L:EIC / SO2122:
Inter-Process Communication
(using the Kernel API and the Standard C Library)

Q1. Consider the following program that implements a "pipe" between a process and its child. Compile it and run it. Read the code carefully and try to understand how it works.

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

#define READ_END  0
#define WRITE_END 1

#define LINESIZE  256

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

  if (pipe(fd) < 0) {
    perror("pipe error");
    exit(EXIT_FAILURE);
  }

  if ((pid = fork()) < 0) {
    perror("fork error");
    exit(EXIT_FAILURE);
  }
  else if (pid > 0) {
```

```
    /* parent */
    close(fd[READ_END]);
    printf("Parent process with pid %d\n", getpid());
    printf("Messaging the child process (pid %d):\n", pid);
    snprintf(line, LINESIZE, "Hello! I'm your parent pid %d!\n", getpid());
    if ((nbytes = write(fd[WRITE_END], line, strlen(line))) < 0) {
      fprintf(stderr, "Unable to write to pipe: %s\n", strerror(errno));
    }
    close(fd[WRITE_END]);
    /* wait for child and exit */
    if ( waitpid(pid, NULL, 0) < 0) {
      fprintf(stderr, "Cannot wait for child: %s\n", strerror(errno));
    }
    exit(EXIT_SUCCESS);
  }
  else {
    /* child */
    close(fd[WRITE_END]);
    printf("Child process with pid %d\n", getpid());
    printf("Receiving message from parent (pid %d):\n", getppid());
    if ((nbytes = read(fd[READ_END], line, LINESIZE)) < 0 ) {
      fprintf(stderr, "Unable to read from pipe: %s\n", strerror(errno));
    }
    close(fd[READ_END]);
    /* write message from parent */
    write(STDOUT_FILENO, line, nbytes);
    /* exit gracefully */
    exit(EXIT_SUCCESS);
  }
}
```

Now, change the program so that, instead of the messages above, the parent process opens a text file (whose name is given in the command line), reads the text in it and passes it through the "pipe" to the child process. The latter should receive the text from the parent process and print it in "stdout". Compile and execute your program with a large enough text file, e.g., the file with the source code.

Q2. The following example shows how to connect the "stdout" of a command `cmd1` to the "stdin" of command `cmd2`, using a "pipe". The kernel function that allows this type of mapping is `dup2`. Check the system manual pages to see how it works. Read 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 */
    }
  }
}
```

Note that some times it is possible that the result of the full command is not presented in sync with the "shell prompt", i.e., the "shell prompt" appears first and shortly afterwards the result is printed in the terminal. How do you explain this taking into consideration the way processes are executed in Unix/Linux?

Q3. Generalize the previous code to produce a new command `tube` that receives a string with any two "shell" commands united by a "pipe" (e.g., `tube "ls -l | wc -l"`). The program them automatically produces the "arrays" `cmd1` and `cmd2` and executes the full command as above. How would you generalize the program to allow the execution of "pipes" involving 3 commands? And involving `n` commands?

Q4. The following program uses a communication mechanism between a process and its child called "sockets". Unlike "pipes", "sockets" allow bidirectional communication. Compile and execute the program. Read the code and try to understand what is going on.

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

#define CHANNEL0 0
#define CHANNEL1 1

#define DATA0 "In every walk with nature..."
#define DATA1 "...one receives far more than he seeks."
/* by John Muir */

int main(int argc, char* argv[]) {
  int   sockets[2];
  char  buf[1024];
  pid_t pid;

  if (socketpair(AF_UNIX, SOCK_STREAM, 0, sockets) < 0) {
    perror("opening stream socket pair");
    exit(1);
  }

  if ((pid = fork()) < 0) {
    perror("fork");
    return EXIT_FAILURE;
  }
  else if (pid == 0) {
    /* this is the child */
    close(sockets[CHANNEL0]);
    if (read(sockets[CHANNEL1], buf, sizeof(buf)) < 0)
      perror("reading stream message");
    printf("message from %d-->%s\n", getppid(), buf);
```

