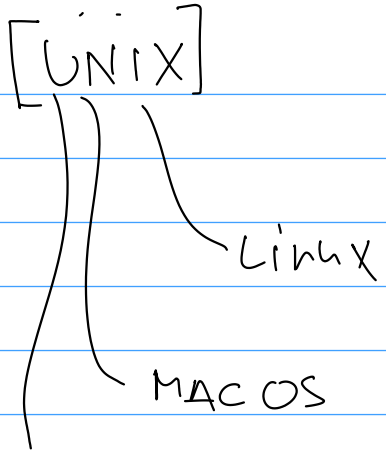


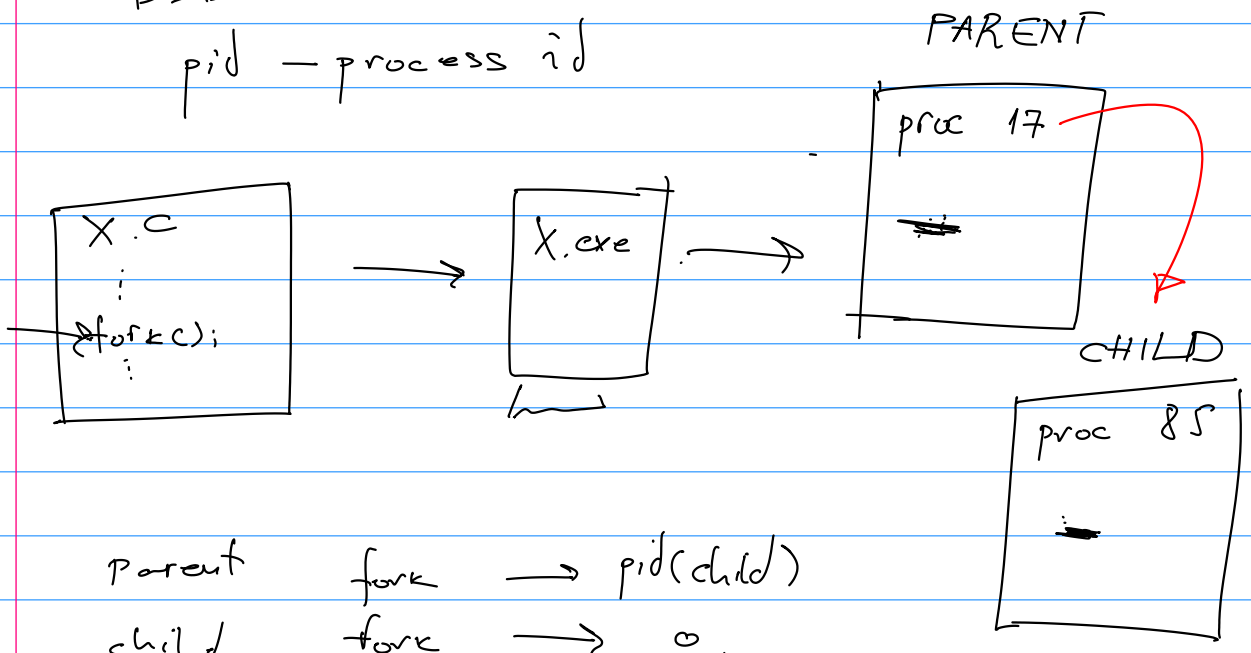
→ Локални процеси
→ Remote процеси
→ системни.

