L:EIC / SO2122:
Comunicação entre Processos
(usando a API do Kernel e a Standard C Library)

Q1. Considere o seguinte programa que implementa uma "pipe" entre processos pai e filho. Compile-o e execute-o. Leia o código com atenção e compreenda-o.

```
#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);
  }
}
```

Altere o programa de tal forma que, em vez das mensagens enviadas, o processo pai abra um ficheiro de texto (cujo nome deve ser dado na linha de comando), leia o seu conteúdo e o passe através da "pipe" para o processo filho. Este deverá receber o conteúdo do ficheiro e escrevê-lo no "stdout". Compile e execute o seu programa com um ficheiro de texto grande (p.e., o ficheiro com este código fonte).

Q2. O seguinte exemplo mostra como ligar o "stdout" do comando `cmd1` ao "stdin" do comando `cmd2`, usando uma "pipe". A função que permite fazer este mapeamento é `dup2`. Consulte o manual para ver como funciona. Analise o código, complete-o, compile-o e execute-o.

```
#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 que, por vezes, é possível que o resultado do comando não seja apresentado no terminal em sincronia com o "prompt" da "shell" (i.e., surge primeiro o "prompt" e depois o resultado). Como explica este fenómeno?

Q3. Use o código anterior para escrever um comando `tube` que recebe uma sequência de argumentos semelhante a uma "pipe" na "shell" (e.g., `tube "ls -l | wc -l"`), que produza os "arrays" `cmd1` e `cmd2` respectivos e execute o comando entre aspas. Como faria para executar três comandos ligados por "pipes"? E `n` comandos?

Q4. O programa seguinte implementa um mecanismo de comunicação entre processos pai e filho mas agora usando um par de "sockets". Ao invés das "pipes", os "sockets" permitem a comunicação bidirecional. Compile e execute o programa. Leia com atenção o código e compreenda-o.

```
#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;
  }
}
```

Modifique o programa de tal forma que o processo pai abra um ficheiro de texto e transfira o seu conteúdo para o processo filho. Por sua vez o processo filho deve receber o conteúdo, passar todos os caracteres para maiúsculas e devolvê-los para o processo pai que os imprime no "stdout".

Q5. O seguinte programa demonstra a utilização de um segmento de memória partilhado entre processos. A função `mmap` é usada para criar o dito segmento cujo apontador é partilhado pelos vários processos criados pelas chamadas a `fork` que se seguem. O exemplo usa o segmento de memória para guardar uma matriz de valores inteiros inicializada a partir de uma ficheiro (`infile`). Os processos calculam em paralelo o número de entradas na matriz maiores do que um valor limite (`threshold`), também fornecido como input ao programa. Cada processo varre apenas algumas linhas da matriz e guarda o número de valores encontrados num vector com os parciais por linha (também partilhado). No final, o processo pai percorre o vector de parciais e soma os valores para obter o total para a matriz. Esta forma de resolver o problema acelera significativamente a sua resolução, no limite executando em T(1)/`nprocs` unidades de tempo, onde T(1) é o tempo de execução com 1 processo e `nprocs` o número de processos criados pelas chamadas a `fork`.

```
#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. O exemplo seguinte mostra a manipulação e o tratamento de sinais pelo utilizador. Na função `main`, aparece a função `signal` que regista qual o tratamento que deve ser dado, quando o processo que executa o código recebe os sinais `SIGUSR1` e `SIGUSR2`. Para testar o exemplo, abra um terminal novo envie o sinal `SIGUSR1` ao processo N usando o comando `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();
}
```

Q7. Estenda o código anterior para que suporte o tratamento dos sinais `SIGTSTP` (enviado pelo terminal quando se usa `CTRL-Z`) e `SIGINT` (enviado pelo terminal quando se usa `CTRL-C`), imprimindo nesses casos uma mensagem adequada. Veja se consegue fazer algo de semelhante com o sinal `SIGKILL`.

Q8. O exemplo seguinte mostra como pode fazer a troca de sinais entre um processo pai e um processo filho. Analise o código, complete-o, compile-o e execute-o.

```
#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();
```

```
  }
}
```

Q9. Estenda o código anterior de forma a que o processo filho envie 3 sinais ao processo pai. O processo pai, que não sabe o número de sinais que irá receber, deverá fazer uma contagem dos sinais que recebe e imprimir essa contagem cada vez que recebe um sinal.

Q10. O seguinte exemplo mostra como os sinais podem ser úteis, por exemplo, para actualizar a configuração de um processo sem ter de o terminar e voltar a executar (muito menos recompilar). Isto é muito útil, por exemplo, no caso de servidores que devem manter-se sempre disponíveis (uma propriedade designada por "availability"). Veja o código seguinte:

```
#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 ( ; ; )
    ;
}
```

Agora, assuma que o guardam num ficheiro `f7q9.c` e execute os comandos seguintes:

```
$ cat > .config
param1: 263
param2: 7912
param3: -2.651178
param4: 5.222693
^D
$
$ gcc f7q9.c -o f7q9
$ ./f7q9 &
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
$
```

Percebeu o que aconteceu? Para perceber o que é o sinal `SIGHUP` (="HangUP") veja aqui (`https://en.wikipedia.org/wiki/SIGHUP`), em especial a secção "Modern usage". O número deste sinal é `1`, pelo que, no exemplo acima:

```
$ kill -HUP 36595
```

poderia ter sido escrito como:

```
$ kill -1 36595
```

Note que neste exercício não é obrigatório usar `SIGHUP`, poderia utilizar qualquer sinal capturável pelo processo.