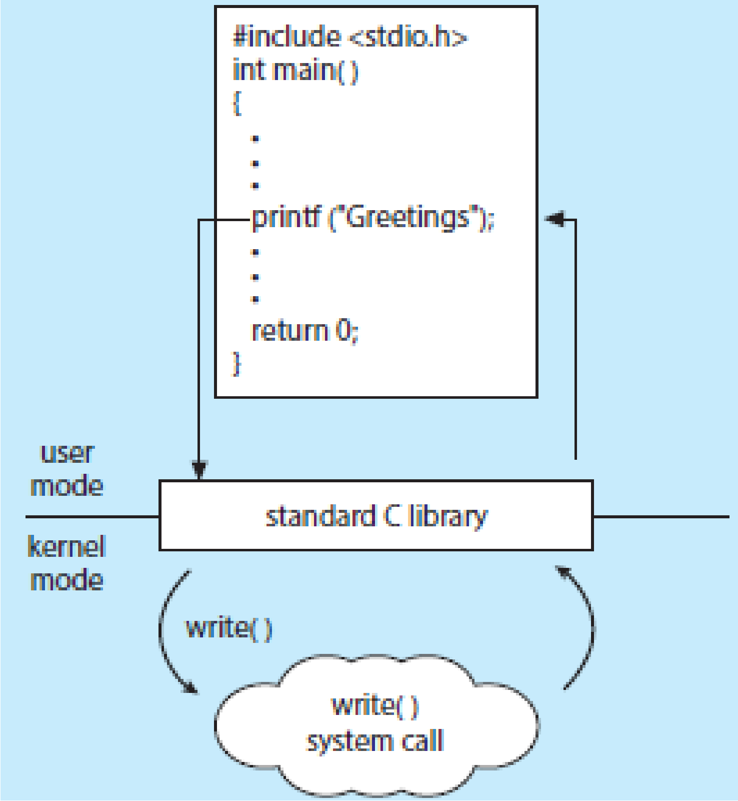


Silberschatz, Galvin and Gagne, “Operating Systems Concepts”, 10th Edition, Chapter 1, pp: 25, Wiley, 2018.

POSIX

- POSIX (Portable OS Interface) Standard
  - API Standard
  - Ensures compatibility across different OS
  - Programs using POSIX API sure to run on POSIX-compliant OS
  - Most OS provide some sort of POSIX compliance

- Libraries provide an easy-to-use interface to make system calls
  - C language has libc library
  - printf() calls the write() system call



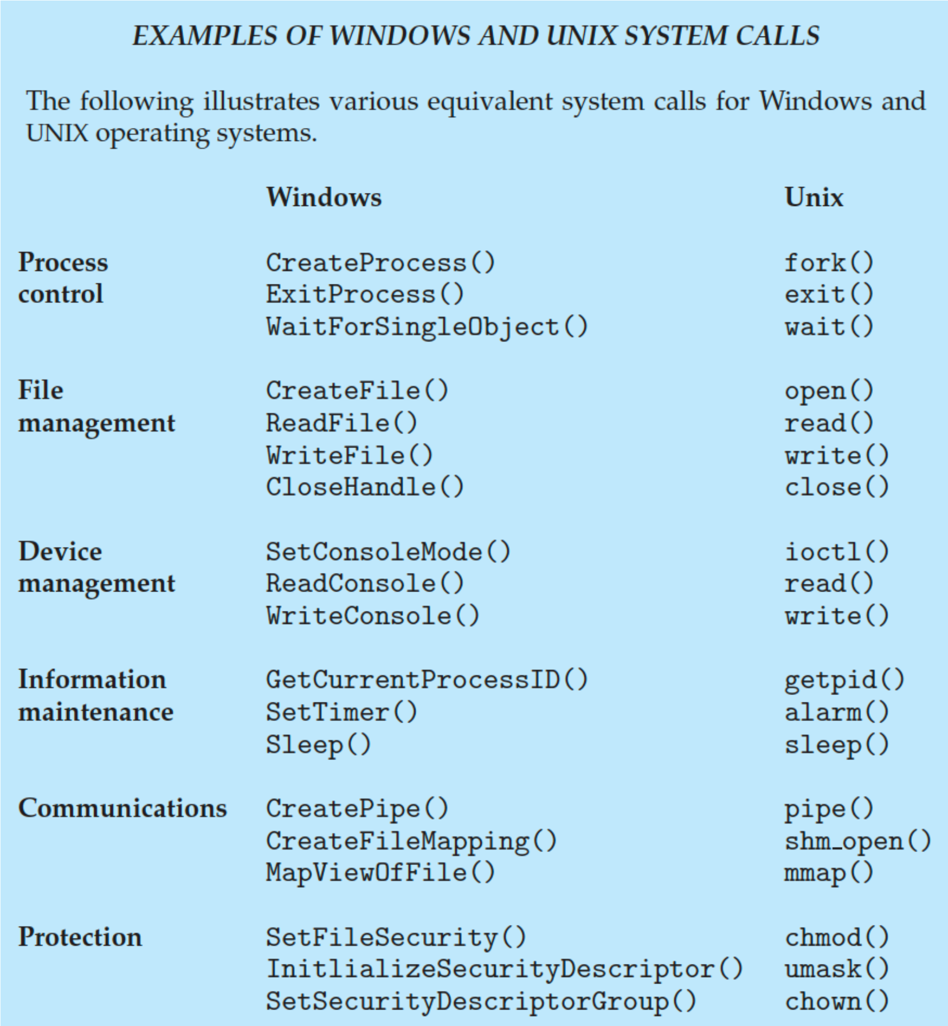
```
#include <stdio.h>
int main( )
{
  .
  .
  .
  printf ("Greetings");
  .
  .
  .
  return 0;
}
```

