Unix Processes

Name	Demo 2
Review	☐
Status	Not started

Task 1:

commands to list these management meta info

```
"ps" for process status - current processes
"ps -a" for list more process - run by other user/terminal
"ps -l" prints longer - ppid/pid
```

Task 2: getpid/getppid/getuid()

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
// getpid() - long integer "%ld"
// getppid()
// getuid()
void main(void){
        printf("Process ID : %ld\n", (long)getpid());
        printf("Parent Process ID: %ld\n", (long) getppid());
        printf("Owner user ID: %ld\n", (long)getuid());
}

// operation: gcc -o demo2-2 getpidDemo.c
// operation: ./demo2-2

// Result
// Process ID : 26991
// Parent Process ID: 25575
// Owner user ID: 51395
```

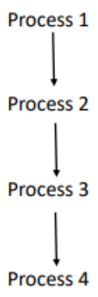
Task 3: fork()

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
void main(void){
        int ret_from_fork, mypid;
        mypid = getpid();
        printf("Before: my pid is %d\n", mypid);

        ret_from_fork = fork(); // child p create

        sleep(1);
        printf("After: my pid is %d, fork() said %d\n", getpid(), ret_from_fork);

}

// return:
// Before: my pid is 27480
// After: my pid is 27480, fork() said 27481 (parent process)
// After: my pid is 27481, fork() said 0 (child process, no child process)

// child process and parent process execue at the same time
```

Task 4: distinguish b/t child process and parent process

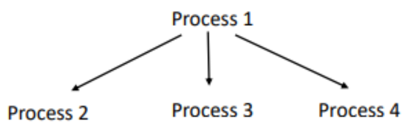
```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

void main(void){
        long ret_from_fork;
        printf("Before: my pid is %d\n", getpid());
        ret_from_fork = fork();

        if (ret_from_fork ==0){
                fprintf(stderr, "I am the child, ID = %ld\n", (long)getpid());
                printf("Child says(PID = %ld): Hello World!\n", (long)getpid());
        }
        else if (ret_from_fork > 0){
                fprintf(stderr, "I am the parent, ID = %ld\n", (long)getpid());
        }
}

// operation: gcc -o demo2-4 forkDemo1.c
// operation: ./demo2-4

// Result
// Before: my pid is 28514
// I am the parent, ID = 28514
// I am the child, ID = 28515
// Child says(PID = 28515): Hello World!

// failure of fork(): return -1
```

Task 5: pid_t



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

void main(void){
        int i, n = 4;
        pid_t childpid;
        for (i = 1; i < n; ++i)
                if (childpid = fork())
                        break; // parent breaks out; child continues
        fprintf(stdout, "This is process %ld with parent %ld, i = %d\n", (long)$

}
// operation
// gcc -o demo2-5 forkDemo2.c
// ./demo2-5

// Result
// This is process 30020 with parent 29580, i = 1
// This is process 30021 with parent 30020, i = 2
// This is process 30022 with parent 30021, i = 3
// This is process 30023 with parent 1, i = 4
```

Task 6: forking fan of process



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

void main(void){
        int i, n = 4;
        pid_t childpid;
        for (i = 1; i < n; ++i)
                if ((childpid = fork()) <= 0)
                        break; // child and error break out; parent continues
        fprintf(stdout, "This is process %ld with parent %ld, i = %d\n", (long)$

}

// operation
// gcc -o demo2-6 forkDemo3.c
// ./demo2-6

// Result
// This is process 30531 with parent 30530, i = 1
// This is process 30530 with parent 29580, i = 4
// This is process 30532 with parent 30530, i = 2
// This is process 30533 with parent 30530, i = 3
```

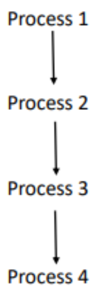
// straight line: child continues, parent break out

// fan: child & error break out, parent continues

Notes

```
wait system call
waitpid
pid_t wait(int *stat);
// cause caller process to pause til child terminates
// orphan process: parent terminates without waiting for its child
// zombie process: child process will remain
// wait returns because child terminated, return alue >0 & =childpid
// otherwise: wait return -1, errno
  // errno = ECHILD - no unwaited-for child
  // errno = EINTR - call interrupted
```

