CS415: Systems Programming

Process Related System Calls (Cont.)

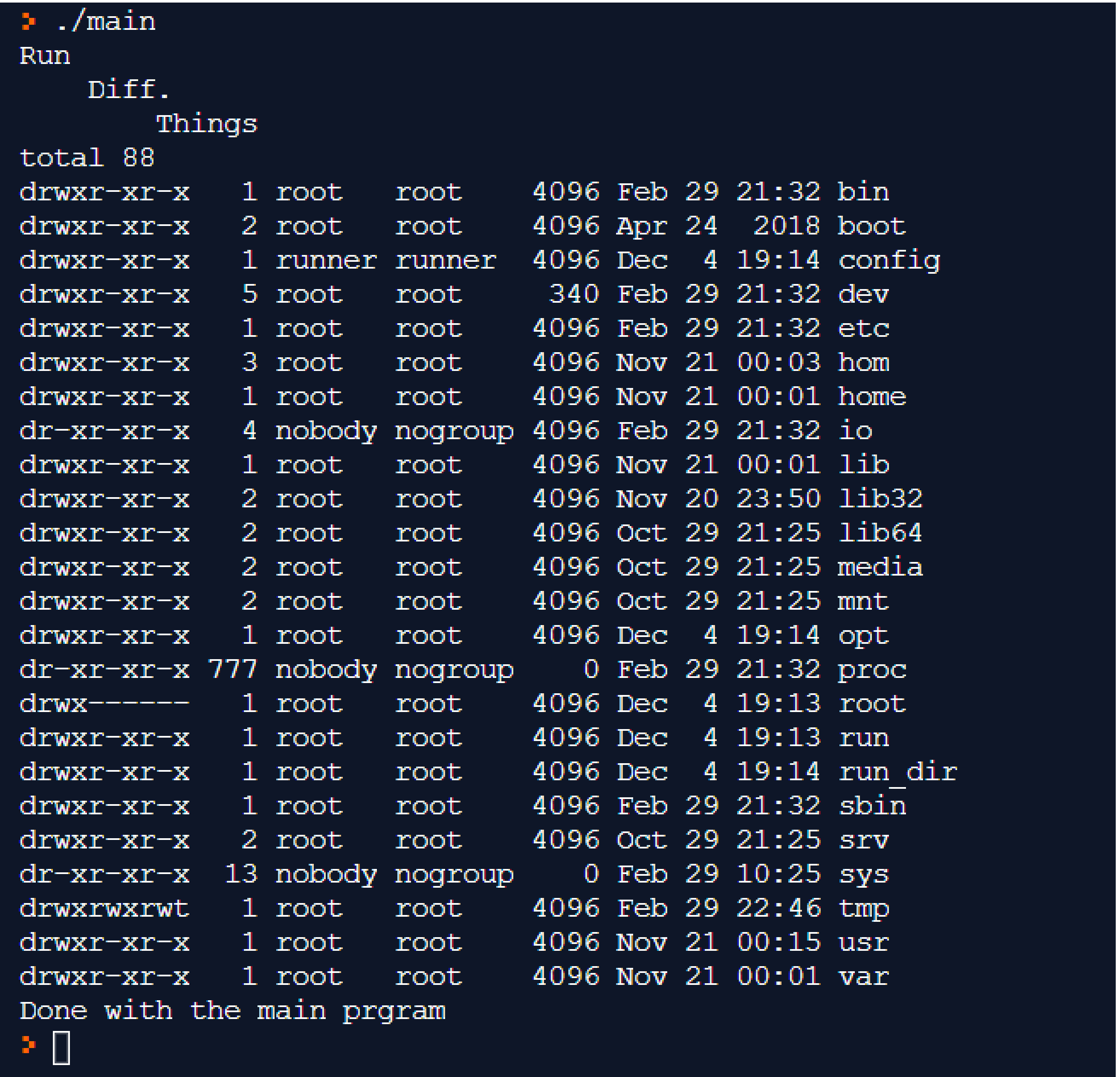
(wait, waitpid)

