



Programiranje za UNIX

Okruženje unix procesa

Sadržaj

- **Memorijska slika UNIX procesa**
- **Argumenti naredbenog retka i varijable okruženja**
- **Životni ciklus procesa**
- **Izlazni status procesa**



Memorijska slika UNIX procesa

high address

Argumenti naredbenog retka
Environment varijable

Stack

Heap

Neinicijalizirani podaci

Inicijalizirani podaci

Text – izvršni kod programa

low address



Memorijska slika unix procesa

➤ Tekst

- strojni kod programa,
- read-only, može ga dijeliti više procesa

➤ Inicijalizirani podaci

- inicijalizirane globalne varijable

➤ Neinicijalizirane podaci (BSS)

- **B**lock **S**tarted by **S**ymbol - neinicijalizirane globalne varijable

➤ Stack (stog)

- okviri funkcijskih poziva

➤ Heap (hrpa)

- dinamička alokacija memorije

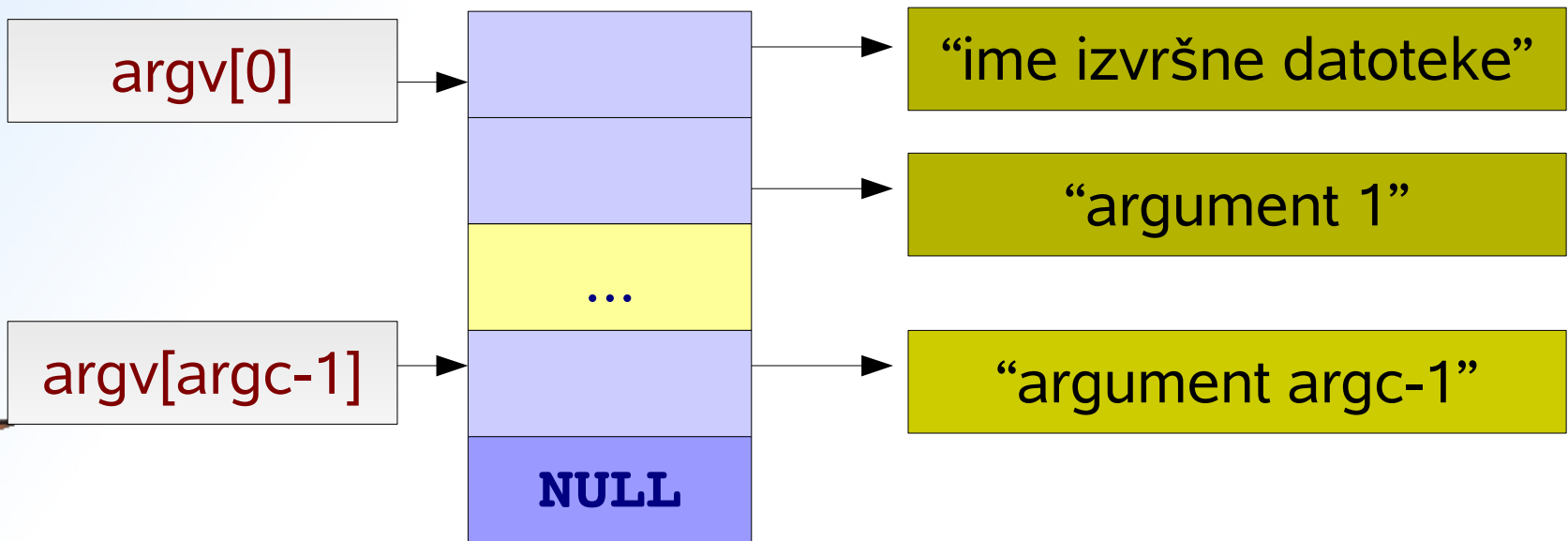
➤ Argumenti naredbenog retka i okruženje procesa



