3. (10 poena)

l – list
p – path

Korišćenjem sistemskih poziva `fork`, `execlp` i `wait`, napisati program koji implementira jedan krajnje jednostavan interpreter komandne linije (engl. *command line interpreter*). Ovaj interpreter treba da učitava niz stringova razdvojenih belinama (razmacima ili novim redovima) sa standardnog ulaza, sve dok ne učita string „q“ koji prekida njegov rad. Kada učita svaki string, interpreter treba da pokrene proces nad programom zadatim stazom u tom stringu, sačeka njegov završetak, i pređe na sledeći string. Ukoliko kreiranje procesa nije uspelo, treba da ispiše poruku o grešci i pređe na sledeći. Pretpostavlja se da svaki string ima najviše 32 znaka. Podsetnik na bibliotečne funkcije i sistemske pozive koje se mogu koristiti:

- `scanf`: učitava sa standardnog ulaza; ukoliko se u formatizacionom specifikatoru, iza znaka %, napiše ceo broj, on označava maksimalan broj znakova koji će se učitati sa stadardnog ulaza; na primer, `%32s` učitava string, ali ne duži od 32 znaka (i dodaje `'\0'` na kraj);
- `void wait(int pid)`: suspenduje pozivajući roditeljski proces dok se ne završi proces-dete sa zadatim PID; ako je vrednost argumenta 0, pozivajući proces se suspenduje dok se ne završe sve procesi-deca;
- `int strcmp(char*,char*)`: poredi dva data stringa i vraća 0 ako su jednaki.

Rešenje:

```
int main() {
    char path[33];
    while (1) {
        scanf("%32s", path);
        if (strcmp(path, "q") == 0) {
            break;
        }
        pid_t pid = fork();
        if (pid < 0) {
            printf("fork failed\n");
            continue;
        }
        if (pid == 0) {
            execlp(path, path, NULL);
            pritf("execlp failed\n");
            exit(1);
        }
        wait(pid);
    }
}
```

pid (child) > 0 parent
= 0 child
< 0 fail.

```
main() {
    ...
    return 45;
}
```

17...
45...

exit(45);

1. (10 poena)

Koristeći samo sistemske pozive *fork*, *wait* i *exit*, kao i funkciju *printf*, napisati C program koji pronalazi maksimalnu vrednost svih elemenata ogromne celobrojne matrice dimenzije *M* puta *N* tako što uporedo pronalazi maksimum u svakoj vrsti matrice (maksimum svake vrste pronalazi u po jednom od pokrenutih procesa-dece), a onda pronalazi maksimum tih maksimuma. U slučaju greške, ovaj program treba da ispiše poruku o grešci i vrati status -1, a u slučaju uspeha treba da ispiše pronađeni maksimum i vrati status 0. Pretpostaviti da je matrica već nekako inicijalizovana.

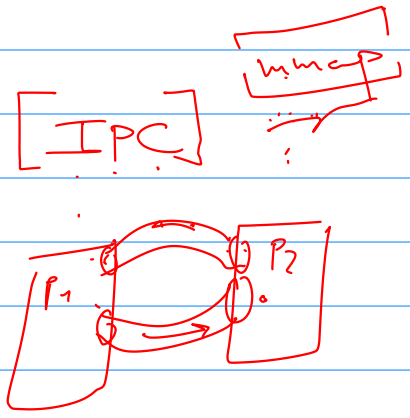
exit(-1)

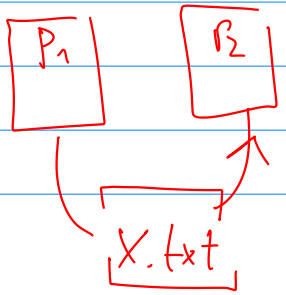
exit(0)

```
const int M = ..., N = ...;
extern int mat[M][N];
```

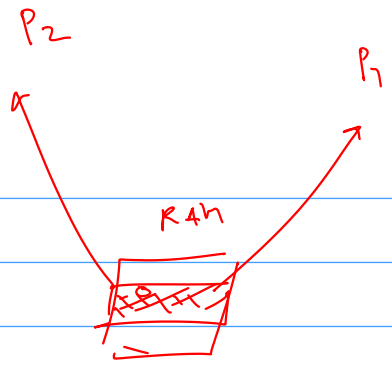
Rešenje:

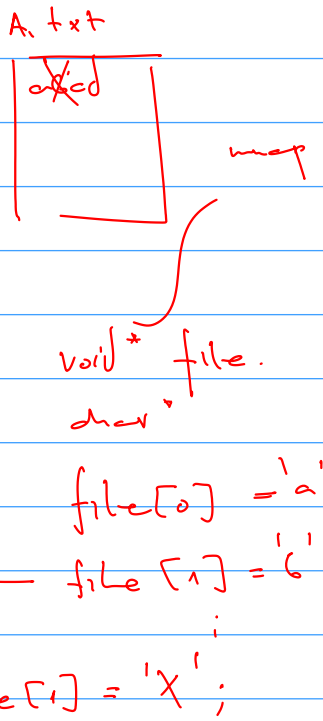
```
int main() {
    pid_t pids[M];
    for (int i=0; i<M; i++){
        pids[i] = fork();
        if (pids[i] < 0) {
            printf("fork failed\n");
            exit(-1);
        }
        if (pids[i]==0) {   child proc
            int max = M[i][0];
            for (int j=0; j<N; j++){
                if (max < M[i][j]){
                    max = M[i][j];
                }
            }
            exit(max);
        }
    }
    max = M[0][0];
    for (int i=0; i<M; i++){
        int exit_code;
        wait(pids[i], &exit_code);
        if (exit_code > max) {
            max = exit_code;
        }
    }
```





```
printf ("%d\n", num);
exit(0);        return 0;
}
```





pid_t my_pid = getpid();

1. (10 poena)

Koristeći Unix sistemske pozive *fork*, *exit*, *wait* i *getpid* koji vraća *pid* pozivajućeg procesa, kao i funkciju *printf*, napisati C program koji, kada se nad njim pokrene (roditeljski) proces, pokreće *N* procesa dece, gde je *N* predefinisana konstanta. Svaki *i*-ti proces dete treba da se vrti u praznoj petlji *i**1.000.000 puta i potom završi. Nakon što pokrene sve procese – svoju decu, roditeljski proces treba da ih sačekuje da završe, jednog po jednog, u bilo kom redosledu, i da za svaki završeni proces dete ispiše njegov redni broj *i* i njegov *pid*. Kada sačeka da sva njegova deca završe, završava se i roditeljski proces.

Rešenje:

```
int main() {

    pid_t pids[N];

    for(int i=0; i<N; i++) {
        pids[i] = fork();
        if (pids[i] == 0){
            pid_t mypid = getpid();
            for(int j=0; j<i*10^6; j++);
            exit(i);
        }
    }

    for (int i=0; i<N; i++) {
        int idx;
        wait(&idx);
        printf("i=%d\n, pid=%d\n", idx, pids[idx]);
    }

    return 0;
}
```

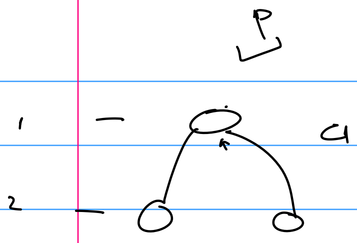
1. (10 poena)

a)(7) Korišćenjem Unix sistemskih poziva *fork*, *exit* i *wait/waitpid* implementirati funkciju `max` koja pronalazi maksimum celobrojnih vrednosti `val` sadržanih u čvorovima binarnog stabla predstavljenih strukturom `Node`, tako što proces koji poziva ovu funkciju sa argumentom koji ukazuje na koren datog stabla obrađuje taj koren i njegovo levo podstablo, a za obradu desnog podstabla kreira poseban proces koja teče uporedo sa obradom levog podstabla, i tako dalje rekurzivno za svako podstablo na isti opisani način. Ignorisati greške.

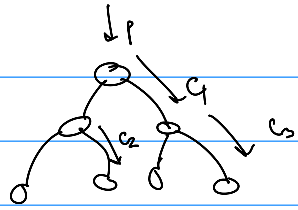
b)(3) Ako se ova funkcija pozove za koren potpunog balansiranog binarnog stabla sa *n* nivoa i 2^n - 1čvorova, koliko ukupno procesa obrađuje ovo stablo, uključujući i početni koji poziva ovu funkciju za koren ovog stabla? Precizno obrazložiti odgovor.

```
struct Node {
  Node *left, *right;
  int val;
};

int max (Node* nd);
```

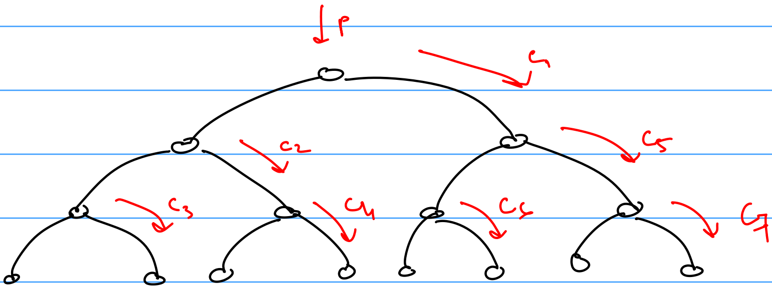
```
int max ( Node* root ){
    int left_max = root->val;
    if ( root -> left != null ){
        left_max = max ( root -> left );
    }
    int right_max = root -> val;
    if ( root -> right != null ){
        pid_t pid = fork();
        if ( pid == 0 ) {
            right_max = max ( root -> left );
            exit ( right_max );
        }
        wait ( pid , &right_max );
    }

    int max = root -> val;
    if ( max < right_max ) max = right_max;
    if ( max < left_max ) max = left_max;
    return max;
}
```



