Considerações Gerais

Tomando como ponto de partida o código apresentado, resolva as 3 questões que se seguem. Para cada alínea, deve copiar o código para um ficheiro, editá-lo para que resolva o problema descrito, compilá-lo e verificar que funciona.

Elimine **todo** o código redundante antes de submeter. **Os programas submetidos não serão avaliados imediatamente pelo sistema, nem haverá feedback**. O código fonte submetido será arquivado para avaliação posterior.

Pode, e deve, usar a shell de comandos e as páginas de manual do sistema.

Problema 1

Considere o seguinte programa incompleto *upcase.c* que abre um ficheiro de texto cujo nome é dado na linha de comando e transfere para o *stdout* o conteúdo do mesmo sem modificações.

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

#define BUFFER_SIZE   1024

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

  /* check if 1 args available */
  if (argc != 2) {
    printf("usage: ./upcase file\n");
    return EXIT_FAILURE;
  }

  /* check if argv[1] can be opened and is readable */
  FILE* fp = fopen(argv[1], "r");
  if(fp == NULL) {
    printf("%s: cannot open %s\n", argv[0], argv[1]);
    return EXIT_FAILURE;
  }

  char buf[BUFFER_SIZE];
  int nchars;
  /* read file and print it to stdout */
  while( (nchars=fread(buf, sizeof(char), BUFFER_SIZE, fp)) != 0)
    fwrite(buf, sizeof(char), nchars, stdout);

  /* close file */
  fclose(fp);
  return EXIT_SUCCESS;
}
```

Modifique o programa **sem transferir caracter a caracter** de forma a que produza o conteúdo do ficheiro original *argv[1]* em maiúsculas e o guarde num segundo ficheiro dado pelo argumento *argv[2]*. Por exemplo:

```
{glaurung: ~luis}$ gcc -Wall upcase.c -o upcase
{glaurung: ~luis}$ cat > input.txt
abcd ef ghijk
   lmn  opq
 rst
uvxyw             z
^D
{glaurung: ~luis}$ ./upcase input.txt output.txt
{glaurung: ~luis}$ cat output.txt
ABCD EF GHIJK
   LMN  OPQ
 RST
UVXYW             Z
```

Sugestão: veja a página do manual para a função *toupper*.

Problema 2

O programa que se segue, *shell.c*, cria um processo filho que executa o comando simples dado em *argv[1]*.

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

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

  /* fork a child process */
  if ((pid = fork()) < 0 ) {
    perror("main@fork");
    return EXIT_FAILURE;
  }
  else if (pid == 0) {
    /* child process */
    if (execlp(argv[1],argv[1],NULL) < 0) {
      perror("main@execlp");
      return EXIT_FAILURE;
    }
  }
  else {
    /* parent process */
    if (waitpid(pid, NULL, 0) < 0) {
      perror("main@waitpid");
      return EXIT_FAILURE;
    }
  }
  return EXIT_SUCCESS;
}
```

Compile e experimente o programa, por exemplo:

```
{glaurung: ~luis}$ gcc -Wall shell.c -o shell
{glaurung: ~luis}$ ./shell ls
...   // output de ls
{glaurung: ~luis}$ ./shell pwd
...   // output de pwd
```

Transforme-o numa *shell* de comandos básica que:

- leia sucessivos comandos simples (sem argumentos) e os execute;
- termine com o comando especial *quit*;
- ignore o ^C (*SIGINT*).

Por exemplo:

```
{glaurung: ~luis}$ gcc -Wall shell.c -o shell
{glaurung: ~luis}$ ./shell
$ ls
... // output de ls
$ pwd
... // output de pwd
$ ^C
$ quit
{glaurung: ~luis}$
```

Sugestão: programe um *handler* para o sinal *SIGINT* (^C) que o ignore e retorne ao *prompt* da shell.

Sugestão: consulte a página de manual referente à função *signal* (da clib).

Problema 3

O programa seguinte cria um processo filho e transfere para ele uma mensagem de texto através de uma *pipe* partilhada.

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

#define READ 0
#define WRITE 1
#define BUFFER_SIZE  256

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

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

  if ((pid = fork()) < 0) {
    perror("main@fork");
    exit(EXIT_FAILURE);
  }
  else if (pid > 0) {
    /* parent */
    char  buf1[] = "To be, or not to be, that is the question";

    close(fd[READ]);
    if (write(fd[WRITE], buf1, strlen(buf1)) < 0) {
      perror("main@write");
      return EXIT_FAILURE;
    }

    /* close pipe end */
    close(fd[WRITE]);

    /* wait for child and exit */
    if ( waitpid(pid, NULL, 0) < 0) {
      perror("main@waitpid");
      return EXIT_FAILURE;
    }

    /* exit gracefully */
    exit(EXIT_SUCCESS);
  }
  else {
    /* child */
    int nchars;
    char  buf2[BUFFER_SIZE];

    close(fd[WRITE]);
    if ( (nchars = read(fd[READ], buf2, BUFFER_SIZE)) < 0 ) {
      perror("main@read");
      return EXIT_FAILURE;
    }

    /* write message from parent */
    write(STDOUT_FILENO, buf2, nchars);

    /* close pipe end */
    close(fd[READ]);

    /* exit gracefully */
    exit(EXIT_SUCCESS);
  }
}
```

Compile e experimente o programa, por exemplo:

```
{glaurung: ~luis}$ gcc -Wall pipe.c -o pipe
{glaurung: ~luis}$ ./pipe
To be, or not to be, that is the question
{glaurung: ~luis}$
```

Altere o programa de modo usar uma *pipe* para redirecionar o output do processo pai para o input do processo filho. Desta forma, o processo pai abre um ficheiro, escreve-o para o *stdout* (que está ligado a um dos lados da *pipe*). Por sua vez, o processo filho lê do *stdin* (associado ao lado de leitura da *pipe*) e escreve o conteúdo do ficheiro original para o *stdout*.

Sugestão: pode aproveitar o código de abertura e leitura de ficheiro da primeira pergunta.

Sugestão: consulte a página de manual referente às chamadas ao sistema *dup* e *dup2*.