→ pid < 0

fork

pid_t

pid — process id



parent fork → pid(child)
child fork → 0.

```
pid_t pid = fork();
printf("Ela!");
if (pid == 0) {
    printf("Hello from child!");
} else {
    printf("Hello from parent!");
}
```

if (pid < 0) { ..... error }

child →

parent →

wait(int* status)

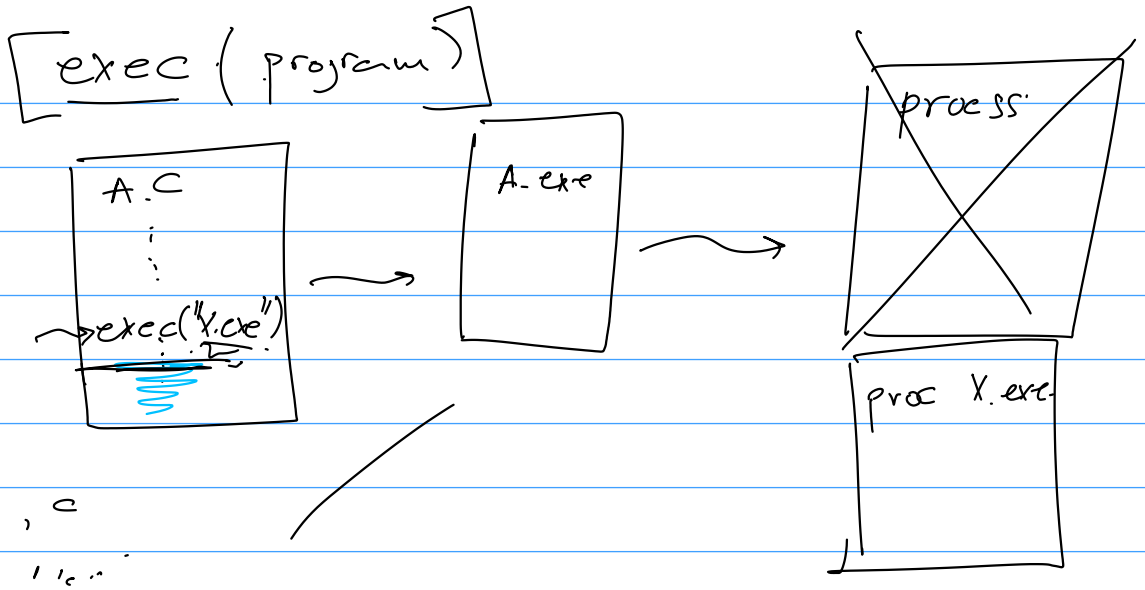
exit code

wait(pid, status)

exit(exit_code);

{variadic

exec (program)



X. a, c, c

"a", "e", "c"

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

}. list

↗

execl (path, arg1, ..., argn, NULL);

execlp (file, arg1, ---, argn, NULL);

list path

execle (path, arg1, ---, argn, NULL, environ);

list environment

"ls"

{p1, p2, ..., NULL}

int main (int argc, char ** argv, char ** environ) {

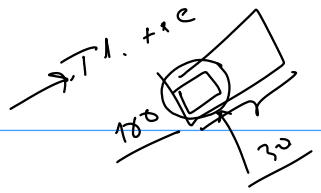
execv (path, argv);

vector

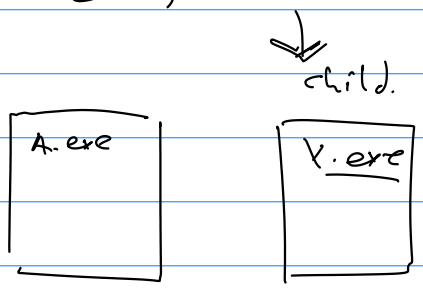
execvp (file, argv);

path execvpe (file, argv, environ);

fork — exec




```
pid_t pid = fork();        0        >0.

if (pid == 0) {
    exec("X.exe");
}
```



4. (10 poena)

U sistemu UNIX i sličnim sistemima sistemski poziv

```
int execvp(const char * filename, char * const args[]);
```

radi isto što i poziv execlp, s tim što drugi argument predstavlja niz pokazivača na nizove znakova (*null-terminated strings*) koji predstavljaju listu argumenata poziva programa koji treba izvršiti. Po konvenciji, prvi argument pokazuje na naziv fajla samog programa koji se izvršava. Ovaj niz pokazivača mora da se završi elementom sa vrednošću NULL.