Argumenti naredbenog retka

➔ Primaju se kao argumenti funkcije **main**

- **int argc** – broj argumenata
- **char **argv** – polje pokazivača

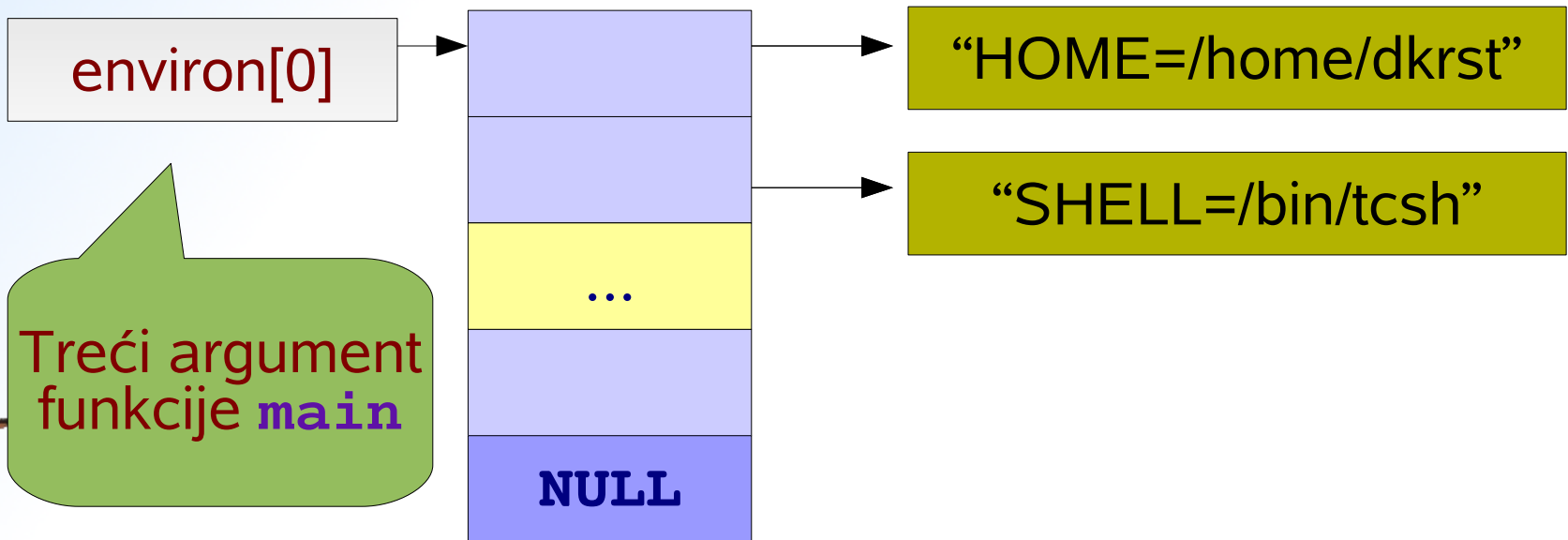


Environment variable

➔ Prosljeđuju se procesu u obliku ime=vrijednost parova

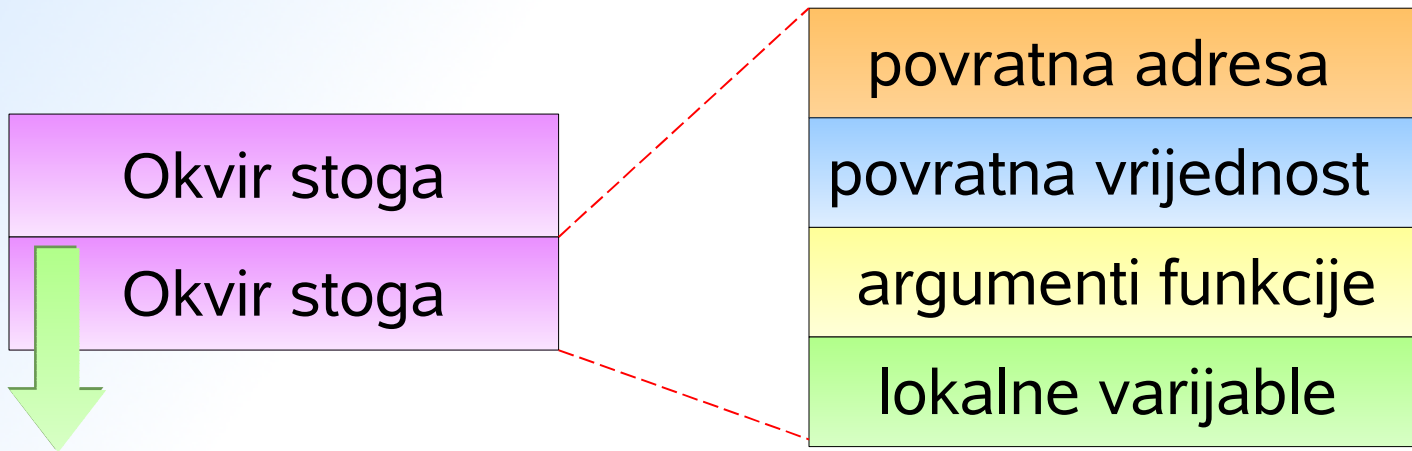
- Funkcije: `getenv()`, `putenv()`, `setenv()`, `unsetenv()`, `clearenv()`

`char **environ` – Nije dio POSIX standarda!



Funkcijski okviri stoga

➔ Okvir stoga kreira se kod svakog poziva funkcije:



➔ Kako rekurzivni funkcijski pozivi mogu utjecati na stog?

- Svaki novi rekurzivni poziv stvara novi okvir na stogu!



Repno rekurzivne funkcije - Tail recursive functions

- Loše napisana rekurzivna funkcija povećava stog linearno sa brojem rekurzivnih poziva!
- Što ako je rekurzivni poziv zadnja operacija u tijelu funkcije?



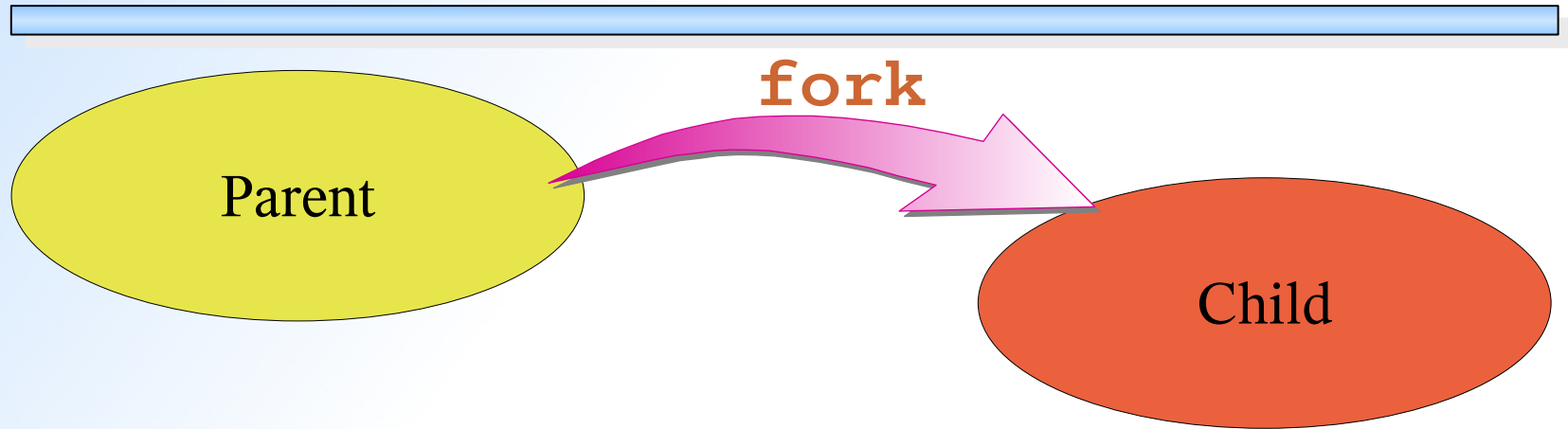
Repno rekurzivne funkcije - Tail recursive functions

