

# Comunicação entre Processos

## (usando a API do kernel e a Standard C Library)

---

1. 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 LINE_SIZE 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[LINE_SIZE] = "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[LINE_SIZE];
        close(fd[1]);
        int nbytes = read(fd[0], line, LINE_SIZE);
        write(STDOUT_FILENO, line, nbytes);
        close(fd[0]);
        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 (e.g., o ficheiro com este código fonte).

**2.** 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>

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);
    }
}
```

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”.

**3.** 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 cujo `pid` é `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("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.** Adapte o código anterior para que suporte também o tratamento dos sinais `SIGTSTP` (enviado pelo terminal quando se usa `CTRL-Z`), `SIGINT` (enviado pelo terminal quando se usa `CTRL-C`) e `SIGHUP` (enviado pelo terminal quando o utilizador faz “logout”) imprimindo nesses casos uma mensagem adequada. Consegue fazer algo de semelhante com o sinal `SIGKILL`?

5. Considere o seguinte programa que executa um ciclo infinito e só pode ser terminado com um sinal externo, e.g., **SIGINT**. Reescreva-o reprogramando o **SIGINT** de tal forma que quando recebe o dito sinal sai do ciclo e termina executando a instrução **exit** imediatamente a seguir. O programa pode assim terminar a sua execução de forma graciosa.

```
#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. 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>

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();
}

```

Guarde o programa num ficheiro `updatable.c` e execute os comandos seguintes:

```

$ 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
$

```

Percebeu o que aconteceu? O número deste sinal é 1, pelo que, no exemplo acima `kill -HUP 36595` poderia ter sido escrito como `kill -1 36595`. Note ainda que neste exercício não é obrigatório usar `SIGHUP`, poderia utilizar qualquer sinal capturável pelo processo.