Silberschatz, Galvin and Gagne, "Operating Systems Concepts", 10th Edition, Chapter 2, pp: 69, Wiley, 2018.

Types of System Calls

- System calls may be grouped into the following 6 types
  - Process control
  - File management
  - Device management
  - Information maintenance
  - Communications
  - Protection

Examples of Windows and Unix System Calls



EXAMPLES OF WINDOWS AND UNIX SYSTEM CALLS

The following illustrates various equivalent system calls for Windows and UNIX operating systems.

	Windows	**Unix**
Process control	`CreateProcess()`	`fork()`
	`ExitProcess()`	`exit()`
	`WaitForSingleObject()`	`wait()`
File management	`CreateFile()`	`open()`
	`ReadFile()`	`read()`
	`WriteFile()`	`write()`
	`CloseHandle()`	`close()`
Device management	`SetConsoleMode()`	`ioctl()`
	`ReadConsole()`	`read()`
	`WriteConsole()`	`write()`
Information maintenance	`GetCurrentProcessID()`	`getpid()`
	`SetTimer()`	`alarm()`
	`Sleep()`	`sleep()`
Communications	`CreatePipe()`	`pipe()`
	`CreateFileMapping()`	`shm_open()`
	`MapViewOfFile()`	`mmap()`
Protection	`SetFileSecurity()`	`chmod()`
	`InitlializeSecurityDescriptor()`	`umask()`
	`SetSecurityDescriptorGroup()`	`chown()`

Silberschatz, Galvin and Gagne, “Operating Systems Concepts”, 10th Edition, Chapter 2, pp: 68, Wiley, 2018.

Process Management System Calls in Unix based Systems

- The following 4 functions are provided for process management
  - fork() : for creation of a new process
  - exec() : for creation of a new process
  - exit() : for termination of a process
  - wait() : to wait for a created process to complete

fork()

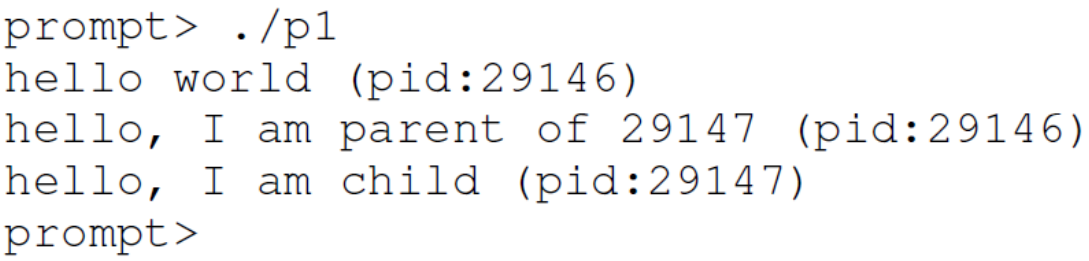
- Creates a new process which is an exact copy of its parent process
- On success, the fork() system call returns twice
  - For the newly created child process, in which case it returns 0
  - For the parent process, in which case it returns the PID of the child process
- Execution of program continues to the statements after fork() with each process having its own address space

Example of fork()

