Process management
Examples and exercises

EPL222 – Lab2

Summary of fork()

- The syntax of the fork() system function is as follows:

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

- The fork() system function does not accept any argument
  - It returns an integer of the type pid_t (defined in library *<sys/types.h>*)
- On success, fork() returns the PID of the child process which is greater than 0
  - Inside the child process, the return value is 0
  - If fork() fails, then it returns -1

Simple pipes
(between parent and child)

- pipe() is used for passing information from one process to another
    - pipe() is unidirectional therefore, for two-way communication between processes, two pipes can be set up, one for each direction. E.g.:
    -

```
int fd[2];
pipe(fd);
fd[0]; //-> for using read end
fd[1]; //-> for using write end
```

Pipe example

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>
int main() {
     int fd[2], i = 0;
     pipe(fd);
     pid_t pid = fork();

     if(pid > 0) {
         wait(NULL);                              // wait for child to finish
         close(fd[1]);                            // no need to use the write end of pipe here so close it

         int arr[10];
         int n = read(fd[0], arr, sizeof(arr));   // n stores the total bytes read successfully

         for ( i = 0;i < n/4; i++) {
               printf("%d ", arr[i]);             // printing the array received from child process
         }
     } else if( pid == 0 ) {
         int arr[] = {1, 2, 3, 4, 5};

         close(fd[0]);                            // no need to use the read end of pipe here so close it
         write(fd[1], arr, sizeof(arr));
     } else {
         perror("error\n"); //fork()
     }
}
```

Wait system calls

```
#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);
```

- All these system calls are used to wait for state changes in a child of the calling process and obtain information about the child whose state has changed. A state change is considered to be:
  - the child terminated;
  - the child was stopped by a signal;
  - or the child was resumed by a signal.
- In the case of a terminated child, performing a wait allows the system to release the resources associated with the child; if a wait is not performed, then the terminated child remains in a "zombie" state
- If a child has already changed state, then these calls return immediately. Otherwise, they block until either a child changes state or a signal handler interrupts the call

Wait system calls

- The wait() system call suspends execution of the current process until one of its children terminates.
  - The call wait(&status) is equivalent to:

```
      waitpid(-1, &status, 0);
```

- The waitpid() system call suspends execution of the current process until a child specified by pid argument has changed state
  - By default, waitpid() waits only for terminated children, but this behaviour is modifiable via the options argument

- The waitid() system call (available since Linux 2.6.9) provides more precise control over which child state changes to wait for

Wait system calls

- If only one child process is terminated, wait() returns the process ID of the terminated child

- If more than one child processes are terminated, then wait() catches any *arbitrarily child* and returns the process ID of that child

- If a process has no child process, then wait() immediately returns “-1”

- When wait() returns it also defines an *exit status* (which tells us why the process terminated)
  - Done via a pointer passed in the call of wait()

Wait example

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

int main() {
  pid_t cpid;
  if (fork()== 0)
     exit(0);
  else
     cpid = wait(NULL);

  printf("Parent pid = %d\n", getpid());
  printf("Child pid = %d\n", cpid);
  return 0;
}
```

Checking the Status of child

- To learn about the exit status of a program we can use the macros from <sys/wait.h> which check the termination status and return the exit status
  - WIFEXITED(status) returns true if the child terminated normally, that is, by calling exit() or _exit(), or by returning from main()
  - WEXITSTATUS(status) returns the exit status of the child. This consists of the least significant 8 bits of the status argument that the child specified in a call to exit() or _exit() or as the argument for a return statement in main()
    - This macro should only be employed if WIFEXITED returned true

Checking the Status of child Example

```
#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();

   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(2);                   //sleep for 2 seconds
      exit(12);                   //exit with status 12

   } else if (c_pid > 0){         //parent
      pid = wait(&status);        //waiting for child to terminate
      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;
}
```

waitpid() options

The value of *options* is the bitwise OR of zero or more of the following constants:

Tag	Description
WNOHANG	return immediately if no child has exited.
WUNTRACED	also return if a child has stopped (but not traced via ptrace()). Status for traced children which have stopped is provided even if this option is not specified.
WCONTINUED	(Since Linux 2.6.10) also return if a stopped child has been resumed by delivery of SIGCONT.

wait() and waitpid() Status Information

Tag	Description
WIFEXITED(status)	returns true if the child terminated normally, that is, by calling exit() or _exit(), or by returning from main().
WEXITSTATUS(status)	returns the exit status of the child. This consists of the least significant 16-8 bits of the status argument that the child specified in a call to exit() or _exit() or as the argument for a return statement in main(). This macro should only be employed if WIFEXITED returned true.
WIFSIGNALED(status)	returns true if the child process was terminated by a signal.
WTERMSIG(status)	returns the number of the signal that caused the child process to terminate. This macro should only be employed if WIFSIGNALED returned true.
WCOREDUMP(status)	returns true if the child produced a core dump. This macro should only be employed if WIFSIGNALED returned true. This macro is not specified in POSIX.1-2001 and is not available on some Unix implementations (e.g., AIX, SunOS). Only use this enclosed in **#ifdef WCOREDUMP ... #endif**.
WIFSTOPPED(status)	returns true if the child process was stopped by delivery of a signal; this is only possible if the call was done using WUNTRACED or when the child is being traced.
WSTOPSIG(status)	returns the number of the signal which caused the child to stop. This macro should only be employed if WIFSTOPPED returned true.
WIFCONTINUED(status)	(Since Linux 2.6.10) returns true if the child process was resumed by delivery of SIGCONT.

Exercise 1

- Write a program to create 4 processes: parent process and its child process which perform various tasks on an array of 10 numbers:
  - Parent process count the frequency of a number
  - 1st child sort the array
  - 2nd child find total even number(s) in the array
  - 3rd child calculate the sum of even numbers in the array

Exercise 2

- Write a C program, fibchild.c, which computes and prints Fibonacci numbers in child processes
  - The program runs in a loop and creates a child process, which computes the current Fibonacci number and prints it.
  - The parent waits for the child finish and then continues executing.
  - Take the number of Fibonacci numbers to be printed as input.
- Fibonacci number are defined as:
  - $fib(n) = fib(n-1) + fib(n-2), n>2$
  - $fib(1) = fib(2) = 1$