Most of the slides in this lecture are either from or adapted from the slides provided by Dr. Ahmad Barghash

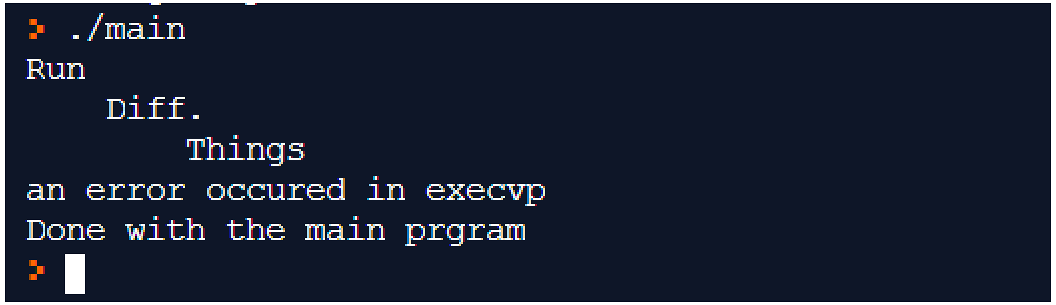
fork and exec (Create spawn function for Linux)

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/types.h>
void spawn(char* program, char** arg_list)
{
        pid_t child_pid;
        child_pid=fork();
        if(child_pid!=0)
        {
                printf("Run\n");
                printf("        Diff.\n");
                printf("                        Things\n");
        }
        else{
                execvp(program,arg_list);
                fprintf(stderr,"an error occured in execvp\n");
                abort();
        }
}

int main()
{
        char* arg_list[]={"ls","-l","/“, NULL};
        spawn(arg_list[0], arg_list);

        printf("Done with the main prgram\n");
        return 0;
}
```

Correct and false runs



```
> ./main
Run
    Diff.
        Things
total 88
drwxr-xr-x   1 root   root    4096 Feb 29 21:32 bin
drwxr-xr-x   2 root   root    4096 Apr 24  2018 boot
drwxr-xr-x   1 runner runner  4096 Dec  4 19:14 config
drwxr-xr-x   5 root   root     340 Feb 29 21:32 dev
drwxr-xr-x   1 root   root    4096 Feb 29 21:32 etc
drwxr-xr-x   3 root   root    4096 Nov 21 00:03 hom
drwxr-xr-x   1 root   root    4096 Nov 21 00:01 home
dr-xr-xr-x   4 nobody nogroup 4096 Feb 29 21:32 io
drwxr-xr-x   1 root   root    4096 Nov 21 00:01 lib
drwxr-xr-x   2 root   root    4096 Nov 20 23:50 lib32
drwxr-xr-x   2 root   root    4096 Oct 29 21:25 lib64
drwxr-xr-x   2 root   root    4096 Oct 29 21:25 media
drwxr-xr-x   2 root   root    4096 Oct 29 21:25 mnt
drwxr-xr-x   1 root   root    4096 Dec  4 19:14 opt
dr-xr-xr-x 777 nobody nogroup    0 Feb 29 21:32 proc
drwx------   1 root   root    4096 Dec  4 19:13 root
drwxr-xr-x   1 root   root    4096 Dec  4 19:13 run
drwxr-xr-x   1 root   root    4096 Dec  4 19:14 run_dir
drwxr-xr-x   1 root   root    4096 Feb 29 21:32 sbin
drwxr-xr-x   2 root   root    4096 Oct 29 21:25 srv
dr-xr-xr-x  13 nobody nogroup    0 Feb 29 10:25 sys
drwxrwxrwt   1 root   root    4096 Feb 29 22:46 tmp
drwxr-xr-x   1 root   root    4096 Nov 21 00:15 usr
drwxr-xr-x   1 root   root    4096 Nov 21 00:01 var
Done with the main prgram
>
```

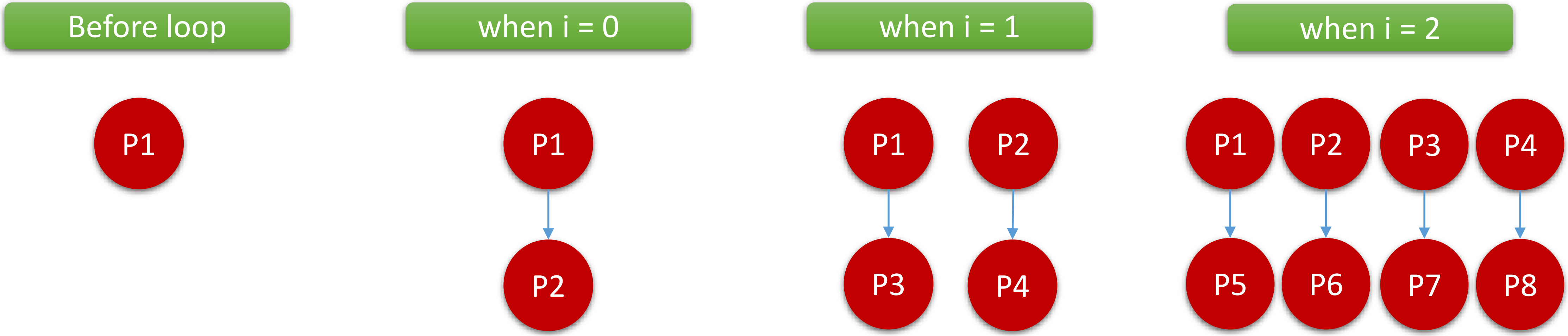