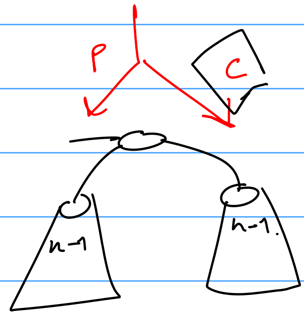
$n = 2 \quad np = 2$



$n = 3 \quad np = 4$



$n = 4 \quad np = 8$



$np(n) \rightarrow$ број потеза које покреће функција при обради стабла са n нива.

$$np(n) = (np(n-1) - 1) + (np(n-1) - 1) + 2 = 2\,np(n-1) \qquad np(1) = 1.$$

$$np(n) = 2\,np(n-1) = 2 \cdot 2 \cdot np(n-2) = \cdots = \underbrace{2 \cdots 2}_{n-1} \underbrace{np(1)}_{1} = 2^{n-1}$$

1. (10 poena)

Dat je deo POSIX API-a za niti (*pthread*). Pomoću ovih funkcija napraviti dole dati objektno orijentisani API koji niti predstavlja kao objekte klase `Thread` sa značenjem funkcija kao u školskom jezgru i sa sinhronizacijom u destruktoru koja obezbeđuje da se objekat te klase neće uništiti sve dok se pripadajuća nit ne završi. Ignorisati greške.

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                   void*(*thread_body)(void*), void *arg);
int pthread_join(pthread_t *thread, void**);

class Thread {
public:
   void start ();
protected:
   Thread();
   virtual ~Thread();
   virtual void run ();
};
```

```
private:
    pthread_t thread;

static void* Thread::wrapper (void* arg) {
        ((Thread*) arg) -> run();
        return NULL;
    }

void Thread::start() {
        pthread_create (&thread, null, &Thread::wrapper, (void*) this);
}

Thread::~Thread() {
        pthread_join (&thread, null);
}
```

4. (10 poena)

U nekom operativnom sistemu sistemski poziv `clone()` pravi novu nit (*thread*) kao klon roditeljske niti, sa istovetnim kontekstom izvršavanja i u istom adresnom prostoru kao što je i roditeljska nit – isto kao i `fork()`, samo što pravi nit u istom adresnom prostoru, a ne proces. Ovaj poziv vraća 0 u kontekstu niti-deteta, a vrednost >0 koja predstavlja ID kreirane niti u kontekstu niti-roditelja. Vraćena vrednost <0 označava neuspešan poziv. Sistemski poziv `terminate(int)` gasi nit sa datim ID. Korišćenjem ovog sistemskog poziva realizovati klasu `Thread` sa istim interfejsom kao u školskom jezgru (kreiranje niti nad virtuelnom funkcijom `run()` i pokretanje niti pozivom funkcije `start()`).

```
Thread {
    virtual void run() { }
    void start();
    virtual ~Thread() {}
    int id; = 0;
}

class A {
    void f();
}
class B extends A {
    @Override
    void f() {
        System.out.println("Hello");
    }
}

A x = new B();
x.f();
```

```
class A {
    [virtual] void f() { }
}
class B : public A {
    int arr = new int[N];
    ~B() {
        delete [] arr;
    }
    void f() override {
        cout << "Hello" << endl;
    }
}
A *p = new B();
p->f();
```

```
void Thread::start() {
    int [tid = clone();]
    if (tid == 0) {
        this -> run();
        while (id == 0);
        terminate(id);
    } else {
        ~> id = tid;
    }
}
```

```
Thread t(...);
t.start();
```

3. (10 poena)

U nekom sistemu sistemski poziv `fork()` kreira nit – klon pozivajuće niti, sa iskopiranim celokupnim stekom, slično istoimenom sistemskom pozivu na sistemu Unix (osim što se ovde radi o nitima, a ne procesima).

Dole je dat program čija je zamisao da izvršava dve uporedne niti, jednu koja učitava znak po znak sa stadardnog ulaza i taj znak prenosi kroz promenljivu `c` drugoj niti, koja taj primljeni znak ispisuje na standardni izlaz, i tako neograničeno.

a)(5) Precizno objasniti problem koji ovaj program ima i ispraviti taj problem.

b)(5) Prepraviti samo funkciju `pipe()` tako da se umesto jednog para niti koje vrše razmenu znakova, formiraju dva para takvih niti; svaki par niti predstavlja odvojeni „tok“, pa je potrebno definisati dva para promenljivih `c` i `flag` (npr. `c1`, `c2`, `flag1` i `flag2`).