U nastavku je dat jedan neispravan program. Namera programera je bila da sastavi program koji prima jedan celobrojni argument (po konvenciji, u programu se on vidi kao drugi argument, pored naziva programa), i da, samo ukoliko je vrednost celobrojnog argumenta veća od 0, kreira proces-dete nad istim programom i sa za jedan manjom vrednošću svog celobrojnog argumenta, potom ispiše svoju vrednost argumenta i sačeka da se proces-dete završi, a onda se i sam završi. Proces-dete radi to isto (i tako rekurzivno).

Napisati ispravnu implementaciju ovog programa.

```
void main (int argc, char* const argv[]){
  if (argc<2) return; // Exception!

  // Get the argument value:
  int myArg = 0;
  sscanf(argv[1],"%d",&myArg);
  if (myArg<=0) return;

  // Prepare the arguments for the child:
  char childArg[10];  // The value of the second argument
  sprintf(childArg,"%d",myArg-1);
  char* childArgs[3];
  childArgs[0] = argv[0];
  childArgs[1] = childArg;
  childArgs[2] = NULL;

  // Create a child:
  execvp(argv[0],childArgs);
  printf("%s\n",myArg);
  wait(NULL);
}
```

argc → 1
argv[0]
argv[1]

sscanf(string, " ", ...)
string
file fscanf(file,
sprintf(string, "%d", x);
string
fprintf

pid_t pid = fork();
if (pid == 0) {
    execvp(argv[0], childArgs);
}

wait(pid);
int *

3. (10 poena)

U nekom operativnom sistemu postoje sledeći sistemski pozivi:

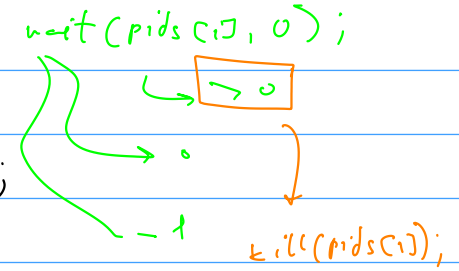
- `fork(), execlp(const char*)`: kao u sistemu Unix i njemu sličnim;
- `int wait(int pid, unsigned timeout)`: suspenduje pozivajući roditeljski proces dok se ne završi proces-dete sa zadatim PID, ali ga suspenduje najduže onoliko koliko je zadato drugim argumentom (vreme čekanja u milisekundama). Ako je vrednost prvog argumenta NULL, pozivajući proces se suspenduje dok se ne završe sva njegova deca (ili ne istekne vreme čekanja); ako je vrednost drugog argumenta 0, poziv odmah vraća rezultat, bez čekanja. Vraćena vrednost 0 označava da su svi procesi koji su se čekali završili (pre isteka vremena čekanja); vraćena vrednost veća od 0 znači da je vreme čekanja isteklo pre nego što je neki od procesa koji su se čekali završili.
- `int kill(int pid)`: gasi proces sa zadaitm PID.

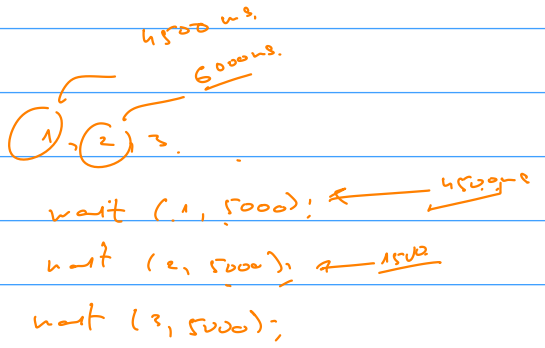
Svi ovi sistemski pozivi vraćaju negativan kod greške u slučaju neuspeha.

