

OS Vezbe

I. cas

Podsecanje iz P1 i P2, i ceste greske na ispitima.

- Neinicijalizovani pokazivac
- Stringovi bez terminirajuće nule kada se ručno radi sa stringovima
 - Ručno postavljanje nule na kraj stringa je obavezno da funkcije za ispis ne bi citale smeće iz memorije
 - Svaka funkcija iz `string.h` automatski stavlja terminirajuću nulu na kraj stringa
- Definisanje makroa za proveru greska koji će biti često korišćen tokom kursa.

```
define check_error(cond, msg)\
    do{\
        if(!(cond)){\
            perror(msg);\
            exit(1);\
        }\
    } while(0)
```

- Ovaj makro se koristi tako što se kao cond stavi uslov koji ne sme da bude true, na primer kod rada sa File Descriptorima kada proveravamo `check_error(fd != -1)`, u slučaju da je fd -1 mi izbacujemo gresku. Takođe je jako bitno da koristimo `errno.h` zaglavlje koje nam omogućava detaljniji uvid u to šta je tačno greska.

⚠ OBAVEZNO ✓

Pokazivace uvek inicijalizovati na NULL

Uvek nakon alokacije proveriti da li je alokacija uspesna

Uvek zatvoriti otvorene fajlove i fajl dekriptore

Uvek osloboditi alociranu memoriju nakon što ona prestane da se koristi

🔗 TLPI poglavlja

- 3.4 Handling errors from Sys Calls

2. cas

Dobijanje informacija o korisniku, grupi i fajlu

Korisnici

- Na linux sistemima sve informacije o korisniku se cuvaju u /etc/passwd fajlu. Dok se informacije o grupi cuvaju u /etc/group
- Za informacije o korisniku moram da `#include <pwd.h>` zaglavlje koje omogucava citanje /etc/passwd fajla, zbog sigurnosnih razloga korisnicke lozinke odnosno hashevi isti su smesteni u /etc/shadow koji zahteva specijalna prava pristupa da bi se otvorio i citao.
- Da bi dobili informacije o korisniku moramo da koristimo staticku strukturu `struct passwd *userInfo`, `userInfo` je samo naziv strukture u ovom slucaju dok nju popunjavamo sistemskim pozivom `getpwnam()` koja kao argument prima niz karaktera koji je korisnicko ime u plain textu, postoji i druga funkcija koja vraca istu strukturu samo prima korisnicki id `uid_t` (User ID) i ona je `getpwuid()`.
- Za citanje celog /etc/passwd fajla koristimo funkcije `setpwent()` i `endpwent()` koje otvaraju i citaju i nakon toga zatvaraju /etc/passwd file.
- Trenutnog korisnika dobijamo tako sto pozovemo `getpwent()` i fajl citamo do kraj dok funkcija ne vrati `null`.

```
setpwent();
struct passwd *user = NULL;

while((user = getpwent()) != NULL){

    print_users(user);
}
endpwent();
```

- Primer kako bi kod koji cita ceo fajl trebao da izgleda.

```
void print_users(struct passwd* userInfo){

    fprintf(stdout, "\n");
    fprintf(stdout, "Username: %s\n", userInfo->pw_name);
    fprintf(stdout, "UID: %d\n", userInfo->pw_uid);
    fprintf(stdout, "GID: %d\n", userInfo->pw_gid);
    fprintf(stdout, "Home dir: %s\n", userInfo->pw_dir);
    fprintf(stdout, "Shell: %s\n", userInfo->pw_shell);
}
```

- Funkcija koja ispisuje korisnike u formatu koji je citljivi korisniku

Grupe

- Grupe se obradjuju na isti nacin kao i korisnici samo sto se za gurpe koristi zaglavlje `grp.h`
- Informacije o grupi takodje mozemo dobijati na osnovu imena grupe i njenog ID-a koriscenjem funkcija `getgrnam()` i `getgrgid()` koje kao parametre primaju niz karaktera (ime grupe), odnosno `gid_t` sistemski tip koji je Group ID
- Podaci o grupi se cuvaju takodje u staticki alociranoj strukturi `struct group*` `groupInfo` gde je `group` tip structure a `groupInfo` samo njen naziv.
- Ona se popunjava gore navedenim funkcijama.
- Kao kod `pwd.h` zaglavlja i `grp.h` omogucava citanje celog `/etc/group` fajla sa funckijama `setgrent()`, `getgrent()` i `endgrent()`

```
setgrent();
struct group *currentGroup = NULL;

while((user = getgrent()) != NULL){

    print_group(currentGroup);
}
endgrent();
```