- Loše napisana rekurzivna funkcija povećava stog linearno sa brojem rekurzivnih poziva!
- Što ako je rekurzivni poziv zadnja operacija u tijelu funkcije?
 - Nakon poziva ne izvršava se procesiranje
 - Ne koriste se lokalne varijable
 - Ne koriste se argumenti funkcije

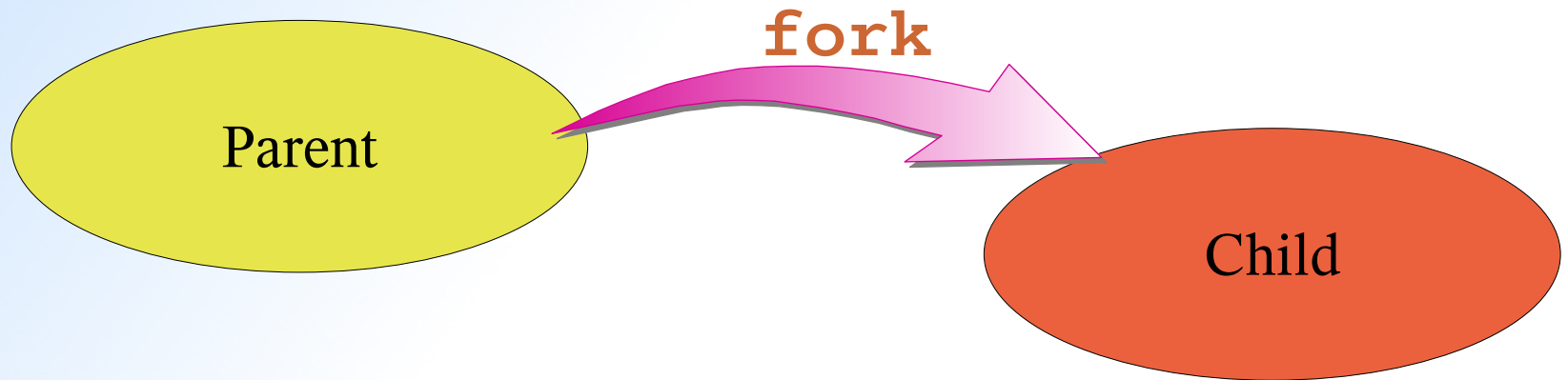
Kompajler može za funkcijski poziv koristiti postojeći okvir stoga!



Stvaranje procesa



Stvaranje procesa



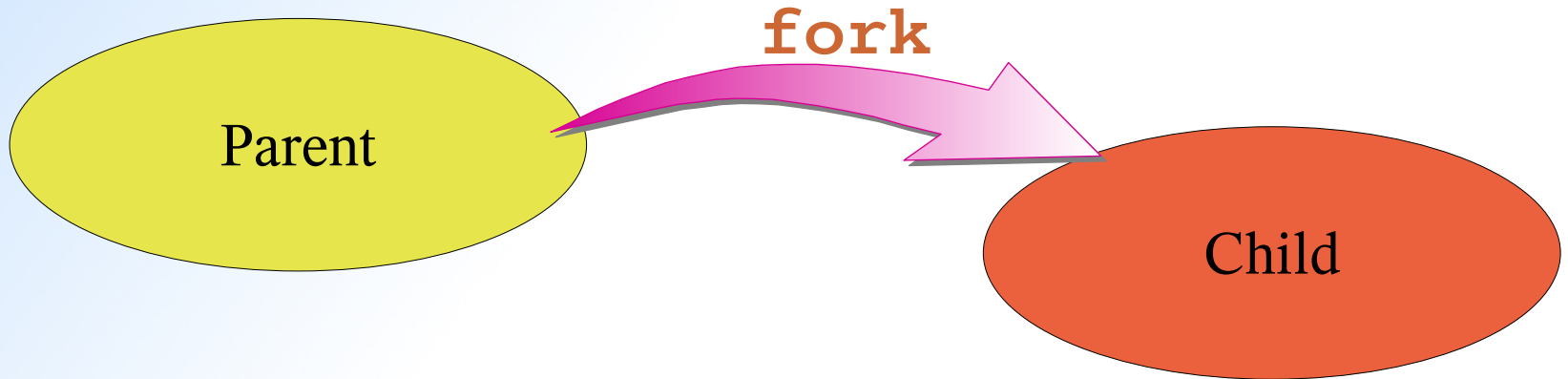
➤ Child je identična kopija parrent procesa, izvršavaju se nezavisno

➤ Povratna vrijednost funkcije fork:

- Parrent: **PID child procesa**
- Child: **0**
- **-1 u slučaju greške (samo u parent procesu, child u ovom slučaju ne postoji!)**