Task 7: wait() example



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

```
#include <sys/wait.h>
#include <unistd.h>
#include <errno.h>

void main(void){
        int i, n = 4, status;
        pid_t childpid, waitreturn;
        for (i = 1; i < n; ++i)
                if (childpid = fork()) break; // parent break
        while (childpid != (waitreturn = wait(&status)))
                if ((waitreturn == -1)&& (errno != EINTR))
                        break;
        fprintf(stdout, "I am process %ld, my parent is %ld\n", (long)getpid(),$


}
// operation
// gcc -o demo2-7 waitDemo.c
// ./demo2-7

// result
// I am process 31990, my parent is 31989
// I am process 31989, my parent is 31988
// I am process 31988, my parent is 31987
// I am process 31987, my parent is 29580
```

Task 8: exec() on creating a process to run “ls -l”

```


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

void main(void){
        int status;
        pid_t childpid;
        if((childpid = fork())== -1){
                perror("Error in the fork");
                exit(1);
        } else if (childpid == 0){
                //child mode
                if (execl("/usr/bin/ls", "ls", "-l", NULL) < 0){
                        perror("Exec of ls failed");
                        exit(1);
                }
        } else if (childpid != wait(&status))
                // parent mode
                perror("A signal occured before child exited");
        exit(0);
}

// operation
// gcc -o demo2-8 execDemo.c
// ./demo2-8

// result
// total 224
// ...
```

Task 9: execvp()

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

void main(void){
        int status;
        pid_t childpid;
        char *argv[3];
        argv[0] = "ls";
        argv[1] = "-l";
        argv[2] = 0; // NULL pointer

        if ((childpid = fork()) == -1) {
                perror("Error in the fork");
                exit(1);
        } else if (childpid == 0) {
                // child code
                if (execvp("ls", argv) < 0) {
                        perror("Exec of ls failed");
                        exit(1);
                }
        } else if (childpid != wait(&status))
                // parent code
                perror("A sinal occured before child exited");
        exit(0);
}

// operation
// gcc -o demo2-9 execDemo1.c
// ./demo2-9

// result
// ...
```

Notes

```
Six variation

int execl(char const *path, char cont *arg0, …);
int execle(char const *path, char const *arg0, …, char const *envp[]);
int execlp(char const *file, char const *arg0, …);
int execv(char const *path, char const *argv[]);
int execve(char const *path, char const *argv[], char const *envp[]);
int execvp(char const *file, char const *argv[]);

// path : C string (a NULL ended array of char) representing the full path of the executable file
// file : C string representing only the file name of the executable file, the path is searched from the PATH environment
// arg0 : should be the same as file name, the 0th argument passed to main.
// … : a list of char const * pointers, each points to a C string, as arguments passed to main; the list must end with a NULL.
// argv : an array of char const * pointers, each points to a C string, as arguments passed to main.
// envp : an array of char const *, each points to a C string of format "environment_variable_name=value".

// l: arguments are passed to main as a list of char const * pointers, each pointing to a C string; the list must end with a NULL.
// v: arguments are passed to main as an array of char const * pointers, each pointing to a C string.
// e: an array of environment variable "environment_variable_name=value" pairs are explicitly passed to the new process image.
// p: use the PATH environment to search for the executable file.
```

Task 10

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

void main(void){
        printf( "Old image: pid = %d\n", getpid());
        execlp("./newimage", "newimage", NULL);
        printf("Old image: hello\n");
}

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

```
void main(void) {
        printf("New image: pid = %d\n", getpid());
}

// operation
// gcc -o newimage newimage.c
// gcc -o oldimage oldimage.c
// ./oldimage

// Result
// Old image: pid = 2581
// New image: pid = 2581
```

Notes

```
// Background process
// operation: touch test.txt
// operation: tail -f test.txt
// result: indefinite loop

// operation: tail -f test.txt &
// result: [5] 2916


// daemons
// background process that normally runs indefinitely (have an infinite loop).
```

Task 11: daemons

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <syslog.h>
#include <string.h>
#include <unistd.h>

#define MAX_I 100
void main(void){
        pid_t pid, sid;
        FILE* p_output;
        int i=0;

        pid = fork();
        if (pid < 0) exit(EXIT_FAILURE);
        if (pid > 0) exit(EXIT_SUCCESS);

        umask(0);
        sid = setsid();
        if(sid < 0) exit(EXIT_FAILURE);
        if ((chdir(".")) < 0) exit(EXIT_FAILURE);

        close(STDIN_FILENO);
        close(STDOUT_FILENO);
        close(STDERR_FILENO);

        while(1){
                if ((p_output = fopen("daemon_output.txt", "a")) != NULL){
                        fprintf(p_output, "%d\n", i++);
                        i %= MAX_I;
                        fclose(p_output);
                }
                sleep(3);
        }
        exit(EXIT_SUCCESS);
}
```