```
    if (write(sockets[CHANNEL1], DATA1, sizeof(DATA1)) < 0)
      perror("writing stream message");
    close(sockets[CHANNEL1]);
    /* leave gracefully */
    return EXIT_SUCCESS;
  }
  else {
    /* this is the parent */
    close(sockets[CHANNEL1]);
    if (write(sockets[CHANNEL0], DATA0, sizeof(DATA0)) < 0)
      perror("writing stream message");
    if (read(sockets[CHANNEL0], buf, sizeof(buf)) < 0)
      perror("reading stream message");
    printf("message from %d-->%s\n", pid, buf);
    close(sockets[CHANNEL0]);
    /* wait for child and exit */
    if (waitpid(pid, NULL, 0) < 0) {
      perror("did not catch child exiting");
      return EXIT_FAILURE;
    }
    return EXIT_SUCCESS;
  }
}
```

Change the program so that the parent process opens a text file (whose name is given in the command line) and transfers the content to the child process. The child process, in turn, upon receiving the text, must transform it in to uppercase characters (when appropriate) and send it back to the parent process. Finally, the latter should print the modified text to the "stdout".

Q5. The following program demonstrates the use of a segment of memory shared between processes. The function `mmap` is used to create the segment whose pointer is shared by the various processes created by the calls to `fork` that follow. The example uses the memory segment to save a matrix of integer values initialized from an input file (`infile`). The processes calculate in parallel the number of of entries in the matrix greater than a threshold value (`threshold`), also provided as input to the program. Every process sweeps only a few lines of the matrix and saves the number of values found in a vector with partials per line (also shared). In the end, the parent process traverses the vector with the parcial values to compute the total for the matrix. This type of solution can significantly improve the speed of the operation, in the limit running on T(1)/`nprocs` time units, where T(1) is the execution time of the program with 1 process and `nprocs` the number of processes created by the `fork` calls.

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

```
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/mman.h>


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

  /* ------ create and read matrix ------ */
  char* infile = argv[1];
  int   nprocs = atoi(argv[2]);
  int   threshold = atoi(argv[3]);

  FILE *fp;
  if((fp = fopen(infile,"r")) == NULL){
    perror("cannot open file");
    exit(EXIT_FAILURE);
  }
  size_t str_size = 0;
  char*  str = NULL;
  getline(&str, &str_size, fp);
  int n = atoi(str);
  int matrix[n][n];
  int i = 0, j = 0;
  while ( getline(&str, &str_size, fp) > 0 ) {
    char* token = strtok(str," ");
    while( token != NULL ) {
      matrix[i][j]=atoi(token);
      token=strtok(NULL," ");
      j++;
    }
    i++;
    j=0;
  }
  fclose(fp);

  for(i = 0; i < n; i++){
    for(j = 0; j < n; j++){
      printf("%d ", matrix[i][j]);
    }
    printf("\n");
  }
```

```
/* ------ setup shared memory ------ */

int *partials = mmap(NULL, nprocs*sizeof(int), PROT_READ|PROT_WRITE,
     MAP_SHARED|MAP_ANONYMOUS, 0, 0);

if(partials == MAP_FAILED){
  perror("mmap");
  exit(EXIT_FAILURE);
}

for(i = 0; i < nprocs; i++)
  partials[i] = 0;

/* ------ start nprocs and do work ------ */

for(i = 0; i < nprocs; i++) {
  pid_t pid;
  if ((pid = fork()) < 0) {
    perror("fork");
    exit(EXIT_FAILURE);
  }
  if(pid == 0) {
    for(int j = 0; j < n; j++)
      if(j % nprocs == i)
        for(int k = 0; k < n; k++)
          if(matrix[j][k] > threshold)
    partials[i]++;
    exit(EXIT_SUCCESS);
  }
}

/* ------ wait for nprocs to finish ------- */
for(i = 0; i < nprocs; i++)  {
  if ( waitpid(-1, NULL, 0) < 0) {
    perror("waitpid");
    exit(EXIT_FAILURE);
  }
}

/* ler resultados enviados pelos processos filhos */
int total = 0;
for(i = 0; i < nprocs; i++)
  total += partials[i];
```

```
  printf("%d\n",total);

  /* ------ release shared memory ------ */
  if (munmap(partials, sizeof(partials)) < 0) {
    perror("munmap");
    exit(EXIT_FAILURE);
  }

  exit(EXIT_SUCCESS);
}
```

Q6. The following program shows how a process can handle some signals sent by to it by other processes. This is yet another form of process communication, usually used to signal the occurrence of special events. In the `main` function we get the `signal` function that registers the action to be performed when a given type of signal is received. The operating system has default behavior for all signals. This code overrides this default behavior, when possible. In this case, the signals are the "user defined" provided by the operating system: `SIGUSR1` and `SIGUSR2`. To test this example, compile and run the program. Then, in a new terminal, find the PID of the running program, say `N` and then send it a signal using the command `kill`, e.g., `kill -SIGUSR1 N`.

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

static void handler1() { printf("received SIGUSR1\n"); }

static void handler2() { printf("received SIGUSR2\n"); }

static void handler3() { printf("received SIGHUP\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);
```

```
  }
  if (signal(SIGHUP, handler3) == SIG_ERR) {
    fprintf(stderr, "Can't catch SIGHUP: %s", strerror(errno));
    exit(EXIT_FAILURE);
  }

  /* stick around ... */
  for ( ; ; )
    pause();
}
```