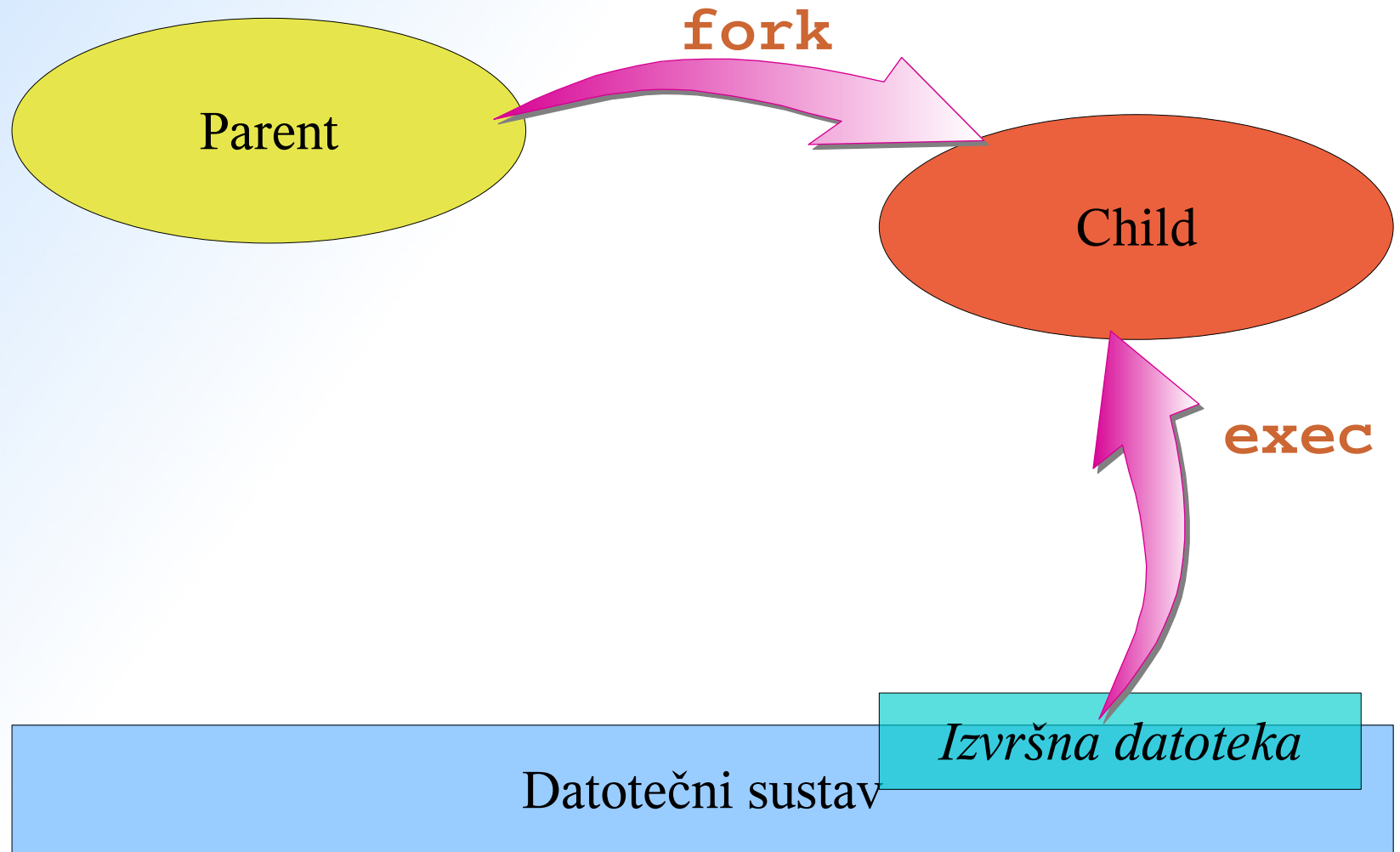
Zamjena izvršnog koda procesa - pokretanje novog programa



Datotečni sustav



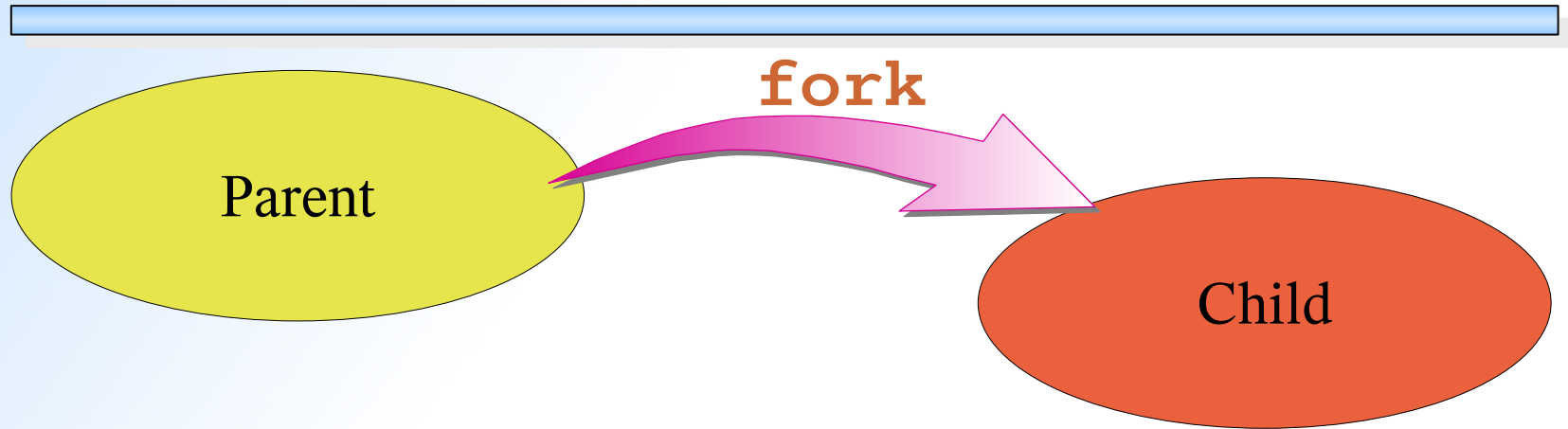
Zamjena izvršnog koda procesa - pokretanje novog programa



