Si chiede di scrivere una semplice shell sotto forma di una funzione int do_shell(const char* prompt); che prende come parametro la stringa di prompt e si comporta come segue:
1) Stampa il prompt;
2) Attende che l'utente inserisca in stdin un comando seguito dai suoi eventuali argomenti (es: ls -l, dove ls è il comando e -l è il suo unico argomento). Per ottenere comando e argomenti da stdin usare il risultato dell'esercizio 2;
3) Se il comando è vuoto (NULL) tornare al punto 1;
4) Se il comando è quit terminare con successo la shell;
5) Creare con fork un nuovo processo che esegua il comando con gli argomenti dati usando execvp;
6) Se il comando si riferisce a un programma inesistente riportare l'errore unknown command seguito dal nome del comando e tornare al punto 1;
7) Attendere con wait la terminazione del processo e tornare al punto 1.

```
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include "e2.h"
#define MAX_LINE    1024
#define MAX_TOKENS  64

void do_cmd(char* argv[MAX_TOKENS]) {
    int res;
    pid_t pid = fork();

    if (pid == -1) {
        perror("fork error");
        exit(EXIT_FAILURE);
    }

    if (pid == 0) {
        res = execvp(argv[0], argv);
        if (res == -1) {
            printf("unknown command %s\n", argv[0]);
            _exit(EXIT_FAILURE);
        }
    }

    res = wait(NULL);
    if (pid == -1) {
        perror("wait error");
        exit(EXIT_FAILURE);
    }
}

char* dup_string(const char* in) {
    size_t n = strlen(in);
```

```
    char* out = malloc(n + 1);
    strcpy(out, in);
    return out;
}

void get_cmd_line(char* argv[MAX_TOKENS]) {
    int argc = 0;
    char line[MAX_LINE];
    fgets(line, MAX_LINE, stdin);
    char* token = strtok(line, " \t\n");
    argc = 0;
    while (argc < MAX_TOKENS && token != NULL) {
        argv[argc++] = dup_string(token);
        token = strtok(NULL, " \t\n");
    }
    argv[argc] = NULL;
}

int do_shell(const char* prompt){
    for (;;) {
        printf("%s", prompt);
        char* argv[MAX_TOKENS];
        get_cmd_line(argv);
        if (argv[0] == NULL) continue;
        if (strcmp(argv[0], "quit") == 0) break;
        do_cmd(argv);
    }
    return EXIT_SUCCESS;
}
```

Scrivere una funzione int copy(const char* src, const char* dst) che realizza la copia di un file di dimensione arbitraria con nome file sorgente src e destinazione dst. La funzione deve restituire EXIT_FAILURE in caso di errore e EXIT_SUCCESS altrimenti.
Suggerimento: allocare dinamicamente un buffer da usare per il travaso di dati dal file sorgente al file destinazione. Provare con dimensioni diverse scegliendo la minima ragionevole per mantenere le prestazioni.
Usare il programma di prova ./test.sh che compila, esegue e verifica il risultato.

```
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include "e2.h"

void check_perror(int res, const char* msg);
#define BUF_SIZE 8192

void check_perror(int res, const char* msg) {
    if (res != -1) return;
```

```
    perror(msg);
    exit(EXIT_FAILURE);
}

int copy(const char* src, const char* dst) {

    int res;
    void* buf;

    // allocazione buffer
    buf = malloc(BUF_SIZE);
    if (buf == NULL) {
        fprintf(stderr, "malloc");
        exit(EXIT_FAILURE);
    }

    // open files
    int fdin = open(src, O_RDONLY);
    check_perror(fdin, "open");

    int fdout = open(dst, O_WRONLY | O_CREAT | O_TRUNC, 0644);
    check_perror(fdout, "open");

    // copy loop
    while(1) {
        ssize_t r = read(fdin, buf, BUF_SIZE);
        check_perror(r, "read");
        if (r==0) break;

        ssize_t w = write(fdout, buf, r);
        check_perror(w, "write");
    }

    // close files
    res = close(fdin);
    check_perror(res, "close");

    res = close(fdout);
    check_perror(res, "close");

    // deallocazione buffer
    free(buf);

    return EXIT_SUCCESS;
}
```