Na jeziku C napisati program koji se poziva sa jednim argumentom koji predstavlja stazu do *exe* fajla. Ovaj program treba da kreira *N* procesa koji izvršavaju program u *exe* fajlu zadatom argumentom (*N* je konstanta definisana u programu), a potom da čeka da se svi ti procesi-deca završe u roku od 5 sekundi. Sve procese-decu koji se nisu završili u tom roku treba da ugasi. Obraditi sve greške u sistemskim pozivima ispisom odgovarajuće poruke.

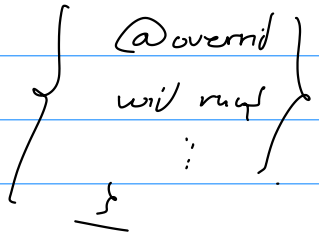
```
#define N ....

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

    for (int i=0 ; i<N; i++){
        pids[i] = fork();
        if (pids[i]<0) {
            printf ("fork fail\n");
            exit(1);
        }.

        if (pids[i] ==0) {
            execlp( argv[1]);
            printf("execlp fail\n");
            exit(1);
        }.
    }.

    int ret = wait (NULL, 5000);
    if (ret<0){
        printf ("wait failed\n");
        exit(1);
    }.
```

```
if (ret > 0) {
    for (int i=0; i<N; i++) {
        int ret = kill(pids[i]);
        if (ret < 0) {
            printf("kill failed\n");
            exit(1);
        }
    }
}
```





JAVE



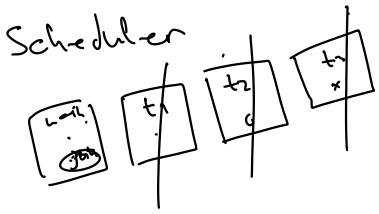
3. (10 poena)

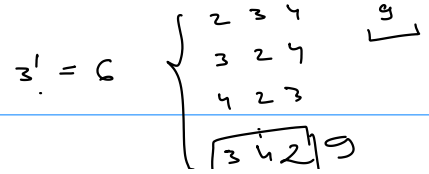
U standardnom jeziku C++ postoje, između ostalog, sledeći elementi podrške za niti:

- `std::thread`: klasa kojom se predstavljaju niti; nova nit se pokreće odmah po kreiranju nekog objekta ove klase;
- `std::thread::thread`: konstruktor klase `std::thread` koji kreira nit nad funkcijom na koju ukazuje prvi argument; ta funkcija prima jedan argument i ne vraća rezultat; novokreirana nit poziva datu funkciju sa stvarnim argumentom jednakim drugom argumentu konstruktora;
- `std::thread::join`: suspenduje pozivajuću nit dok se ne završi nit čija se ova funkcija poziva;
- `thread_local`: specifikator životnog veka i načina alokacije koji se navodi u deklaraciji objekta; objekti ove kategorije životnog veka alociraju se kad god se kreira nova nit i nestaju sa završetkom te niti; svaka kreirana nit, uključujući i „glavnu“ nit kreiranu nad pozivom funkcije `main`, ima svoju sopstvenu kopiju (instancu) ovog objekta, različitu od instanci tog objekta u drugim nitima.

Precizno navesti šta sve može da ispiše sledeći C++ program (izlazne operacije ispisa na standardni izlaz su atomične):

```
thread_local int i=0;

void f (int id) {
  i=id;
  ++i;
  std::cout<<i;
}

void main() {
  i=9;
  std::thread t1(f,1);
  std::thread t2(f,2);
  std::thread t3(f,3);
  t1.join();
  t2.join();
  t3.join();
  std::cout<<i<<std::endl;
}
```





3. (10 poena)

U nastavku je data donekle izmenjena i pojednostavljena specifikacija sistemskog poziva iz standardnog *POSIX thread* API (*Pthreads*):