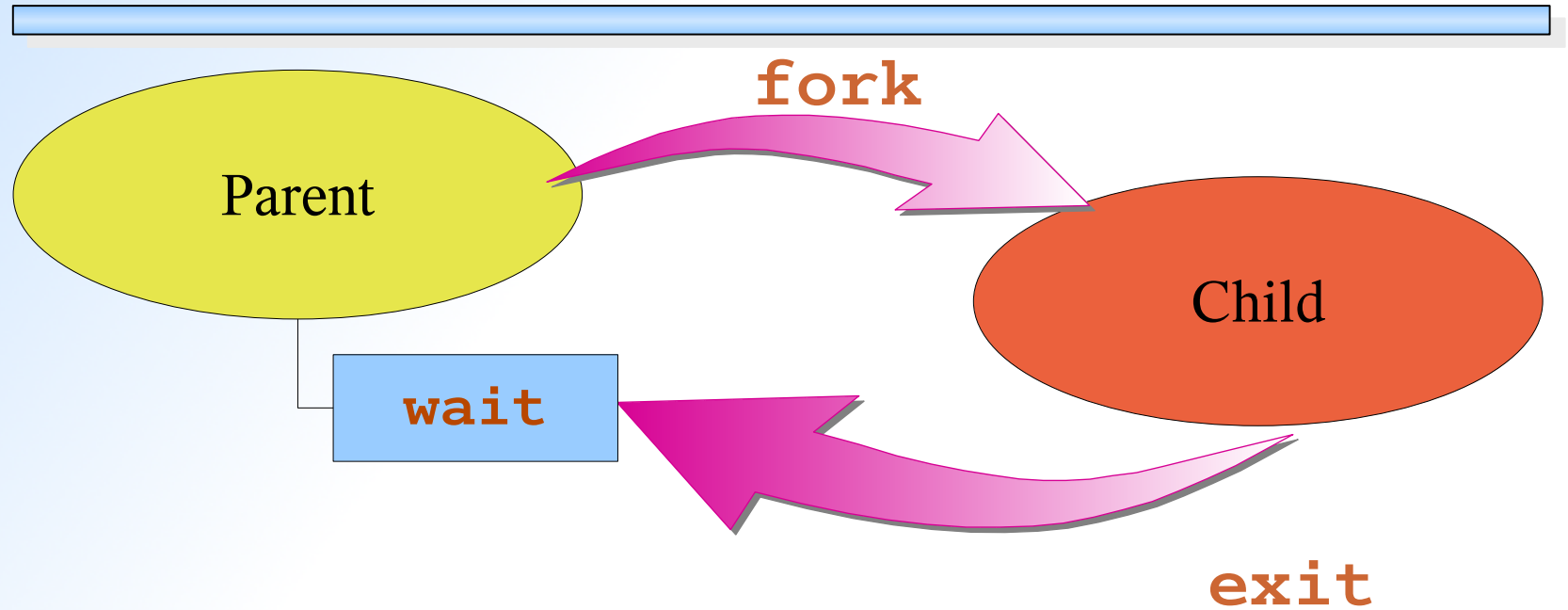
Zamjena izvršnog koda procesa - pokretanje novog programa