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

int main(int argc, char *argv[]) {
   printf("hello world (pid:%d)\n", (int) getpid());
   int rc = fork();
   if (rc < 0) {
      // fork failed
      fprintf(stderr, "fork failed\n");
      exit(1);
   } else if (rc == 0) {
      // child (new process)
      printf("hello, I am child (pid:%d)\n", (int) getpid());
   } else {
     // parent goes down this path (main)
     printf("hello, I am parent of %d (pid:%d)\n",rc,(int)
                                                      getpid());
   }
   return 0;
}
```

Output

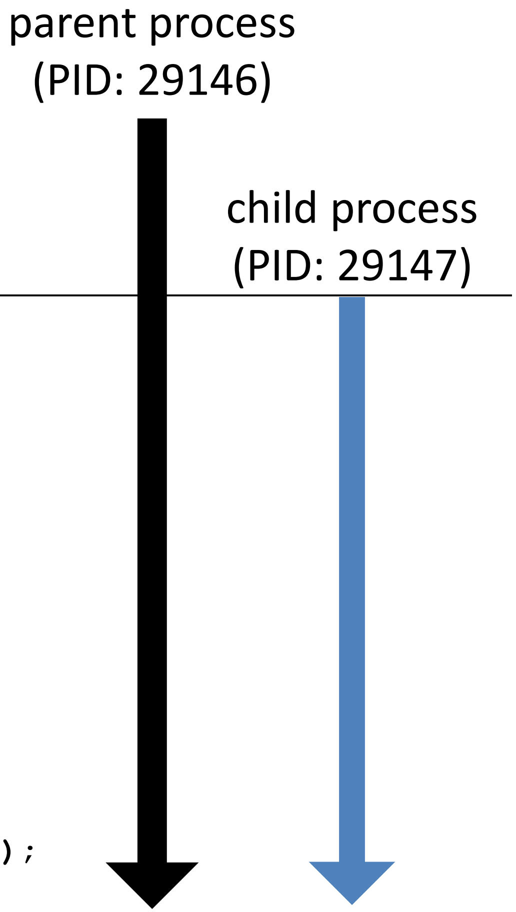


```
prompt> ./p1
hello world (pid:29146)
hello, I am parent of 29147 (pid:29146)
hello, I am child (pid:29147)
prompt>
```

What’s going on?

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

int main(int argc, char *argv[])
{
   printf("hello world (pid:%d)\n", (int) getpid());
   int rc = fork();
   if (rc < 0)
   {
      // fork failed
      fprintf(stderr, "fork failed\n");
      exit(1);
   }
   else if (rc == 0)
   {
      // child (new process)
      printf("hello, I am child (pid:%d)\n", (int) getpid());
   }
   else
   {
     // parent goes down this path (main)
     printf("hello, I am parent of %d (pid:%d)\n",rc,(int)getpid());
   }
   return 0;
}
```

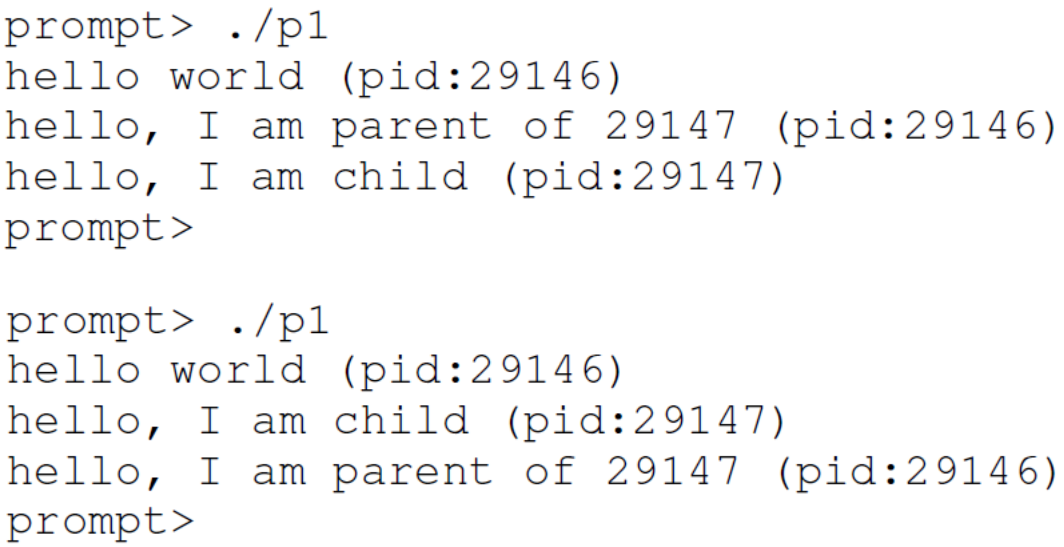


parent process
(PID: 29146)

child process
(PID: 29147)

Issues in the last code?

