Inter-Process Communication
(using the kernel API and the Standard C Library)

---

1. Consider the following program that implements a "pipe" between a process and its child. Compile it and run it. Read the code carefully and understand it.

```
#include <sys/wait.h>
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#define LINESIZE  256

int main(int argc, char* argv[]) {
  int   fd[2];
  pid_t pid;
  if (pipe(fd) == -1) { perror("pipe"); exit(EXIT_FAILURE); }
  if ((pid = fork()) == -1) { perror("fork"); exit(EXIT_FAILURE); }
  if (pid > 0) { /* parent */
    char  line[LINESIZE] = "Lorem ipsum dolor sit amet ...";
    close(fd[0]);
    write(fd[1], line, strlen(line));
    close(fd[1]);
    if ( wait(NULL) == -1) { perror("wait"); exit(EXIT_FAILURE); }
    exit(EXIT_SUCCESS);
  } else { /* child */
    char  line[LINESIZE];
    close(fd[1]);
    int nbytes = read(fd[0], line, LINESIZE);
    write(STDOUT_FILENO, line, nbytes);
    close(fd[0]);
    exit(EXIT_SUCCESS);
  }
}
```

Change the program so that, instead of the current message exchange, the parent process opens a text file, whose name is provided in the command line, reads its contents and sends it through the pipe to the child process. The child process, on the other hand, receives the contents of the file and prints it to the `stdout`. Compile and execute your program with a large enough file, e.g., your source code file.

2. The following program allows a process and its child to communicate via a pair of "sockets". Unlike "pipes", "sockets" allow bidirectional communication between connected processes. Compile and run the program. Read the following code and understand it.

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

int main(int argc, char* argv[]) {
  char  buf[1024];
  int   sockets[2], retv;
  retv = socketpair(AF_UNIX, SOCK_STREAM, 0, sockets);
  if (retv == -1) { perror("socketpair"); exit(EXIT_FAILURE); }
  retv = fork();
  if (retv == -1) { perror("fork"); exit(EXIT_FAILURE); }
  if (retv > 0) { /* parent */
    char  string1[] = "In every walk with nature...";
    close(sockets[1]);
    write(sockets[0], string1, sizeof(string1));
    read(sockets[0], buf, sizeof(buf));
    printf("message from %d-->%s\n", getpid(), buf);
    close(sockets[0]);
    retv = wait(NULL);
    if (retv == -1) { perror("wait"); exit(EXIT_FAILURE); }
  } else { /* child */
    char  string2[] = "...one receives far more than he seeks.";
    close(sockets[0]);
    read(sockets[1], buf, sizeof(buf));
    printf("message from %d-->%s\n", getppid(), buf);
    write(sockets[1], string2, sizeof(string2));
    close(sockets[1]);
    exit(EXIT_SUCCESS);
  }
}
```

Modify the program so that the parent process opens a file, whose name is given in the command line, and transfers its contents to the child process. The latter should receive the contents of the file via a socket and convert all characters to uppercase before sending them back to the parent process. The parent process then prints the processed text to the `stdout`.

3. This example shows the use of signals to control the execution of processes. In function `main`, the function `signal` is used to map signals given by their identifiers, e.g., `SIGUSR1` e `SIGUSR2`, to the functions that should be executed when the process receives one such signal. These functions are called "handlers". To test this example, compile and run the program. Then create a new terminal and use the command `kill` to send a signal `SIGUSR1` to the process using its PID: `kill -SIGUSR1 PID`.

```
#include <errno.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

static void handler1() { printf("caught SIGUSR1\n"); }
static void handler2() { printf("caught SIGUSR2\n"); }

int main(int argc, char* argv[]) {
  printf("my PID is %d\n", getpid());
  if (signal(SIGUSR1, handler1) == SIG_ERR) {
    fprintf(stderr, "Can't catch SIGUSR1: %s", strerror(errno));
    exit(EXIT_FAILURE);
  }
  if (signal(SIGUSR2, handler2) == SIG_ERR) {
    fprintf(stderr, "Can't catch SIGUSR2: %s", strerror(errno));
    exit(EXIT_FAILURE);
  }
  /* stick around ... */
  for ( ; ; )
    pause();
}
```

4. Change the previous code so that it suports also the following signals: `SIGTSTP` (generated by the keyboard when you do `CTRL-Z`), `SIGINT` (generated by the keyboard when you do `CTRL-C`) and `SIGHUP` (sent by the terminal when the user logs out). For each case, an appropriate message should be printed that allows the identification of the signal received. Can you also support signal `SIGKILL`?

5. Consider the following program that runs a cycle that can only be stopped via an external signal, say this case, `SIGINT`. Re-write it by programming the handler for `SIGINT` such that it allows the program to leave the cycle and end by executing `exit`.

```
#include <errno.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

static int flag = 1;

int main(int argc, char* argv[]) {
  printf("my PID is %d\n", getpid());
  /* enter potentially infinite loop ... */
  while(flag)
    pause();
  exit(EXIT_SUCCESS);
}
```

6. The following example shows how signals can be used to re-configure processes on-the-fly, i.e., without having to terminate and restart them. This is extremely useful, e.g., in the case of servers that should be 100% available.