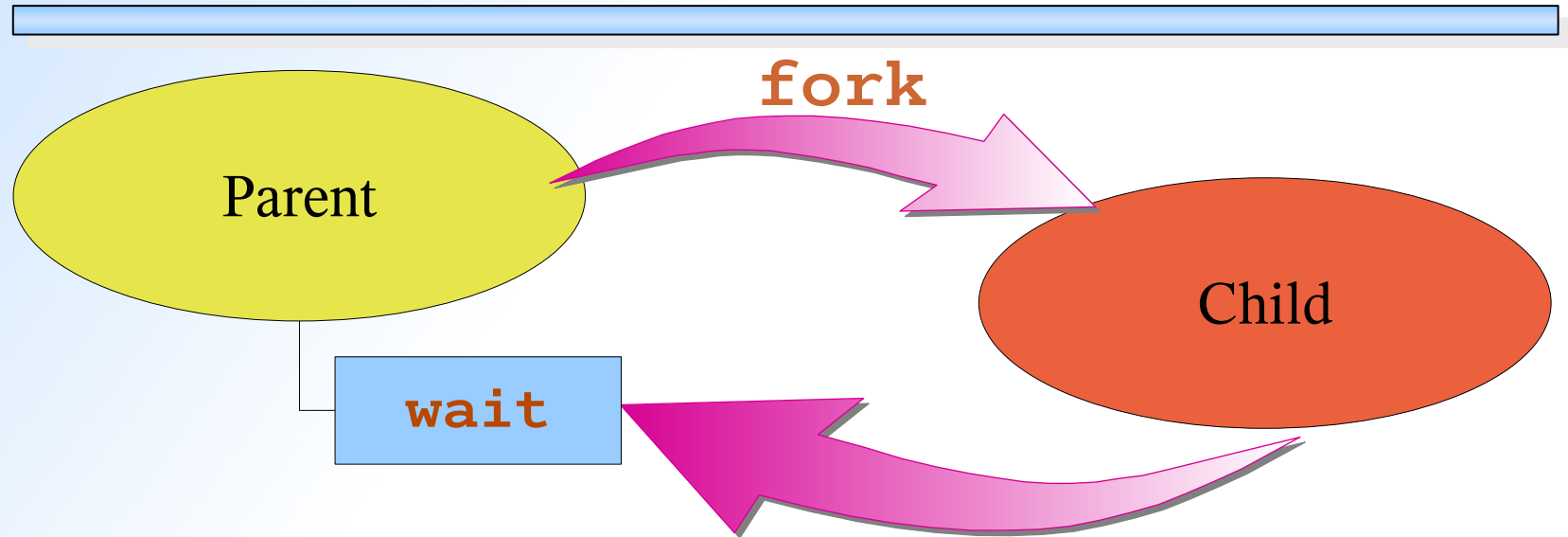
Prekid izvršavanja procesa



Prekid izvršavanja procesa



Prekid izvršavanja procesa



➤ Funkcija **exit** prekida izvršavanje procesa

- Argument funkcije je izlazni status procesa

➤ Izlazni status može pokupiti samo parent proces pozivom funkcije **wait** ili **waitpid**



Mogući uzroci prekida procesa

➤ Normalan završetak (**normal termination**)

- **return** iz funkcije **main** - ekvivalentno pozivu funkcije **exit**
- **exit** - uobičajeni prekid uz pozivanje **exit handlera**
- **_exit, _Exit** - trenutni prekid izvršavanja (uz zatvaranje otvorenih file deskriptora)

➤ Neočekivani izlaz (**abnormal termination**)

- **abort** – svi otvoreni ulazno/izlazni tokovi se prazne i zatvaraju - **dobrovoljno!**
- prekid signalom



fork

➔ **Sistemi poziv, poziva se jednom, a vraća dva puta: klonira postojeći proces**

- U slučaju greške, ne kreira se **CHILD** proces, povratna vrijednost **-1**
- Oba procesa nastavljaju izvršavanje sa prvom slijedećom instrukcijom

```
#include <sys/types.h>
#include <unistd.h>

pid_t fork (void);
```

CHILD process dobiva:

- kopiju adresnog prostora **PARENT** procesa (**copy on write** – segment se kopira kada ga **CHILD** proces adresira)
- **Procesi dijele tekst segment**



exit

➤ Normalan prekid izvršavanja procesa

- Izvršavaju se funkcije (**exit handleri**) registrirane sa **atexit** i **on_exit**
- Svi ulazno/izlazni tokovi se prazne (**flush**) i zatvaraju

➤ Parametar funkcije **exit** je **izlazni status** procesa

```
#include <stdlib.h>
void exit (int status);
```



wait, waitpid

➤ Čekanje na promjenu stanja **CHILD** procesa

- Ukoliko za proces koji je završio izvođenje nije primijenjen wait poziv, nastaje tzv. **zombi** proces
- Kernel očekuje da netko pokupi izlazni status procesa

➤ Kada proces završi, njegov **PARENT** proces dobiva **SIGCHLD** signal

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

pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);
```



exec

➔ Mijenja memorijsku sliku procesa

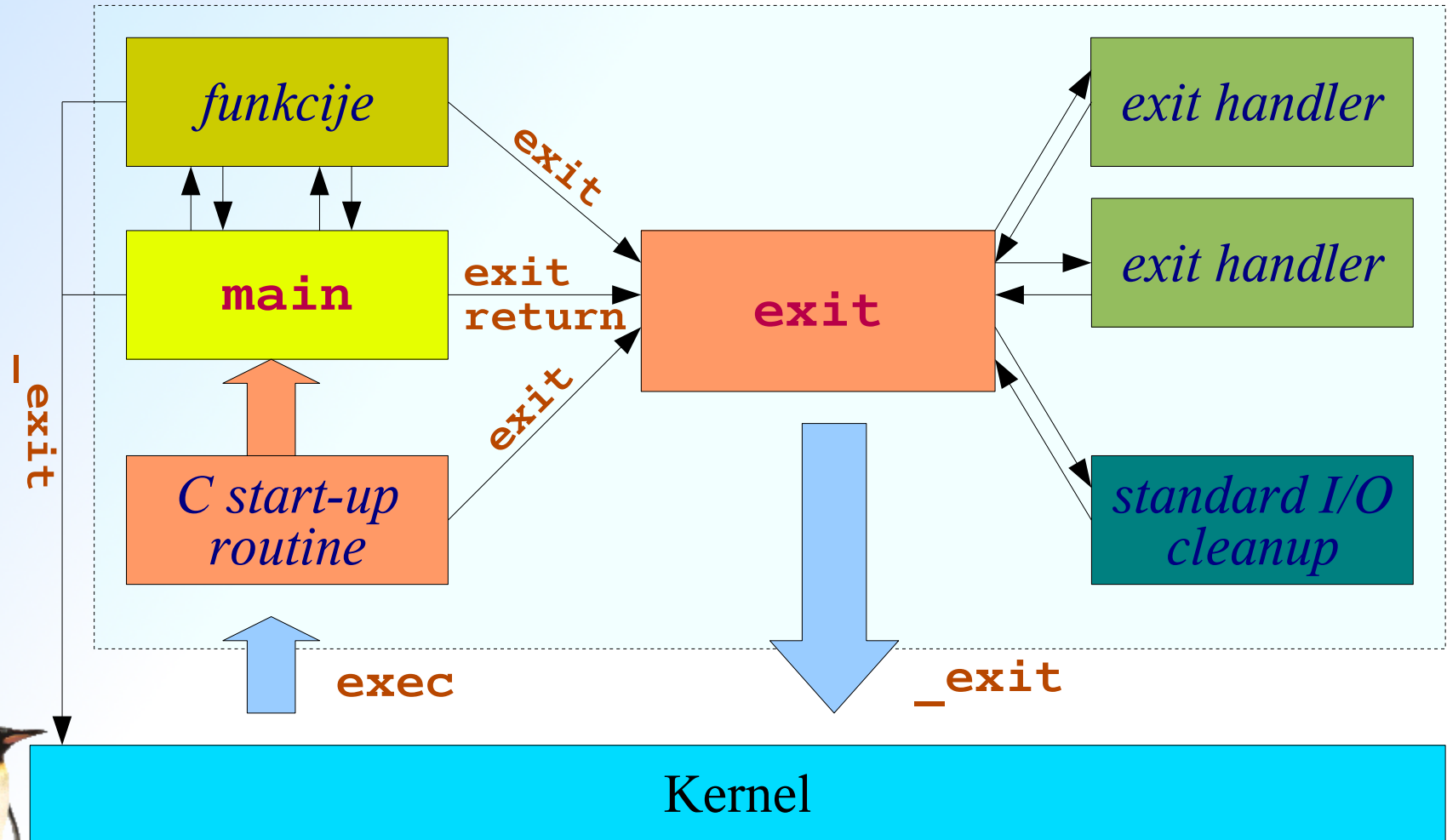
- Mijenjaju se tekst segment, inicijalizirani i neinicijalizirani (BSS) segment podataka, argumenti naredbenog retka, environment i stog
- Nasljeđuje se **PID** i **svi otvoreni file deskriptori**