Si vogliono usare i segnali per creare un indicatore di progresso per la funzione do_sort, che implementa un semplice algoritmo di ordinamento a bolle. L'indicatore di progresso è la percentuale di n coperta da i, vale a dire 100.0*i/n.

```
#include "e2.h"
#include "signal.h"
#include "stdlib.h"
#include "stdio.h"
int i, max;

void handler(int signo){
      float perc = (float) 100.0 * i/max;
      printf("%f%%\n", perc);
      ualarm(500000U, 0);
}

static void do_sort(int *v, int n) {
    int j;
    for (i=0; i<n; ++i)
        for (j=1; j<n; ++j)
            if (v[j-1] > v[j]) {
                int tmp = v[j-1];
                v[j-1] = v[j];
                v[j] = tmp;
            }
}

void sort(int *v, int n) {
    max = n;
    struct sigaction a = {0};
    a.sa_handler = handler;
    int res = sigaction(SIGALRM, &a, NULL);
    if (res == -1){
      perror("errore sigaction");
      exit(EXIT_FAILURE);
    }
    ualarm(500000U, 500000U);
    do_sort(v, n);
}
```

L'obiettivo dell'esercizio è implementare l'operazione di deallocazione di un semplice allocatore di memoria. L'operazione mymalloc è fornita e si richiede di scrivere l'operazione myfree. L'operazione deve prendere il blocco deallocato e inserirlo in testa alla lista di blocchi liberi puntata dalla variabile globale free_list come descritto nella dispensa del corso. I blocchi liberi dell'allocatore hanno una header di 4 byte che contiene la dimensione del blocco, seguita da un campo next.

```
#include <unistd.h>
#include <stdlib.h>
#include "e2.h"

header_t* free_list = NULL;

void* mymalloc(size_t m) {
```

```
    header_t *p, *q = NULL;
    m = ((m+3)/4)*4;
    if (m < 8) m = 8;
    for (p=free_list; p != NULL; q = p, p = p->next)
        if (p->size >= m) break;
    if (p == NULL) p = sbrk(4 + m);
    else
        if (q == NULL) free_list = free_list->next;
        else q->next = q->next->next;
    ((header_t*)p)->size = m;
    return (char*)p+4;
}


void myfree(void* p) {
     header_t * q = (header_t*) (p - 4);
     q->next = free_list;
     free_list = q;
}
```

Scrivere una funzione archiver che crea un file archivio in cui inserisce un numero arbitrario di file in modo simile a quanto avviene per un file tar.
1) archive è il pathname del file archivio di output (es. archive.dat)
2) files è un array di stringhe che rappresentano i pathname dei file di input da archiviare in archive
3) n è il numero di file da archiviare
Ogni file archiviato ha una header formata da: 1) 256 byte che contengono il pathname del file (come stringa C, quindi con terminatore zero alla fine del pathname - non è necessario ovviamente che il pathname effettivo usi tutti i 256 byte disponibili e i byte extra saranno padding) e 2) 8 byte che rappresentano la dimensione del file. Alla header seguono i byte del file stesso.
Se un file con il pathname archive esiste già , il suo contenuto deve essere inizialmente troncato a dimensione zero dalla funzione archiver. Il file archivio creato deve avere privilegi di lettura e scrittura per l'utente proprietario, sola lettura per il gruppo proprietario, e nessun permesso per tutti gli altri.

```
#include "e2.h"
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

void check_error(long res, char* msg) {
    if (res != -1) return;
    perror(msg);
```

```
    exit(EXIT_FAILURE);
}

#define FILENAME_LEN 256
#define BUF_SIZE 4096

void copy_file(int fd_dest, int fd_src) {
    ssize_t res;
    char buf[BUF_SIZE];
    for (;;) {
        res = read(fd_src, buf, BUF_SIZE);
        check_error(res, "read");
        if (res == 0) break;
        res = write(fd_dest, buf, res);
        check_error(res, "write");
    }
}

void archiver(const char* archive, const char** files, int n) {
    char filename[FILENAME_LEN];
    int fd_archive, fd, i;
    long res;

    fd_archive = open(archive, O_WRONLY | O_CREAT | O_TRUNC, 0644);
    check_error(fd_archive, "open");

    for (i=0; i<n; ++i) {
        fd = open(files[i], O_RDONLY);
        check_error(fd, "open");

        long size = lseek(fd, 0, SEEK_END);
        check_error(size, "lseek");

        res = lseek(fd, 0, SEEK_SET);
        check_error(res, "lseek");

        strcpy(filename, files[i]);

        res = write(fd_archive, filename, FILENAME_LEN);
        check_error(res, "write");

        res = write(fd_archive, &size, sizeof(size));
        check_error(res, "write");

        copy_file(fd_archive, fd);

        res = close(fd);
        check_error(res, "close");
    }

    res = close(fd_archive);
    check_error(res, "close");
```

```
}
```