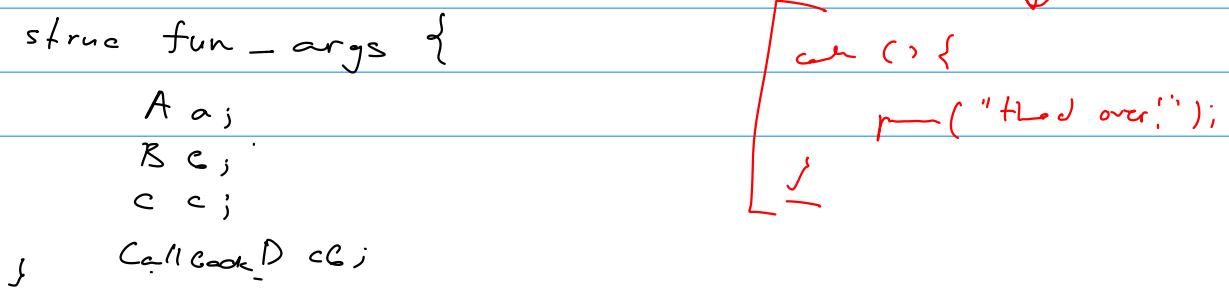
`int pthread_create(pthread_t *thread, void *(*routine)(void *), void *arg)`: kreira novu nit nad funkcijom na koju ukazuje `routine`, dostavljajući joj argument `arg`; identifikator novokreirane niti smešta u loakciju na koju ukazuje `thread`; vraća negativnu vrednost u slučaju greške, a 0 u slučaju uspeha.

Dat je potpis funkcije `fun`, pri čemu su `A`, `B`, `C` i `D` neki tipovi:

```
extern D fun (A a, B b, C c);
```

Potrebno je realizovati potprogram `fun_async` čiji će poziv izvršiti asinhroni poziv funkcije `fun`, tj. inicirati izvršavanje funkcije `fun` u posebnoj niti i potom odmah vratiti kontrolu pozivaocu. Po završetku funkcije `fun`, ta nit treba da pozove povratnu funkciju na koju ukazuje argument `cb` (engl. *callback*) i da joj dostavi vraćenu vrednost iz funkcije `fun`. Ignorisati greške.

```
typedef void (*CallbackD)(D);
void fun_async (A a, B b, C c, CallbackD cb);
```

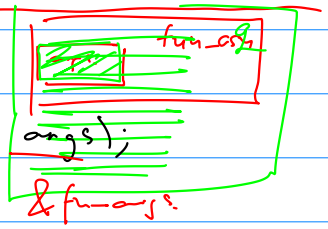


```
void * fun_wrapper(void *arg) {
    struct fun_args args = *((struct fun_args) arg);
    D ret = fun(args.a, args.b, args.c);
    args.callbackD(D);
    free(args);
}
```

delete.

```
void fun_async(A a, B b, C c, callbackD cb) {
    struc fun_args *args = malloc(sizeof(struc fun_args));
    args->a = a;
    args->b = b;
    args->c = c
    args->callbackD = callbackD.
    pthread_t tid;
    pthread_create(&tid, fun_wrappe, args);
}
```

new



3. (10 poena)

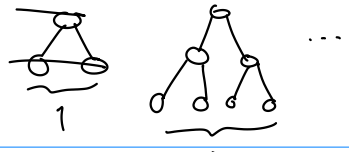
U nastavku je data donekle izmenjena i pojednostavljena specifikacija dva sistemska poziva iz standardnog *POSIX thread* API (*Pthreads*). Obe ove funkcije vraćaju negativnu vrednost u slučaju greške, a 0 u slučaju uspeha.

- `int pthread_create(pthread_t *thread, void *(*start_routine)(void *), void *arg)`: kreira novu nit nad funkcijom na koju ukazuje `start_routine`, dostavljajući joj argument `arg`; identifikator novokreirane niti smešta u lokaciju na koju ukazuje `thread`;
- `int pthread_join(pthread_t thread, void **retval)`: čeka da se nit identifikovana sa `thread` završi (ukoliko se već završila, odmah vraća kontrolu); ako `retval` nije `NULL`, kopira povratnu vrednost funkcije `start_routine` koju je ta nit izvršavala u lokaciju na koju ukazuje `retval`.