```
#include <iostream>

void writer (char* c, int* flag) {
  while (1) {
    while (*flag==1);
    cin>>(*c);
    *flag = 1;
  }
}

void reader (char* c, int* flag) {
  while (1) {
    while (*flag==0);
    cout<<(*c);
    *flag = 0;
  }
}

void pipe () {
  char c;
  int flag = 0;
  if (fork())
    writer(&c,&flag);
  else
    reader(&c,&flag);
}

void main () {
  pipe();
}
```

```
void pipe() {
    static char c1, c2;
    static int flag1 = 0, flag2 = 0;

    if (fork()) {
        writer(&c1, &flag1);
    else {
        if (fork()) {
            reader(&c1, &flag1);
        } else {
            if (fork()) {
                writer(&c2, &flag2);
            else {
                reader(&c2, &flag2);
            }
        }
    }
}
```

3. (10 poena)

Dat je neki veliki jednodimenzioni niz `data` veličine `N`, čiji su elementi tipa `Data` (deklaracije dole). Jedan element niza obrađuje procedura `processData`. Ovaj niz potrebno je obraditi pomoću `n` uporednih niti (procedura `parallelProcessing`), gde je `n` zadati parametar, tako što svaka od tih uporednih niti iterativno obrađuje približno isti broj elemenata ovog velikog niza. Drugim rečima, niz treba particionisati na `n` disjunktnih podnizova, što približnije jednakih, i te particije obrađivati uporedo. Implementaciju dati u školskom jezgru.

```
const int N = ...;
class Data;
Data data[N];
void processData(Data*);
void parallelProcessing (int n);
```

$N \% n \neq 0$

$N/n \qquad N \% n \in [0, n)$

```
class Worker : public Thread {
public:
    int start_idx, end_idx;          [start_idx, end_idx)
    void run() override {
        for (int i = start_idx; i < end_idx; i++) {
            processData(&data[i]);
        }
    }
    Worker(int start_idx, int end_idx)
        : start_idx(start_idx), end_idx(end_idx) {}
};

void parallelProcessing(int n) {
    int rem = N % n;
    int base = N / n;
    int start_idx = 0;
    int end_idx;
    Worker **workers = new Worker*[n];
    for (int i = 0; i < n; i++) {
        end_idx = start_idx + base;
        if (rem > 0) {
            end_idx++;
            rem--;
        }
        workers[i] = new Worker(start_idx, end_idx);
    }
```

```
for (int i=0; i<n ; i++) {
        workers[i] -> start();
}.
```

~ Thread — join

```
for (int i=0 ; i<n ; i++) {
        delete workers[i];
}.

delete [] workers;
}.
```

4. (10 poena)

Korišćenjem niti u školskom jezgru realizovati klasu `Search` koja ima dole dati interfejs. Objekat ove klase obavlja pretragu datog niza celih brojeva `a` u opsegu indeksa počev od `i` zaključno sa `j`, tražeći u njemu vrednost `x`, ali tako što pretragu obavlja konkurentno i binarno-rekurzivno: jedna već kreirana nit nastavlja pretragu jedne polovine datog niza, a kreira novu nit koja vrši istu takvu pretragu druge polovine niza, i tako dalje rekurzivno. Ako se na nekom elementu *n* pronađe vrednost *x*, treba ispisati „Found at *n*!“, a ako se tu ne nađe ta vrednost, treba ispisati „Not found at *n*!“.

```
class Search : public Thread {
public:
  Search (int a[], int i, int j, int x);
};
```

Primer korišćenja ove klase je sledeći:

```
int array[N];
Thread *t = new Search(array,0,N-1,5);
t->start();
```

[0, N-1]

Rešenje:

[1/2] [2/2]

for(--) {

if (a[i] == x) {

i

}.

her Search (a, , , x)

class Search : public Thread {

private:

int a[], i, j, x;

Search (int a[], int i, int j, int x) :

: a(a), i(i), j(j), x(x) { }.

```
void run() override {
    if (i == j) {
        if (a[i] == x) {
            cout << "Found at " << i << endl;
        } else {
            cout << "not Found at " << i << endl;
        }
    } else {
        int m = (i + j) / 2
        Search * rh = new Search(a, m, j, x);
        rh -> start();
        for (int k = i; k < m; k++) {
            if (a[k] == x)
                cout << "Found at " << k << endl;
            } else {
                cout << "not Found at " << k << endl;
            }
        }
        delete rh;
    }
}
};
```