- Ovaj kod prolazi kroz ceo `/etc/group` fajl i ispisuje njegov sadrzaj onako kako mi odlucimo da ga formatiramo u fukciji `print_group()`

```
void print_groups(struct group *grinfo){

    fprintf(stdout, "\n");
    fprintf(stdout, "%s\n", grinfo->gr_name);
    fprintf(stdout, "%d\n", grinfo->gr_gid);

    for(int i = 0; grinfo->gr_mem[i] != NULL; i++){
        fprintf(stdout, "%s\n", grinfo->gr_mem[i]);
    }

}
```

- Ova fukcija za dati argument `*grinfo` ispisuje ime grupe, njen id i sve clanove koji su u toj grupi

Fajlovi

- Na linux sistemima postoji 7 vrsta fajlova;
 1. regularni fajlovi -

2. direktorijumi `d`
 3. blok fajlovi, oni su obicno hardverske komponente i nalaze su `/dev` direktorijumu `b`
 4. char fajlovi, ovo su ulazni i izlazni streamovi, terminal je jedan od char fajlova `c`
 5. pipe-ovi, ovo su fajlovi koji sluze za redirekciju izlaza jednog programa u ulaz drugog `p`
 6. simbolicki linkovi, oni sadrze stvarne putanje do nekog fajla `l`
 7. Socketi, koriste se za komunikaciju izmedju aplikacija `s`
- Sve informacije o fajlu mozemo dobiti pomocu sistemskog poziva `stat()`, koji kao argumente prima putanju do fajla i pokazivac na strukturu `struct stat fileinfo` gde je `fileinfo` samo naziv.
 - Iz `stat` funkcije mozemo da dobijemo informacije kao sto su prava pristupa u formatu `rwxrwxrwx` gde je `r` - read, `w` - write i `x` - executable. i redosled kojim su rasporedjeni oznacava prava korisnika, grupe i ostalih na sistemu
 - Pre ovih 9 karaktera ide jedan karakter koji oznacava tip fajla (jedan od onih 7 sa pocetka).
 - Struktura `stat` takodje sadrzi i informacije o velicini fajla, vremenu nastanka fajla, vremenu pristupa i vremenu modifikacije, `uid_t` korisnika koji je napravio fajl kao i `gid_t` kojoj taj korisnik pripada.

TLPI poglavlja i materijali

- 8.1 - 8.4, I5.1, I5.4-I5.4.3
- man 5 passwd i man 5 group

3. cas

Sistemi pozivi za rad sa fajlovima

Mkdir

- Poziv `mkdir` kreira direktorijum, kao argumente prima putanju na koji fajl treba biti kreiran i mod u kojem treba biti kreiran, odnosno `mode_t` promenljivu koja u sebi sadrzi prava pristupa u hexadecimalnom zapisu, na linuxu se prava pristupa oznacavaju trocifrenim hexadecimalnim brojem.

User	Group	Other
<code>rwX</code>	<code>rwX</code>	<code>rwX</code>
<code>III</code>	<code>IOI</code>	<code>IOI</code>

- Ovo je ekvivalent kao da smo napisali `0755` za prava pristupa, ovo takodje znaci da korisnik moze da cita, pise i izvrsava dok svi ostali mogu samo da citaju i izvrsavaju

Unlink i rmdir

- Brisanje fajlova se vrši pozivom `unlink` koji uklanja fajl sa date putanje. Dok se za brisanje direktorijuma koristi `rmdir`, da bi `rmdir` uspesno radio direktorijum koji brisemo mora da bude prazan, kod neke implementacije mogli bi smo rekurzivno da prodjemo kroz zadati direktorijum pobrisemo sve fajlove sa `unlink` ili `rmdir` ako naidjemo na neki novi direktorijum i zatim obrisemo nas zadati direktorijum.

Otvaranje fajlova

- Svaki otvoren fajl na sistemu ima svoj fajl deskriptor, on nam govori koji je fajl otvoren, sa kojim pravima pristupa i sa kojim flagovima smo otvorili taj fajl odnosno sta mozemo sa njim da radimo
- Za otvaranje fajlova koristimo funkciju `open` kojoj saljemo putanju do naseg fajla, flagove koji odredjuju kako se kreira fajl i prava pristupa.