```
#include <signal.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/time.h>
#include <sys/errno.h>

static int param;  /* program parameter */

void read_parameter() {
  FILE *fp = fopen(".config", "r");
  fscanf(fp, "param: %d\n", &param);
  fclose(fp);
}

void write_parameter() {
  printf("param: %d\n", param);
```

```
}

void handler (int signum) {
  read_parameter();
  write_parameter();
}

int main (int argc, char* argv[]) {
  if (signal(SIGHUP, handler) == SIG_ERR) {
    fprintf(stderr, "Can't catch SIGHUP: %s", strerror(errno));
    exit(EXIT_FAILURE);
  }
  printf("my PID is %d\n", getpid());
  read_parameter();
  printf("waiting...");
  for ( ; ; )
     pause();
}
```

To test the program, save it to a file `updatable.c` and execute the following commands:

```
$ cat > .config
param: 263
^D
$ gcc updatable.c -o updatable
$ ./updatable &
my PID is 36595
waiting...
$ kill -HUP 36595
param value is: 263
$ emacs .config (change param to 321)
$ kill -HUP 36595
param value is: 321
$
```

Did you understand what happened? The code for signal `HUP` is `1` so, `kill -HUP 36595` could also have been writen as `kill -1 36595`. Also note that it is not mandatory to use signal `SIGHUP` to make the exercise work. Any signal that can be captured by the process (via `signal`) will do.

7. Programs `p1.c` and `p2.c` communicate via a "named pipe"created with the system call `mkfifo`. Using this type of pipe you can make any two processes talk with each other - they do not have to be parent and child. Here is `p1.c`:

```
/* complete what is missing here ... */
#define BUF_SIZE 128

int main(int argc, char* argv[]) {
    char* myfifo = "/tmp/myfifo";
    int rv = mkfifo(myfifo, 0666);
    if (rv == -1) { perror("mkfifo"); exit(EXIT_FAILURE); }
    int fd = open(myfifo, O_WRONLY);
    if (fd == -1) { perror("open"); exit(EXIT_FAILURE); }
    while (1) {
        char text[BUF_SIZE];
        fgets(text, BUF_SIZE, stdin);
        write(fd, text, strlen(text)+1);
    }
    close(fd);
    exit(EXIT_SUCCESS);
}
```

and here is `p2.c`:

```
/* complete what is missing here ... */
#define BUF_SIZE 128

int main(int argc, char* argv[]) {
    char* myfifo = "/tmp/myfifo";
    int fd = open(myfifo,O_RDONLY);
    if (fd == -1) { perror("open"); exit(EXIT_FAILURE); }
    while (1) {
        char text[BUF_SIZE];
        read(fd, text, BUF_SIZE);
        printf("%s", text);
    }
    close(fd);
    exit(EXIT_SUCCESS);
}
```

Read the code carefully and then compile and execute each in a different terminal or Terminal tab as follows:

```
$ gcc -Wall p1.c -o p1
$ ./p1
```

and:

```
$ gcc -Wall p2.c -o p2
$ ./p2
```

Now `p1` is waiting for you to write something (and press the RETURN key). Do it and see what happens.

8. The following program exemplifies how to connect the `stdout` of command `cmd1` to the `stdin` of command `cmd2`, using a pipe. The system call used to perform this mapping is `dup2`. Analyse the code, complete it, compile it and execute it.

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

#define READ_END  0
#define WRITE_END 1

char* cmd1[] = {"ls", "-l", NULL};
char* cmd2[] = {"wc", "-l", NULL};

int main (int argc, char* argv[]) {
  int fd[2];
  pid_t pid;

  if (pipe(fd) < 0) {
    /* pipe error */
  }

  if ((pid = fork()) < 0) {
    /* fork error */
  }

  if (pid > 0) {
    close(fd[READ_END]);
    dup2(fd[WRITE_END], STDOUT_FILENO);  /* stdout to pipe */
    close(fd[WRITE_END]);
    /* parent writes to the pipe */
    if (execvp(cmd1[0], cmd1) < 0) {
      /* exec error */
    }
  } else {
    close(fd[WRITE_END]);
    dup2(fd[READ_END], STDIN_FILENO);  /* stdin from pipe */
```

```
    close(fd[READ_END]);
    if (execvp(cmd2[0], cmd2) < 0) {
      /* exec error */
    }
  }
}
```

Did you understand how the connection between the processes is made? Write a more generic program `mypipe` that receives a string with two commands connected by a pipe symbol and executes them so that the `stdout` of the first is connected to the `stdin` of the second. Your program should run as follows:

```
$ cat > words.txt
earth
water
air
fire
aether
^D
$ gcc -Wall mypipe.c -o mypipe
$ ./mypipe "cat words.txt | xargs"
earth water air fire aether
```

Re-write the new program using a "named pipe"as exemplified in the previous exercise. Suggestion: search for information about using the system calls `dup2` and `mkfifo`.