// **Program-1: Get process ID and Parent process ID**

#include <stdio.h>

#include <sys/types.h> // - data types

#include <unistd.h> // explain the head file

int main (void) {

pid\_t getpid();

printf("Child's PID = %d\n", getpid());

pid\_t getppid();

printf("Parent's PID = %d\n", getppid());

return 0;

}

**// Output:**

~/Week-6$ ./a.out

Child's PID = 527

Parent's PID = 302

~/Week-6$ ps

PID TTY TIME CMD

302 pts/0 00:00:00 bash

528 pts/0 00:00:00 ps

要解释为什么ppid 不变，但是pid会变

**// Program-2: fork.c**

#include<sys/types.h>

#include<unistd.h>

extern int errno;

int main(void) {

int pid, status;

pid = fork();

if (pid == -1) {

printf("Fork failed.\n");

return (1);

}

/\* Child process \*/

if (pid == 0) {

pid\_t getpid();

printf("\n CHILD: Child here with PID = %d.\n", getpid());

pid\_t getppid();

printf("CHILD: My Mom has PID = %d.\n", getppid());

return (0);

}

/\* Parent process \*/

pid\_t wait(int \*status);

wait(&status);

printf("\n PARENT: Mom here with PID = %d.\n", getpid());

printf("PARENT: Well done, child!\n\n");

return (0);

}

**// Output**

./a.out -w fork.c -o fork

CHILD: Child here with PID = 891.

CHILD: My Mom has PID = 890.

PARENT: Mom here with PID = 890.

PARENT: Well done, child!

3. **// Program-3: Program shown in the following session creates a child process that //overwrites itself.**

#include<stdio.h>

#include<sys/types.h>

#include<unistd.h>

extern int errno;

int main(void) {

int pid, status, e;

pid = fork();

if (pid == -1) {

printf("Fork failed.\n");

return (1);

}

/\* Child process \*/

if (pid == 0) {

printf("exec1 failed.\n");

return (1);

}

/\* Parent process \*/

printf("Child PID = %d.\n", (int) pid);

pid\_t wait(int \*status);

pid = wait(&status);

printf("Child with PID %d has terminated.\n", (int) pid);

return (0);

}

// Output:

~/Week-6$ !v

vi fork\_exec\_1.c

~/Week-6$ cc -w fork\_exec\_1.c -o forke1

~/Week-6$ ./forke1

Child PID = 1441.

exec1 failed.

Child with PID 1441 has terminated.

4. **// Program-4: Create a zombie process.** A process whose parent is not waiting when it terminates cannot report its termination status to its parent and is called a zombie process \*/

#include<stdio.h>

#include<sys/types.h>

#include<unistd.h>

int main(void) {

int pid;

pid = fork();

if (pid == -1) {

printf("Fork failed.\n");

return(1);

}

/\* Child process \*/

if (pid == 0) {

printf("Child's PID = %d\n", getpid());

return (0);

}

/\* Parent process \*/

printf("Parent's PID = %d\n", getpid());

sleep(10);

/\* Parent process does not wait for the child \*/

/\* process and child become a zombie process \*/

return (0);

}

**// Output**

~/Week-6$ !.

./a.out

Parent's PID = 2220

Child's PID = 2221

~/Week-6$ !.

./a.out

Parent's PID = 2231

Child's PID = 2232

5. **// Program-5: signals.c**

// Takes programmer defined actions for SIGINT (signal number 2) and SIGALARM (signal number 14). For all other signals, the process takes the system-defined default action.

#include <sys/signal.h>

#include <stdio.h>

#define TRUE 1

void nicetry(void);

void onalarm(int);

int main(void) {

signal(SIGHUP, SIG\_IGN);

signal(SIGINT, nicetry);

signal(SIGALRM, onalarm);

alarm(10);

while (TRUE) {

printf("Waiting for alarm.\n");

sleep(9);

}

}

void nicetry(void) {

signal(SIGINT, nicetry);

printf("Nice try! Sorry you cannot terminate me like this.\n");

}

void onalarm(int signal) {

printf("Caught signal number %d. Going back to work.\n",signal);

}

6. // **Program-6: killer.c**

#include <stdio.h>

#include <sys/types.h>

int main(int argc, char \*argv[]) {

pid\_t pid;

int signal;

if (argc != 3) {

printf("Inappropriate number of command line arguments. \n");

return(0);

}

pid = atoi(argv[1]);

signal = atoi(argv[2]);

if (kill(pid, signal) == -1) {

printf("Kill failed.\n");

return(1);

}

return(0);

}

// Output:

// PID TTY TIME CMD

302 pts/0 00:00:00 bash

3097 pts/0 00:00:00 a.out

3591 pts/0 00:00:00 ps

~/Week-6$ ./killer 3591 9

7. // Program-7: dup.c

// Program takes a file as a command line argument, opens the file for writing, and saves the file descriptor for the file in fd.

#include <fcntl.h>

#include <stdio.h>

#include <sys/stat.h>

int main(int argc, char \*argv[]) {

int fd;

/\* Open file \*/

if (argc == 1) {

printf("No file specified as coomand line argument. \n");

return(1);

}

if ((fd = open(argv[1], O\_WRONLY | O\_CREAT | O\_TRUNC, S\_IREAD|S\_IWRITE)) == -1) {

printf("File opening.\n");

return(1);

}

/\* Close standard output \*/

close (1);

/\* Duplicate fd into file descriptor 1, i.e. stdout \*/

if (dup(fd) == -1) {

printf("Duplicating file descriptor");

return(1);

}

/\* Close fd in order to release the extra slot in the PPFDT \*/

if (close (fd) == -1) {

printf("File closing.\n");

return(1);

}

/\* Stdout redirected to the file passed as command line argument \*/

printf("Hello, world!\n");

return(0);

}

~/Week-6$ cc -w dup.c -o dup

~/Week-6$ ./dup foo1

~/Week-6$ vi foo1

~/Week-6$ cat foo1

Hello, world!

~/Week-6$ //Program-7: dup2

#include <fcntl.h>

#include <stdio.h>

#include <sys/stat.h>

int main(int argc, char \*argv[]) {

int fd;

/\* Open file \*/

if (argc == 1) {

printf("No file specified as coomand line argument. \n");

return(1);

}

if ((fd = open(argv[1], O\_WRONLY | O\_CREAT | O\_TRUNC, S\_IREAD|S\_IWRITE)) == -1) {

printf("File opening.\n");

return(1);

}

/\* Duplicate fd into file descriptor 1, i.e. stdout \*/

if (dup2(fd,1) == -1) {

printf("Duplicating file descriptor");

return(1);

}

/\* Close fd in order to release the extra slot in the PPFDT \*/

if (close (fd) == -1) {

printf("File closing.\n");

return(1);

}

/\* Stdout redirected to the file passed as command line argument \*/

printf("Hello, world!\n");

~/Week-6$ !c

cc dup2.c -o dup2

~/Week-6$ ./dup2 foo

~/Week-6$ vi foo

~/Week-6$ cat foo

Hello, world!