Flag	Purpose	SUS?
O_RDONLY	Open for reading only	v3
O_WRONLY	Open for writing only	v3
O_RDWR	Open for reading and writing	v3
O_CLOEXEC	Set the close-on-exec flag (since Linux 2.6.23)	v4
O_CREAT	Create file if it doesn't already exist	v3
O_DIRECT	File I/O bypasses buffer cache	
O_DIRECTORY	Fail if <i>pathname</i> is not a directory	v4
O_EXCL	With O_CREAT: create file exclusively	v3
O_LARGEFILE	Used on 32-bit systems to open large files	
O_NOATIME	Don't update file last access time on <i>read()</i> (since Linux 2.6.8)	
O_NOCTTY	Don't let <i>pathname</i> become the controlling terminal	v3
O_NOFOLLOW	Don't dereference symbolic links	v4
O_TRUNC	Truncate existing file to zero length	v3
O_APPEND	Writes are always appended to end of file	v3
O_ASYNC	Generate a signal when I/O is possible	
O_DSYNC	Provide synchronized I/O data integrity (since Linux 2.6.33)	v3
O_NONBLOCK	Open in nonblocking mode	v3
O_SYNC	Make file writes synchronous	v3

- Ovo su flagovi sa kojima otvaramo fajlove.
- Nakon sto smo pozvali funkciju `open` ona vraca ceo broj int koji je nas file deskriptor koji nam omogucava da kasnije pisemo ili citamo otvoreni fajl.
- Fajl deskriptor ne sme da bude -1, u tom slucaju se dogodila greska pri otvaranju fajla.
- Na linux sistemima deskriptori pocinju od broja 3 posto su:
 1. 0 - STDIN - 0 je standardni ulaz
 2. 1 - STDOUT - 1 je standardni izlaz
 3. 2 - STDERR - 2 je standardni izlaz za greske

Read i write

`read` poziv pokušava da cita podatke iz datog file deskriptora `read(int fd, void buf[.count], size_t count);`, `fd` je file deskriptor, `buf` je memorija u koju upisujemo procitanih `count` bajtova fajla, ako `read` vrati `-1` znamo da citanje nije bilo uspesno.

```
{  
    int fd = open(argv[1], O_RDONLY);  
    check_error(fd != -1, "open failed");  
  
    char buf[BUFFER_SIZE];  
    int readBytes = 0;  
  
    while(readBytes = read(fd, buf, BUFFER_SIZE)){  
        check_error(write(1, buf, readBytes) != -1, "write  
failed");  
    }  
  
    check_error(readBytes != -1, "read failed");  
  
    close(fd);  
}
```

- U ovom kodu otvaramo fajl sa `O_RDONLY` flagom koji nam omogućava citanje fajla zatim alociramo niz karaktera `char buf[]` u koji cemo da smesatamo procitane podatke.
- Funkcija `read` vraća broj procitanih bajtova fajla i sve dok broj procitanih bajtova nije 0 nastavljamo da citamo, u slučaju da je `readBytes = -1` to znači da je došlo do greske.
- Ova funkcija cita dati fajl i ispisuje ga na standardni izlaz, slično kao program `cat` u linuxu.

Kopiranje fajlova

```
int srcfd = open(argv[1], O_RDONLY);  
check_error(srcfd != -1, "srcfd");  
  
int destfd = open(argv[2], O_WRONLY | O_TRUNC | O_CREAT, 0644);  
check_error(destfd != -1, "destfd");  
  
char *buf = malloc(BUFF_SIZE);  
check_error(buf != NULL, "buffer");  
  
int readBytes = 0;  
  
while(readBytes = read(srcfd, buf, BUFF_SIZE)){  
    check_error(write(destfd, buf, readBytes) != -1, "write");  
}
```

```

check_error(readBytes != -1, "read");

close(srcfd);
close(destfd);
free(buf);

```

- Kreiramo `int srcFd, int destFd`, koji su fajl dekriptori nasih fajlova `srcFd` je fajl koji kopiramo, `destFd` je fajl u koji kopiramo, zatim inicializujemo bafer za citanje i citamo sve dok `read` ne vrati 0.
- Problem kod ovog pristupa je sto mi ne zadržavamo prava pristupa fajla koji kopiramo nakon sto se kopiranje završi posto je `destFd` otvoren sa razlicitim pravima i on ce uvek imati prava 0644 ili `rw-r--r--` sto znaci da ce samo korisnik moci da cita i pise dok ce ostali moci samo da citaju.