```
#include <unistd.h>

int execl(const char *path, const char *arg, ...);
int execlp(const char *file, const char *arg, ...);
int  exect(const char *path, const char *arg, \
          ..., char *const envp[]);
int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);
int execve(const char *filename, \
          char *const argv[], char *const envp[]);
```



Životni ciklus izvođenja programa



Izlazni status procesa

➔ Završetkom procesa oslobađaju se njegovi memorijski segmenti, **ali se čuvaju podaci u tablici procesa**

- podaci se čuvaju sve dok parent proces ne pokupi izlazni status
- **ukoliko parent ne pozove wait nastaje zombi proces!**

➔ Što ukoliko **parent** proces pozove **exit** prije **child** procesa?

- **Ako parent proces završi prije svojih child procesa, svim child procesima se dodjeljuje novi roditelj: proces init sa proces ID 1**

➔ **init prima signal SIGCHLD za svaki završeni child proces i poziva wait**



Zadatak: Razbijanje veze parent-child

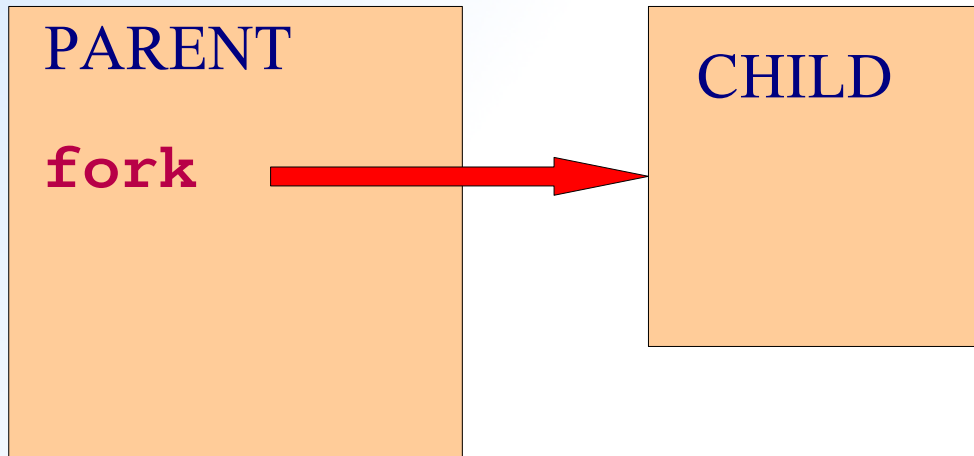
➡ Može li parent proces “prepustiti” svoj child proces bez da završi rad?

- Napišite program koji stvara proces čiji je parent proces **INIT (process ID 1)**, a da pri tom sam proces koji je novi proces stvorio **nastavi s radom**



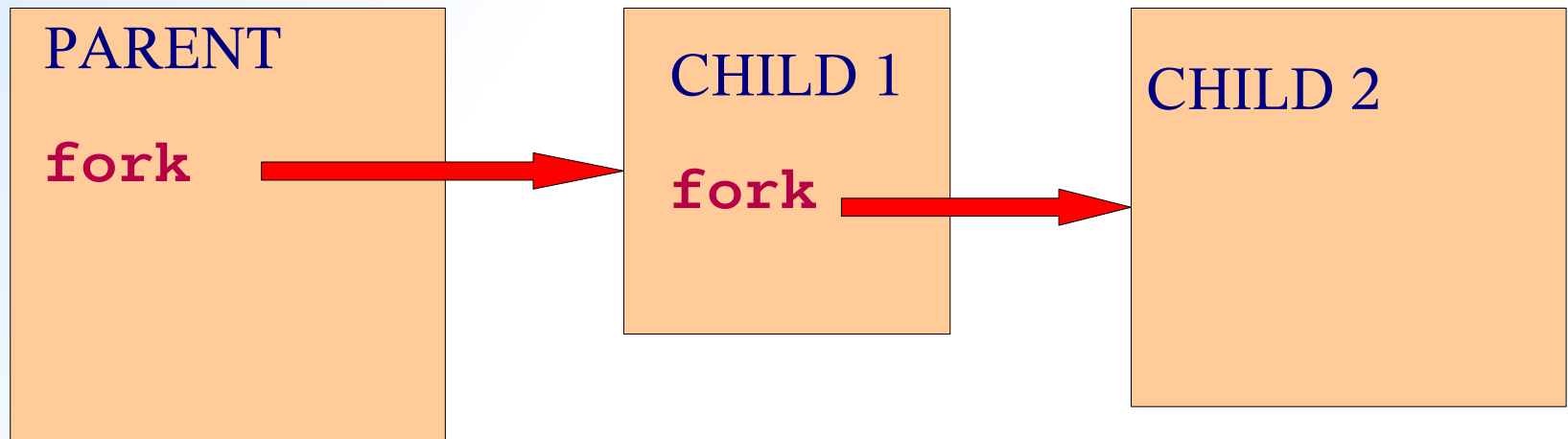
Razbijanje veze parent-child

➔ **PARENT** proces stvara novi proces
pozivom funkcije **fork**



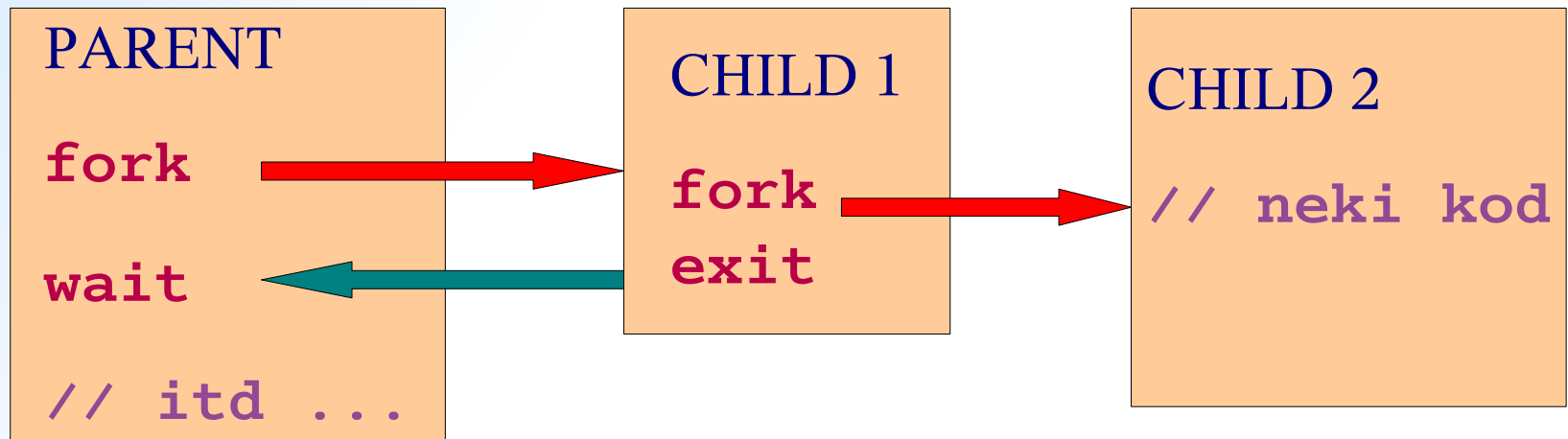
Razbijanje veze parent-child

➤ **CHILD 1** proces poziva **fork** – stvara se proces čiji je **PARENT** prvi child **CHILD 1**



Razbijanje veze parent-child

➡ **PARENT** poziva `wait`, **CHILD 1** poziva `exit`