Scrivere una funzione C int par_find(int* v, unsigned n, int x) che cerca se x appartiene all'array v di dimensione n. La soluzione deve usare almeno due processi distinti che effettuano la ricerca in sottoparti distinte di v in modo indipendente l'uno dall'altro.

Suggerimento: fare in modo che il processo restituisca al genitore come stato di exit 1 se x è presente nella porzione esplorata dal processo, e 0 altrimenti.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include "e1.h"

#define PROC 10

int find(int* v, unsigned n, int x) {
    int i;
    for (i=0; i<n; ++i)
        if (v[i]==x) return 1;
    return 0;
}

int par_find(int* v, unsigned n, int x){
    int i, res = 0;
    for (i=0; i<PROC; ++i) {
        pid_t pid = fork();
        if (pid == -1) {
            perror("fork");
            _exit(EXIT_FAILURE);
        }
        if (pid == 0) {
            int res = find(v+i*n/PROC, n/PROC, x);
            _exit(res);
        }
    }
    for (i=0; i<PROC; ++i) {
        int status;
        wait(&status);
        if (WIFEXITED(status)) {
            res = res || WEXITSTATUS(status);
        }
    }
    return res || v[n-1] == x;
}
```

Si vuole scrivere un programma che calcola la somma dei valori senza segno a 32 bit contenuti in un file, ignorando eventuali byte finali resto della divisione per 4, se la lunghezza del file in byte non è divisibile per 4. La lettura deve avvenire 4 byte alla volta.

```
#include <unistd.h> // read, write, close
#include <fcntl.h>  // open
#include <stdlib.h> // exit
#include <stdio.h>  // perror
#include "e1.h"

int sum(const char* filename, unsigned long *psum) {
    unsigned long s = 0;
    int fd = open(filename, O_RDONLY);
    if (fd == -1) {
        perror("errore in open");
        exit(EXIT_FAILURE);
    }

    for(;;) {
        unsigned data;
        ssize_t res = read(fd, &data, sizeof(data));
        if (res == -1) {
            perror("errore in read");
            exit(EXIT_FAILURE);
        }

        if (res == 4) s += data;

        if (res == 0) break;
    }

    fd = close(fd);
    if (fd == -1) {
        perror("errore in close");
        exit(EXIT_FAILURE);
    }

    *psum = s;

    return EXIT_SUCCESS;
}
```

Lo scopo dell'esercizio è quello di scrivere file binari con la seguente struttura composti da numeri pseudo-casuali:

```
 0 |magic number|
 4 |    size     |
 8 |    rnd 1    |
12 |    rnd 2    |
16 |     ...     |
```

Il magic number deve essere 0xEFBEADDE per tutti i file generati con questo programma. I magic number sono codici associati ai tipi di file, specialmente se binari, per identificarli univocamente al momento dell'apertura, e sono disposti all'inizio del file. Il magic number è considerato a scopo di appredimento. Subito dopo c'è il numero di valori casuali del file.
Si richiede di scrivere una funzione int make_rnd_file(int size, int seed, int mod, char *filename); con i seguenti parametri:
1) size: numero di valori casuali registrati nel file;
2) seed: seme del generatore pseudo-casuale;
3) mod: limite superiore per i valori casuali, valore assoluto;
4) filename: percorso assoluto o relativo del file da generare.

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <fcntl.h>
#include "e2.h"

int make_rnd_file(unsigned size, unsigned seed, unsigned mod, char
*filename) {

    int res, data, fd, i, magic = MAGIC_NUMBER;

    // set pseudo-random generator seed
    srand(seed);

    // open file
    mode_t mode = S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH;
    fd = open(filename, O_WRONLY | O_TRUNC | O_CREAT, mode);
    check_perror(fd, "open");

    // write file magic code
    res = write(fd, &magic, sizeof(magic));
    check_perror(res, "write");

    // write size of the file in terms of random ints
    res = write(fd, &size, sizeof(size));
    check_perror(res, "write");

    // write random data
    for (i=0; i < size; ++i){
        data = rand() % mod;
        res = write(fd, &data, sizeof(data));
        check_perror(res, "write");
    }

    res = close(fd);
    check_perror(res, "write");

    return EXIT_SUCCESS;
```

```
}
```

Ottimizzare la soluzione dell'esercizo "somma valori" leggendo a blocchi di 4KB e non a word di 32 bit.

```
#include <unistd.h> // read, write, close
#include <fcntl.h>  // open
#include <stdlib.h> // exit
#include <stdio.h>  // perror
#include "e3.h"
#define BLOCK_SIZE 1024

int sum(const char* filename, unsigned long *psum) {

    unsigned *block = malloc(BLOCK_SIZE*sizeof(unsigned)), i;
    if (block == NULL) {
        perror("errore in malloc");
        exit(EXIT_FAILURE);
    }

    unsigned long s = 0;

    int fd = open(filename, O_RDONLY);
    if (fd == -1) {
        perror("errore in open");
        exit(EXIT_FAILURE);
    }

    for(;;) {
        ssize_t    res    =    read(fd,    block,    sizeof(BLOCK_SIZE)*
sizeof(unsigned));
        if (res == -1) {
            perror("errore in read");
            exit(EXIT_FAILURE);
        }

        if (res > 0){
            for (i = 0; i < res/4; ++i)
                s += block[i];
        }
        else break; //  termina quando il file si svuota
    }

    fd = close(fd);
    if (fd == -1) {
        perror("errore in close");
        exit(EXIT_FAILURE);
    }

    free(block);
```

```
    *psum = s;

    return EXIT_SUCCESS;
}
```

Scrivere una funzione int file_eq(char* f1, char* f2) che, dati i percorsi di due file f1 e f2, restituisce zero se i file sono uguali, un valore maggiore di zero se i file sono diversi, e -1 in caso di errore.

```
#include "e4.h"
#include <errno.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

#define BUF_SIZE 4096

int file_eq(char* f1, char* f2) {

    int res, equal = 1;
    char* buf1 = NULL;
    char* buf2 = NULL;
    int fd1 = -1;
    int fd2 = -1;

    buf1 = malloc(BUF_SIZE);
    if (buf1 == NULL) goto error;

    buf2 = malloc(BUF_SIZE);
    if (buf2 == NULL) goto error;

    fd1 = open(f1, O_RDONLY);
    if (fd1 == -1) goto error;

    fd2 = open(f2, O_RDONLY);
    if (fd2 == -1) goto error;

    for (;;) {
        int r1 = read(fd1, buf1, BUF_SIZE);
        if (r1 == -1) goto error;

        int r2 = read(fd2, buf2, BUF_SIZE);
        if (r2 == -1) goto error;

        if (r1 != r2 || memcmp(buf1, buf2, r1) != 0) {
            equal = 0;
            break;
        }

        if (r1 == 0) break;
```

```
    }

    free(buf1);
    free(buf2);
    res = close(fd1);
    if (res == -1) goto error;
    res = close(fd2);
    if (res == -1) goto error;

    return !equal;

error: ;
    int e = errno;
    if (buf1 != NULL) free(buf1);
    if (buf2 != NULL) free(buf2);
    if (fd1 != -1) close(fd1);
    if (fd2 != -1) close(fd2);
    errno = e;
    return -1;
}
```