TLPI poglavlja

- 4-4.7
- 18.3 - unlink
- 18.6 rmdir i mkdir
- 15.2-15.2.1

4. cas

Umask i promena prava pristupa

- Umask koristimo da bi dobili prava pristupa koja zelimo posto sistem ne dozvoljava kreiranje fajla sa `0777` pravima na primer. Sistem iz sigurnosnih razloga ne dozvoljava kreiranje fajlova koje mogu svi da citaju, pisu i izvorsavaju pa su default prava pristupa uglavnom `0775` ili `0664`
- Zeljena prava dobijamo kada zeljena prava `&` (and-ujemo) sa invertovanim umaskom.
- Za razliku od `chmod`, `umask` ne radi sa vec postojećim fajlovima vec samo omogucava kreiranje fajla sa zeljenim pravima.
- Komanda `chmod` menja prava datom fajlu `chmod [OPTION] mode ... FILE`

```

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

    check_error(argc == 3, "./chmod file permissions");

    int prava = strtol(argv[2], NULL, 8);

    //kreira se fajl

```

```

int fd = open(argv[1], O_CREAT, prava);
check_error(fd != -1, "open");
close(fd);

//menjaju mu se prava pristupa
check_error(chmod(argv[1], prava) != -1, "chmod");

exit(EXIT_SUCCESS);
}

```

- Fukcija koja emulira sistemsku komandu `chmod` i menja mu prava pristupa, problem sa ovom funkcijom je taj sto mi ne bi smeli i ne bi trebali da otvaramo fajl ako zelimu da mu samo promenimo prava

```

/* izdvajamo stara prava pristupa fajlu*/
struct stat fileinfo;
check_error(stat(argv[1], &fileinfo) != -1, "stat");
mode_t current_mode = fileinfo.st_mode;

/* dodajemo i oduzimamo trazena prava*/
mode_t new_mode = (current_mode | S_IWGRP) & ~S_IROTH;

/* menjaju mu se prava pristupa */
check_error(chmod(argv[1], new_mode) != -1, "chmod");

exit(EXIT_SUCCESS);

```

- Ova funkcija ne otvara fajl ali mu menja prava pristupa tako sto ih prvo dobije iz `stat` strukture. Ova funkcija konkretno oduzima prava citanja za `other` a dodaje prava pisanja za `group`

Obilazak direktorijuma

- Obilazak direktorijuma mozemo uraditi na 2 nacina:
 - tako sto cemo rucno napisati funkciju za rekurzivni prolaz kroz direktorijum
 - koriscenjem `nftw` funkcije iz zaglavlja `<ftw.h>` (file tree walk)
- Rucna implementacija obilaska direktorijuma u dubinu

```

void sizeofDir(char* putanja, unsigned* psize) {
    /* citamo informacije o trenutnom fajlu */
    struct stat fInfo;
    check_error(lstat(putanja, &fInfo) != -1, "...");

    /* dodajemo velicinu fajla na tekuci zbir */
    *psize += fInfo.st_size;
}

```



```

if (!S_ISDIR(fInfo.st_mode)) {
    /* prekidamo rekurziju */
    return;
}

/* ako je u pitanju direktorijum, otvaramo ga */
DIR* dir = opendir(putanja);
check_error(dir != NULL, "...");

/* u petlji citamo sadrzaj direktorijuma */
struct dirent* dirEntry = NULL;
errno = 0;
while ((dirEntry = readdir(dir)) != NULL) {

    char* path = malloc(strlen(putanja) + strlen(dirEntry->d_name) +
2);
    check_error(path != NULL, "...");

    /* formiramo putanju na gore opisani nacin */
    strcpy(path, putanja);
    strcat(path, "/");
    strcat(path, dirEntry->d_name);

    if (!strcmp(dirEntry->d_name, ".") || !strcmp(dirEntry->d_name,
"..")) {
        check_error(stat(path, &fInfo) != -1, "...");
        *psize += fInfo.st_size;

        free(path);
        errno = 0;
        continue;
    }
    sizeofDir(path, psize);
    free(path);

    errno = 0;
}

check_error(errno != EBADF, "readdir");
/* zatvaramo direktorijum */
check_error(closedir(dir) != -1, "...");
}

```

- Drugi nacin je samo koriscenjem funkcije `nftw` tako sto cemo joj posalti samo `const char *path*` putanju do fajla, i pokazivac na funkciju kojom ce da obradi fajlove, `int f(const char *fpath, const struct stat *sb, int typeflag, struct FTW *ftwbuf`, broj otvorenih fajl deskriptora `int nopenfd` i `int statusflag`,