```
> ./main
Run
    Diff.
        Things
an error occured in execvp
Done with the main prgram
>
```

Example – Process Creation using fork

- How many processes will be running by the end of the following snippet of code?

```
for(int i = 0; i < 3; i++)
{
        fork();
}
```

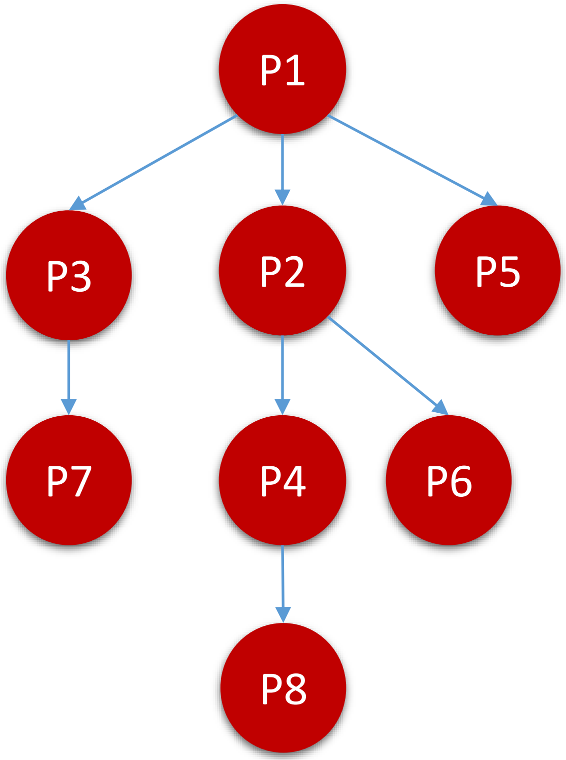


Example – Process Creation using fork