- Non-deterministic
  - After the fork call, the child process or parent process might run its statements depending on who gets scheduled on the CPU



```
prompt> ./p1
hello world (pid:29146)
hello, I am parent of 29147 (pid:29146)
hello, I am child (pid:29147)
prompt>
```

```
prompt> ./p1
hello world (pid:29146)
hello, I am child (pid:29147)
hello, I am parent of 29147 (pid:29146)
prompt>
```

wait()

- Sometimes, its useful for a parent process to wait for a child process to finish.
- Achieved through wait() or waitpid().
- Parent process calls wait() to delay its execution until the child finishes.
- When child is done, wait() returns to the parent
- Why add wait()
  - Makes output **deterministic** that is you are always sure that the output statements of child process will be printed first before the parent’s output statements are printed

Example of wait()

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

int main(int argc, char *argv[]) {
   printf("hello world (pid:%d)\n", (int) getpid());
   int rc = fork();
   if (rc < 0) {
      // fork failed
      fprintf(stderr, "fork failed\n");
      exit(1);
   } else if (rc == 0) {
      // child (new process)
      printf("hello, I am child (pid:%d)\n", (int) getpid());
   } else {
     // parent goes down this path (main)
     int rc_wait = wait(NULL);
     printf("hello, I am parent of %d (rc_wait:%d) (pid:%d)\n",rc,rc_wait,(int)
                                                getpid());
   }
   return 0;
}
```

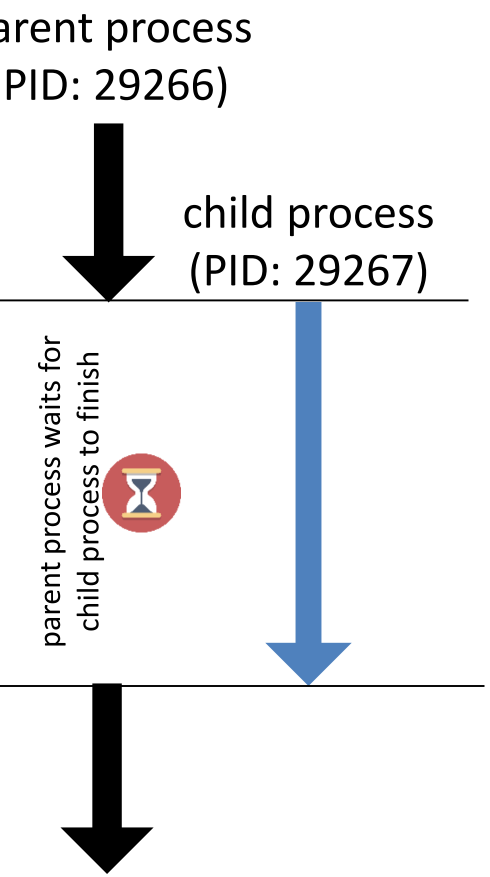
Output

```
prompt> ./p2
hello world (pid:29266)
hello, I am child (pid:29267)
hello, I am parent of 29267 (rc_wait:29267) (pid:29266)
prompt>
```

What’s going on?

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

int main(int argc, char *argv[]) {
   printf("hello world (pid:%d)\n", (int) getpid());
   int rc = fork();
   if (rc < 0) {
      // fork failed
      fprintf(stderr, "fork failed\n");
      exit(1);
   } else if (rc == 0) {
      // child (new process)
      printf("hello, I am child (pid:%d)\n", (int) getpid());
   } else {
     // parent goes down this path (main)
     int rc_wait = wait(NULL);
     printf("hello, I am parent of %d (rc_wait:%d) (pid:%d)\n",rc,
                                             rc_wait,(int)getpid());
   }
   return 0;
}
```



Output

```
prompt> ./p2
hello world (pid:29266)
hello, I am child (pid:29267)
hello, I am parent of 29267 (rc_wait:29267) (pid:29266)
prompt>
```

exec()

- Used when you want to run a program which is different from the calling program
- Linux has six variants of exec()
  - execl, execlp(), execle(), execv(), execvp(), and execvpe()
- The example code on the next slide runs a word counting program (wc) with the source file given as argument
  - wc returns no. of lines, words and bytes in the given file

Example of exec()