Razbijanje veze parent-child

➔ **CHILD 2** nasljeđuje **INIT**

PARENT

`fork`

`wait`

`// itd ...`

~~CHILD 1~~

~~`fork`~~

~~`exit`~~

CHILD 2

`// neki kod`

`init`



Razbijanje veze parent-child

➡ Kada novi proces (**CHILD 2**) pozove **exit**, signal dobiva **INIT** proces

PARENT

```
fork
```

```
wait
```

```
// itd ...
```

CHILD 2

```
// neki kod
```

```
exit
```



init

signal handler: **wait**



Limiti

➤ Postavljeni za svaki proces

- RLIM_AS, RLIMIT_CPU, RLIMIT_DATA, RLIMIT_FSIZE, RLIMIT_NPROC, RLIMIT_STACK

➤ Vrijednost RLIM_INFINITY isključuje limit na danom resursu

```
#include <sys/time.h>
#include <sys/resource.h>

int getrlimit(int resource, struct rlimit *rlim);
int setrlimit(int resource, \
               const struct rlimit *rlim);

struct rlimit {
    rlim_t rlim_cur;    /* Soft limit */
    rlim_t rlim_max;    /* Hard limit */
};
```



Tablica procesa - Process Table

➔ **Struktura u jezgri, sadrži informacije o svim procesima**

- **Osnovni podaci:** PID, parent PID, UID, GID, ...
- **Signali:** informacije o statusu svih signala
- **Memorija:** pokazivači na sve memorijske segmente procesa
- **Informacije za upravljanje procesom (Scheduling Parameters):** prioritet, nice value, CPU time, sleep time, ...



Informacije o procesu

➔ Prikupljanje osnovnih informacija o procesu:

- proces ID
- parent proces ID
- real user ID
- effective user ID
- real group ID
- effective group ID

```
#include <unistd.h>
#include <sys/types.h>

pid_t getpid(void);
pid_t getppid(void);

uid_t getuid(void);
uid_t geteuid(void);

gid_t getgid(void);
gid_t getegid(void);
```