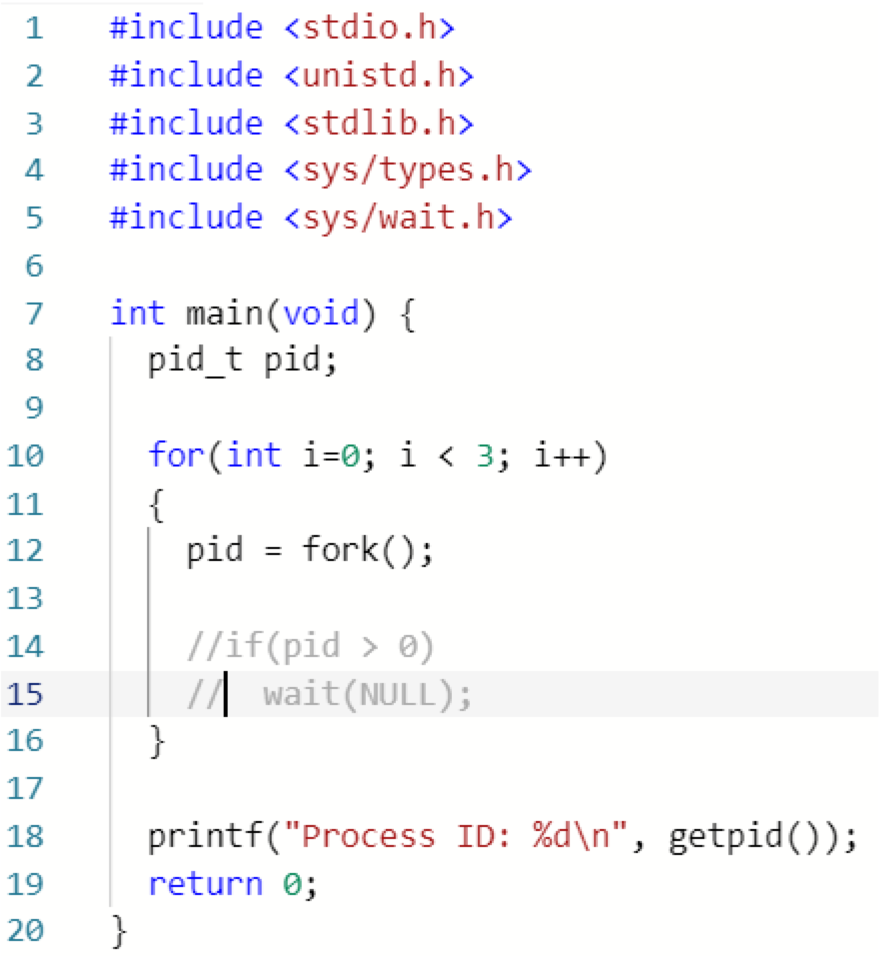
- How many processes will be running by the end of the following snippet of code?

```
for(int i = 0; i < 3; i++)
{
        fork();
}
```

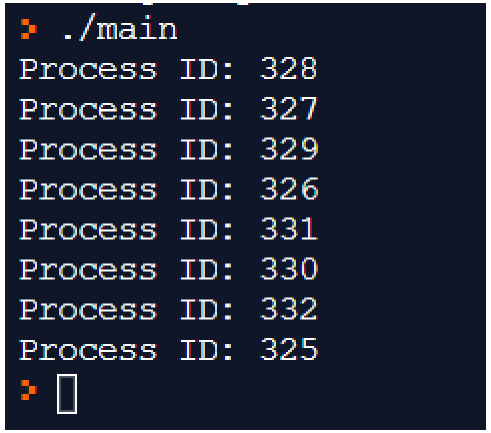
Processes Tree (parent-child relationship)



Example



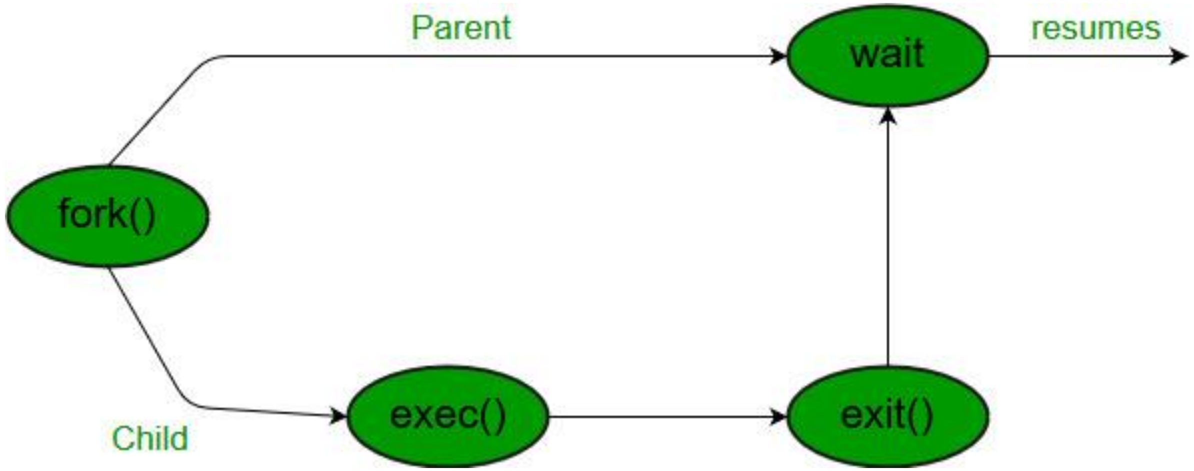
```c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/wait.h>