Data je struktura `Node` koja predstavlja jedan čvor binarnog stabla. Korišćenjem datih sistemskih poziva implementirati funkciju `createSubtree` koja rekurzivno kreira binarno stablo dubine `n` i vraća pokazivač na njegov koreni čvor, tako što u istoj niti kreira levo podstablo, a u novokreiranoj niti uporedo kreira desno podstablo. Ignorisati eventualne greške.

→ tipa

```
struct Node {
  Node () : leftChild(NULL), rightChild(NULL) {}
  Node *leftChild, *rightChild;
  ...
};

Node* createSubtree (int n);
```




```
void * createSubtreeWrapper (void * arg) {
    int n = *((int *) arg);
    Node * root = createSubtree(n);
    return (void *) root;
}
```

```
Node * createSubtree(int n) {
    if (n == 0) return new Node();
    int arg = n-1;
    pthread_t tid;
    pthread_create(&tid, createSubtreeWrapper, &arg);
    Node * left = createSubtree(n-1);
    Node * right;
    pthread_join(tid, (void**)&right);
    Node * root = new Node();
    root->leftChild = left;
    root->rightChild = right;
    return root;
}
```

4. (10 poena)

U nekom sistemu svi sistemski pozivi vrše se softverskim prekidom broj 44h, pri čemu u registru `r0` sistem očekuje identifikator sistemskog poziva. U ovom sistemu postoji sistemski poziv za kreiranje i pokretanje niti nad potprogramom koji prihvata jedan argument tipa pokazivača. Ovaj poziv u registrima očekuje sledeće argumente:

- u `r0` treba da bude identifikator ovog poziva, 0 u ovom slučaju;
- u `r1` treba da bude adresa potprograma nad kojim se kreira nit;
- u `r2` treba da bude argument potprograma nad kojim se kreira nit.

Svaki sistemski poziv vraća rezultat u registru `r0`, a za ovaj sistemski poziv rezultat je identifikator niti u jezgru (ceo broj veći od 0), odnosno kod greške (ceo broj manji od 0).

U asemblerskim blokovima unutar C/C++ koda može se koristiti simbolička konstanta sa nazivom formalnog argumenta funkcije unutar koje se nalazi dati asemblerski kod; ova simbolička konstanta ima vrednost odgovarajućeg pomeraja lokacije u kojoj se nalazi dati formalni argument u odnosu na vrh steka (kao što je pokazano u primerima na predavanjima). C/C++ funkcije vraćaju vrednost u registru `r0` ukoliko je tip povratne vrednosti odgovarajući.

a)(5) Implementirati funkciju

```
int create_thread(void (*f)(void*), void* arg);
```

koja vrši opisani sistemski poziv (kreira nit nad funkcijom `f` sa argumentom `arg`) i koja može da se poziva iz C koda sa zadatim argumentima. Ova funkcija vraća identifikator kreirane niti koji se može koristiti u ostalim sistemskim pozivima da identifikuje tu nit u jezgru, odnosno kod greške.

b)(5) Korišćenjem prethodne funkcije, implementirati funkciju `start` klase `Thread`. Ova klasa obezbeđuje koncept niti u objektnom duhu, poput onog u školskom jezgru (kreira nit nad virtuelnom funkcijom `run`).

```
class Thread {
public:
  Thread () : pid(0) {}
  int start ();
  virtual void run() {}
private:
  int pid;  // Thread ID
};
```

```
class Worker : public Thread {
    void run() {
        ...
    }
};
Worker worker;
worker.start();
```

```
int create_thread ( void (*f)(void *), void *arg){
    asm {
        li   r0, #0
        ld   r1, #f(sp)
        ld   r2, #arg(sp)
        int 44h.
    }
}
```

class Worker : pub Thread.

```
void wrapper (void *arg) {
    ((Thread *)arg) -> run();
}.
```

```
radias Thread {.
    int start() {.
        pid = create_thread (wrapper, this );
        return pid;
    }.
```