Si vuole scrivere un programma che crea n figli tale che il figlio i-esimo dorme i secondi, con i=1,...,n e termina con successo. Per ogni figlio creato, il programma deve stampare "- creato figlio <pid>", dove pid è il pid del processo generato. Dopo aver creato i figli, il genitore rimane in attesa perenne. La terminazione dei figli è catturata tramite il segnale SIGCHLD. Il gestore del segnale (che gira nel processo genitore) fa una wait; quando l'ultimo processo termina, il gestore del segnale manda in segnale SIGTERM a se stesso per terminare anche il genitore. Per ogni figlio terminato, il gestore del segnale stampa "* terminato figlio <pid>", dove pid è il pid del processo terminato.
//Parametro il numero n di processi da creare.

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

int last;

void handler(int signum) {
    int status;
    pid_t pid = wait(&status);
    if (pid == -1) {
        perror("wait error");
        exit(EXIT_FAILURE);
    }
    printf("* terminato figlio %d\n", pid);
    if (pid == last)
        kill(getpid(), SIGTERM);
```

```
}

void do_work(int n){

    int i;
    struct sigaction act = { 0 };
    act.sa_handler = handler;
    int ret = sigaction(SIGCHLD, &act, NULL);
    if (ret == -1) {
        perror("sigaction error");
        exit(EXIT_FAILURE);
    }

    for (i=1; i<=n; ++i) {
        pid_t pid = fork();
        if (pid == -1) {
            perror("fork error");
            exit(EXIT_FAILURE);
        }

        if (pid == 0) {
            printf("- creato figlio %d\n", getpid());
            sleep(i);
            _exit(EXIT_SUCCESS);
        }

        if (pid > 0 && i == n)
            last = pid;
    }

    while(1)
        pause();
}
```

Scrivere una funzione proc_gettime che prende come parametri il pathname di un eseguibile file, un puntatore a una funzione callback get_arg fornita dall'utente, un puntatore a dei dati data da fornire alla funzione callback, e un numero di ripetizioni n. La funzione proc_gettime esegue file per n volte, misurando il tempo in secondi (double) richiesto dall'esecuzione e restituisce la media dei tempi richiesti sulle n esecuzioni. Ad ogni esecuzione, la funzione chiama la callback per ottenere gli argomenti argv da passare al processo in modo da rendere possibile comportamenti diversi del processo ad ogni ripetizione. La callback deve ricevere il puntatore data preso come parametro da proc_gettime e il numero della ripetizione i con i compreso tra 0 e n-1.

```
#include "e2.h"
#include <stdlib.h>
#include <time.h>
#include <stdio.h>
#include <unistd.h>
```

```
#include <sys/wait.h>


double get_real_time() {
    struct timespec tp;
    clock_gettime(CLOCK_MONOTONIC, &tp);
    return tp.tv_sec + tp.tv_nsec*1E-9;
}

double proc_gettime(const char* file, char** (*get_argv)(int i, void*
data), void *data, int n){
    int i;
    double start, elapsed = 0.0;
    for (i=0; i<n; ++i) {
        pid_t pid = fork();
        if (pid == -1) {
            perror("fork");
            exit(EXIT_FAILURE);
        }
        if (pid == 0) {
            execvp(file, get_argv(i,data));
            perror("execvp");
            _exit(EXIT_FAILURE);
        }
        start = get_real_time();
        pid = wait(NULL);
        if (pid == -1) {
            perror("wait");
            exit(EXIT_FAILURE);
        }
        elapsed += get_real_time() - start;
    }
    return elapsed/n;
}
```

Si vuole scrivere una funzione che misura il tempo medio in millisecondi per accesso a disco in lettura e scrittura in modo casuale e in modo sequenziale. La funzione ha il seguente prototipo:
void file_access_time(const char* file, unsigned trials, time_rec_t *t)
dove file è il file da accedere (sovrascrivendolo), trials è il numero di accessi casuali da effettuare sia per le letture che per le scritture, e t è una struttura definita come segue in E3-files/e3.h:

```
typedef struct {
    unsigned file_size;
    double seq_read_t;
    double seq_write_t;
    double rnd_read_t;
    double rnd_write_t;
} time_rec_t;
```

I campi della struttura vanno interpretati come segue:
1) file_size è la dimensione in byte del file
2) seq_read_t è il tempo medio in millisecondi per lettura sequenziale di 4 byte
3) seq_write_t è il tempo medio in millisecondi per scrittura sequenziale di 4 byte
4) rnd_read_t è il tempo medio in millisecondi per lettura casuale di 4 byte
5) rnd_write_t è il tempo medio in millisecondi per scrittura casuale di 4 byte

Tutti i campi della struttura devono essere riempiti dalla funzione file_access_time. Le letture e le scritture sequenziali devono scorrere l'intero file. I valori che vengono scritti nel file non sono rilevanti e possono essere arbitrari. L'eseguibile può essere compilato con make ed eseguito con ./e3 file trials dove file e trials sono definiti come sopra.

```
#include "e3.h"
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <sys/types.h>
#include <unistd.h>

double get_real_time_msec() {
    struct timespec tp;
    clock_gettime(CLOCK_MONOTONIC, &tp);
    return tp.tv_sec*1E3 + tp.tv_nsec*1E-6;
}

void check_error(int err, char* msg) {
    if (err != -1) return;
    perror(msg);
    exit(EXIT_FAILURE);
}

void file_access_time(const char* file, unsigned trials, time_rec_t *t)
{
    unsigned i, val, cnt;
    double start, elapsed;
    off_t off;
    int res, fd = open(file, O_RDWR);
    check_error(fd, "open");

    // get file size
    t->file_size = lseek(fd, 0, SEEK_END);
    check_error(t->file_size, "lseek");

    printf("File size: %f MB\n", t->file_size/(1024.0*1024.0));
```

```
// random read
printf("Performing %u random reads...\n", trials);
start = get_real_time_msec();
for (i=0; i<trials; ++i) {
    off = rand() % t->file_size;
    off = lseek(fd, off, SEEK_SET);
    check_error(off, "lseek");
    res = read(fd, &val, sizeof(val));
    check_error(res, "read");
}
t->rnd_read_t = (get_real_time_msec() - start)/trials;

// random write
printf("Performing %u random writes...\n", trials);
start = get_real_time_msec();
for (i=0; i<trials; ++i) {
    off = rand() % t->file_size;
    off = lseek(fd, off, SEEK_SET);
    check_error(off, "lseek");
    res = write(fd, &val, sizeof(val));
    check_error(res, "read");
}
t->rnd_write_t = (get_real_time_msec() - start)/trials;

// sequential read
printf("Reading the file sequentially...\n");
off = lseek(fd, 0, SEEK_SET);
check_error(off, "lseek");
cnt = 0;
start = get_real_time_msec();
for (;;) {
    cnt++;
    res = read(fd, &val, sizeof(val));
    check_error(res, "read");
    if (res == 0) break;
}
t->seq_read_t = (get_real_time_msec() - start)/cnt;

// sequential write
printf("Writing the file sequentially...\n");
off = lseek(fd, 0, SEEK_SET);
check_error(off, "lseek");
start = get_real_time_msec();
for (i=0; i<cnt; ++i) {
    res = write(fd, &val, sizeof(val));
    check_error(res, "read");
    if (res == 0) break;
}
t->seq_write_t = (get_real_time_msec() - start)/cnt;

res = close(fd);
```

```
    check_error(res, "close");
}
```