int main(void) {
  pid_t pid;

  for(int i=0; i < 3; i++)
  {
    pid = fork();

    //if(pid > 0)
    //  wait(NULL);
  }

  printf("Process ID: %d\n", getpid());
  return 0;
}
```



```
❯ ./main
Process ID: 328
Process ID: 327
Process ID: 329
Process ID: 326
Process ID: 331
Process ID: 330
Process ID: 332
Process ID: 325
❯
```

The wait System Calls: wait, waitpid

- A process may **wait** on another process to complete its execution.
- In most systems, a parent process can create an independently executing child process.
- The parent process may then issue a wait system call, which suspends the execution of the parent process while the child executes.
- When the child process terminates, it returns an exit status to the operating system, which is then returned to the waiting parent process.
- The parent process then resumes execution.



Processes - Orphans and Zombies

- A child process whose parent has terminated is referred to as **orphan**.
- When a child exits while its parent is not currently executing a **wait()**, a zombie emerges.
  - A zombie or a defunct process is a process that has been completed, but its entry remains in the process table due to lack of correspondence between the parent and child processes.
  - Usually, a parent process keeps a check on the status of its child processes through the wait() function. When the child process has finished, the wait function signals the parent to completely exit the process from the memory. However, if the parent fails to call the wait function for any of its children, **the parent process still shows an entry in a process table, so this process is named a zombie process.** These zombie processes might accumulate, in large numbers, on your system and affect its performance.

The wait System Calls: wait vs. waitpid

- The **wait**() system call suspends execution of the current process until one of its children terminates.
  - How a process can be terminated?
    - It calls exit().
    - It returns (an int) from the main function.
    - It receives a signal (from the OS or another process) whose default action is to terminate.
- The **waitpid**() system call suspends execution of the current process until a child specified by the “*pid*” argument has a changed state. By default, **waitpid**() waits only for terminated children, but this behavior is modifiable via the *options* argument, as described later.

The syntax of the “wait” function:
pid_t **wait** (int *status)

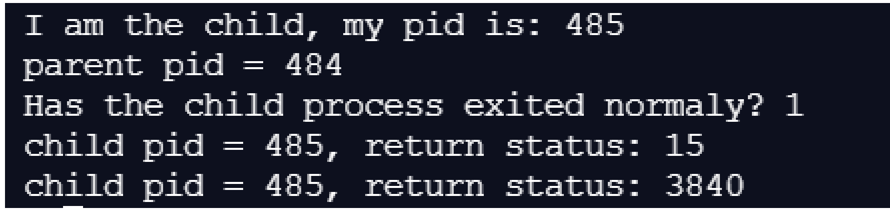
- You can store the location of the status information of the child process from the *status parameter.
- The function can return:
  - Process ID of the terminated process.
  - -1 if the process has no child processes at all!

- To return the status code of the child, you have to use the **WEXITSTATUS** macro.
- If a process has more than one child processes, then after calling wait(), the parent process has to be in wait state if no child terminates.

“wait” Example

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

int main(void) {
    pid_t pid;
    int status;
    int x = 5;
    int y = 10;

    if(fork() == 0)
    {
        printf("I am the child, my pid is: %d\n", getpid());
        return (x+y);
    }
    else
    {
        printf(“parent pid = %d\n", getpid());
        pid = wait(&status);
    }

    printf("Has the child process exited normaly? %d\n", WIFEXITED(status));

    if(WIFEXITED(status)){
        /* Child process exited normally, through `return` or `exit` */
        printf("child pid = %d, return status: %d\n", pid, WEXITSTATUS(status));
        printf("child pid = %d, return status: %d\n", pid, status);
    }
    return 0;
}
```