Change the above code so that it supports two additional signals: `SIGTSTP` (sent by the terminal when you hit `CTRL-Z` and `SIGINT` (sent by the terminal when you hit `CTRL-C`). Print an adequate message in each handler. Try to do same for signal `SIGKILL`. What happened? How do you explain it?

Q7. The following example shows how you can exchange signals between a process and its child. read the code, complete it and run it.

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

static void handler_parent()
  { printf("%d: Parent received signal\n", getpid()); }
static void handler_child()
  { printf("%d: Child received signal\n", getpid()); }

int main(int argc, char* argv[]) {
  pid_t pid;
  if (signal(SIGUSR1, handler_parent) == SIG_ERR)
    { /* signal error */}
  if (signal(SIGUSR2, handler_child) == SIG_ERR)
    { /* signal error */}
  if ((pid = fork()) < 0)
    { /* fork error */}
  else if (pid > 0) {
    /* parent's code */
    kill(pid, SIGUSR2);
    pause();
```

```
  } else {
    /* child's code */
    kill(getppid(), SIGUSR1);
    pause();
  }
}
```

Change the previous code so that the child process sends 3 signals to the parent process. The parent process does not know how many signals it will receive. It should count the incoming signals and print the count in the terminal.

Q8. The following example shows that signals can be very useful for managing running processes. The program uses a configuration file that has some attribute values (imagine a server configuration). We can use signals to force the program to re-read the configuration file at runtime without having to stop it and rerun it (let alone recompile it). This is very useful, for example, in the case of servers that must be active all the time (a property known as "availability"). Pay attention to the code, compile it and follow the examples.

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

/* program parameters, this is just an example */
static int   param1;
static int   param2;
static float param3;
static float param4;

void read_parameters() {
  FILE *fp;

  if ( (fp = fopen(".config", "r")) == NULL ){
    printf("Missing configuration file\n");
    exit(EXIT_FAILURE);
  }

  fscanf(fp, "param1: %d\n", &param1);
  fscanf(fp, "param2: %d\n", &param2);
  fscanf(fp, "param3: %f\n", &param3);
```

```
  fscanf(fp, "param4: %f\n", &param4);

  fclose(fp);
}

void print_parameters() {
  printf("param1: %d\n", param1);
  printf("param2: %d\n", param2);
  printf("param3: %f\n", param3);
  printf("param4: %f\n", param4);
}

void handler (int signum) {
  read_parameters();
  printf ("read new parameters,values are:\n");
  print_parameters();
}

int main (int argc, char* argv[]) {
  if (signal(SIGHUP, handler) == SIG_ERR) {
    fprintf(stderr, "Can't catch SIGHUP: %s", strerror(errno));
    exit(EXIT_FAILURE);
  }

  /* print PID for reference */
  printf("my PID is %d\n", getpid());

  /* read initial parameters */
  read_parameters();

  /* stick around and catch SIGHUP signals */
  printf("working...\n");
  for ( ; ; )
    ;
}
```

Assume you stored the above program in a file `f6q8.c`. Execute the following commands:

```
$ cat > .config
param1: 263
param2: 7912
param3: -2.651178
param4: 5.222693
```

```
^D
$
$ gcc f6q8.c -o f6q8
$ ./f6q8 &
my PID is 36595
working...
$
$ kill -HUP 36595
read new parameters,values are:
param1: 263
param2: 7912
param3: -2.651178
param4: 5.222693
$
$ emacs .config (change some values)
$
$ kill -HUP 36595
read new parameters,values are:
param1: 263
param2: 321
param3: -2.651178
param4: 3.333895
$
```

Did you understand what just happened? To understand what is the signal `SIGHUP` (="HangUP") see here (`https://en.wikipedia.org/wiki/SIGHUP`), especially the section "Modern usage". The number associated with this signal is `1` so that the following command:

```
$ kill -HUP 36595
```

can also be written as:

```
$ kill -1 36595
```

Note that this example works for any signal that can be captured by the process, and not just `SIGHUP`.