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

int main(int argc, char *argv[]) {
   printf("hello world (pid:%d)\n",(int)getpid());
   int rc = fork();
   if (rc < 0) {
      // fork failed
      fprintf(stderr, "fork failed\n");
      exit(1);
   } else if (rc == 0) {
      // child (new process)
      printf("hello, I am child (pid:%d)\n", (int) getpid());
      char *myargs[3];
      myargs[0] = strdup("wc");   // program: "wc" (word count)
      myargs[1] = strdup("p3.c"); // argument: file to count
      myargs[2] = NULL;           // marks end of array
      execvp(myargs[0], myargs);  // runs word count
      printf("this shouldn’t print out");
   } else {
     // parent goes down this path (main)
     int rc_wait = wait(NULL);
     printf("hello, I am parent of %d (rc_wait:%d) (pid:%d)\n",rc,rc_wait,(int)
                                                    getpid());
   }
   return 0;
}
```

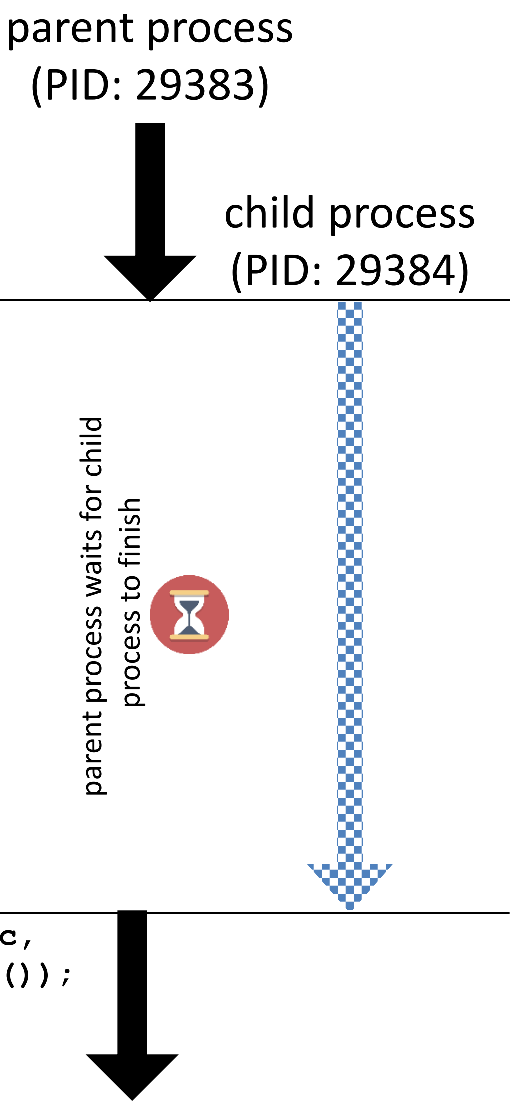
Output

```
prompt> ./p3
hello world (pid:29383)
hello, I am child (pid:29384)
      29     107    1030 p3.c
hello, I am parent of 29384 (rc_wait:29384) (pid:29383)
prompt>
```

What’s going on?

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

int main(int argc, char *argv[]) {
   printf("hello world (pid:%d)\n",(int)getpid());
   int rc = fork();
   if (rc < 0) {
      // fork failed
      fprintf(stderr, "fork failed\n");
      exit(1);
   } else if (rc == 0) {
      // child (new process)
      printf("hello, I am child (pid:%d)\n", (int) getpid());
      char *myargs[3];
      myargs[0] = strdup("wc");    // program: "wc" (word count)
      myargs[1] = strdup("p3.c"); // argument: file to count
      myargs[2] = NULL;            // marks end of array
      execvp(myargs[0], myargs);  // runs word count
      printf("this shouldn’t print out");
   } else {
     // parent goes down this path (main)
     int rc_wait = wait(NULL);
     printf("hello, I am parent of %d (rc_wait:%d) (pid:%d)\n", rc,
                                         rc_wait,(int) getpid());
   }
   return 0;
}
```



More about exec()

- exec() does not create a new process, it transform the currently running process (p3) into a different program (wc)
- How?
  - From the given executable name and arguments, it loads code (and static data) from the executable and overwrites current process's code (and static data)
  - Stack, heap and other parts of memory space are reinitialized
- exec() if successful never returns
- Combining fork() and exec() allows creation of Shells, output redirection (>) etc.
- Unix pipes (|) are implemented by combination of fork() with pipe() system call

Summary

- We looked at three process creation API, fork(), exec() and wait()
  - fork() is used to create a new child process which is exact replica of the parent process
  - exec() allows a child to execute an entirely new program
  - wait() allows a parent to wait for its child to complete execution
- Unix shell uses fork(), wait(), and exec() to launch user commands
- Separation of fork() and exec() enables features like input/output redirection, pipes, and other cool features