Output:



```
I am the child, my pid is: 485
parent pid = 484
Has the child process exited normaly? 1
child pid = 485, return status: 15
child pid = 485, return status: 3840
```

“waitpid” Syntax:

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

pid	Description
< -1	meaning wait for any child process whose process group ID is equal to the absolute value of pid.
-1	meaning wait for any child process.
0	meaning wait for any child process whose process group ID is equal to that of the calling process (i.e., child and parent in the same process group).
> 0	meaning wait for the child whose process ID is equal to the value of pid.

options	Description
0	returns when a child has terminated (i.e., similar to `wait`)
WNOHANG	Do not block if no child changed its state.
WUNTRACED	returns if a child has stopped. Status for traced children which have stopped is provided even if this option is not specified.
WCONTINUED	returns if a stopped child has been resumed.

Note:

- **wait(&status)** is equivalent to **waitpid(-1,&status,0)**

Waitpid contd. (status)

Macro: short for macroinstruction which means rule or pattern

After the call to `waitpid`, the status information stored at the location pointed to by **statusPtr** can be evaluated with the following macros:

- WIFEXITED(*statusPtr)
  evaluates to a nonzero (true) value if the specified process terminated normally.
- WEXITSTATUS(*statusPtr)
  if the specified process terminated normally, this macro evaluates the lower 8 bits of the value passed to the exit or _exit function or returned from main.
- WIFSIGNALED(*statusPtr)
  evaluates to a nonzero (true) value if the specified process terminated because of an unhandled signal.
- WTERMSIG(*statusPtr)
  if the specified process is ended by an unhandled signal, this macro evaluates to the number of that signal.
- WIFSTOPPED(*statusPtr)
  (true) value if the specified process is currently stopped but not terminated.
- WSTOPSIG(*statusPtr)
  if the specified process is currently stopped but not terminated, then this macro evaluates to the number of the signal that caused the process to stop

One example

```
#include <sys/wait.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
int main()
{
    pid_t cpid, w;
    int status;
    cpid = fork();
    if (cpid == 0) {       /* Code executed by child */
                    printf("Child PID is %ld\n", (long) getpid());
                    return 18;
     }
     else {                /* Code executed by parent */

                         w = waitpid(cpid, &status, 0);
                         if (w == -1)
                                     { perror("waitpid"); exit(EXIT_FAILURE); }
                         if (WIFEXITED(status))
                                     { printf("exited, status=%d\n", WEXITSTATUS(status)); }
                         else if (WIFSIGNALED(status))
                                     {printf("killed by signal %d\n", WTERMSIG(status)); }
                         else if (WIFSTOPPED(status))
                                     {printf("stopped by signal %d\n", WSTOPSIG(status));}
                         else if (WIFCONTINUED(status))
                                     {printf("continued\n");}
      }
}
```

```
Child PID is 3676
Exited, status=18
```

The parent process waits for the child process until it terminates normally.

perror prints a descriptive error message

Summary about fork, exec, wait

- What is the fork-exec-wait pattern?
  - A common programming pattern is to call fork followed by exec and wait.
  - 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.

It enables preventing zombies!