- `int f(const char *fpath, const struct stat *sb, int typeflag, struct FTW *ftwbuf)` ova funkcija je zaduzena za obradjivanje fajla koji funkcija `ntfw` nadje;
 1. `const char *fpath`, je putanja koji `ntfw` prosledjuje ovoj funkciji
 2. `const struct stat *sb`, je stat struktura pomocu koje mozemo da obradjujemo informacije o trenutnom fajlu koje je `ntfw` pronasao
 3. `int typeflag` nam pomaze da brze utvrdimo tip fajla, na primer `FTF_F` znaci da je fajl na putanje `fpath` regularan fajl
 4. `ftwbuf` pruza funkciji uvid u struktur `FTW` koja sadrzi dve promenljive, `int base` i `int level`, `base` oznacava koliko daleko smo odmakli od pocetka nase putanje a `level` koliko smo duboko,
- Implementacija prolaza sa `ntfw`

```
int days = 0;

int filterByTime(const char* fpath, const struct stat* sb, int typeflag,
struct FTW* ftwbuf) {

    time_t now = time(NULL);
    time_t diffInSec = now - sb->st_mtime;

    if (diffInSec/DAY_IN_SEC < days)
        printf("%-80s\n", fpath);

    return 0;
}

int main(int argc, char** argv) {
    check_error(argc == 3, "./filterExt path days");

    /* prebacivanje stringa u broj */
    days = atoi(argv[2]);

    /* provera da li se radi o direktorijumu */
    struct stat fInfo;
    check_error(stat(argv[1], &fInfo) != -1, "stat failed");
    check_error(S_ISDIR(fInfo.st_mode), "not a dir");

    /* obilazak direktorijuma pomocu ugradjene funkcije */
    check_error(ntfw(argv[1], filterByTime, 50, 0) != -1, "ntfw failed");

    exit(EXIT_SUCCESS);
}
```

- Ova funkcija ispisuje sve fajlove koji su modifikovani u poslenjih `days` dana

I5.4.6, I5.4.7
I8.1, I8.2, I8.3, I8.8

🔗 Korisno

- komandom `ln` se mogu kreirati hard linkovi. Komandom `ln -s` se mogu kreirati simbolički linkovi. Može biti korisno zarad testiranja na ispitu.

5. cas

- Ovde je uglavnom sve isto kao na 4. casu, prolazak kroz direktorijum sa `nftw` funkcijom i obrada fajlova

```
~/dev/os/vidoje/cas5 on main > ll -sname
Permissions Size User Date Modified Git Name
-rw-r--r-- 181 tox 9 Nov 13:37 -- README.md
-rw-r--r-- 1.6k tox 9 Nov 13:37 -- 6_zameniPrava.c
-rw-r--r-- 1.7k tox 9 Nov 13:37 -- 5_promeniExt.c
-rw-r--r-- 1.3k tox 9 Nov 13:37 -- 4_printtime.c
-rw-r--r-- 3.2k tox 9 Nov 13:37 -- 3_gettime.c
-rw-r--r-- 1.9k tox 9 Nov 13:37 -- 2_filterExt.c
-rw-r--r-- 2.0k tox 9 Nov 13:37 -- 1_filteByTime_pogresno.c
-rw-r--r-- 1.9k tox 9 Nov 13:37 -- 1_filteByTime.c
```

- Jedina nova stvar je obrada vremena koju me sad mrzi da pisem

🔗 TLPI poglavlja

- I8 - nftw
- IO-IO.2

6. cas

Procesi

- Za rad sa procesima najbitnije su nam funkcije `fork`, `exit`, `wait` i neka od varijacija `exec` funkcije.

- `fork()` sistemski poziv omogućava procesu tj. roditelju da kreira `detete` proces koji je u potpunosti nezavisan on njega i sa kojim nema direktan način komunikacije, ali `detete` proces dobija istu kopiju podataka koje je imao i roditelj (`stack`, `heap`, `data` i `text` segmente iz memorije).
- `exit()` kada pozovemo, proces se prekida i tada su svi resursi koje je on koristio oslobodjeni i prepusteni kernelu da ih on realocira, funkcija `exit()` obično prima kao argument `int EXIT_STATUS` koji ima u glavnom vrednost 0 ili 1, gde 0 označava uspeh a 1 neuspeh u izvršavanju.
- `wait()` koristimo da bi suspendovali dalje izvršavanje procesa dok se `detete` ne završi odnosno prekine `exit` pozivom, `wait` takodje vraća status završetka `deteta`

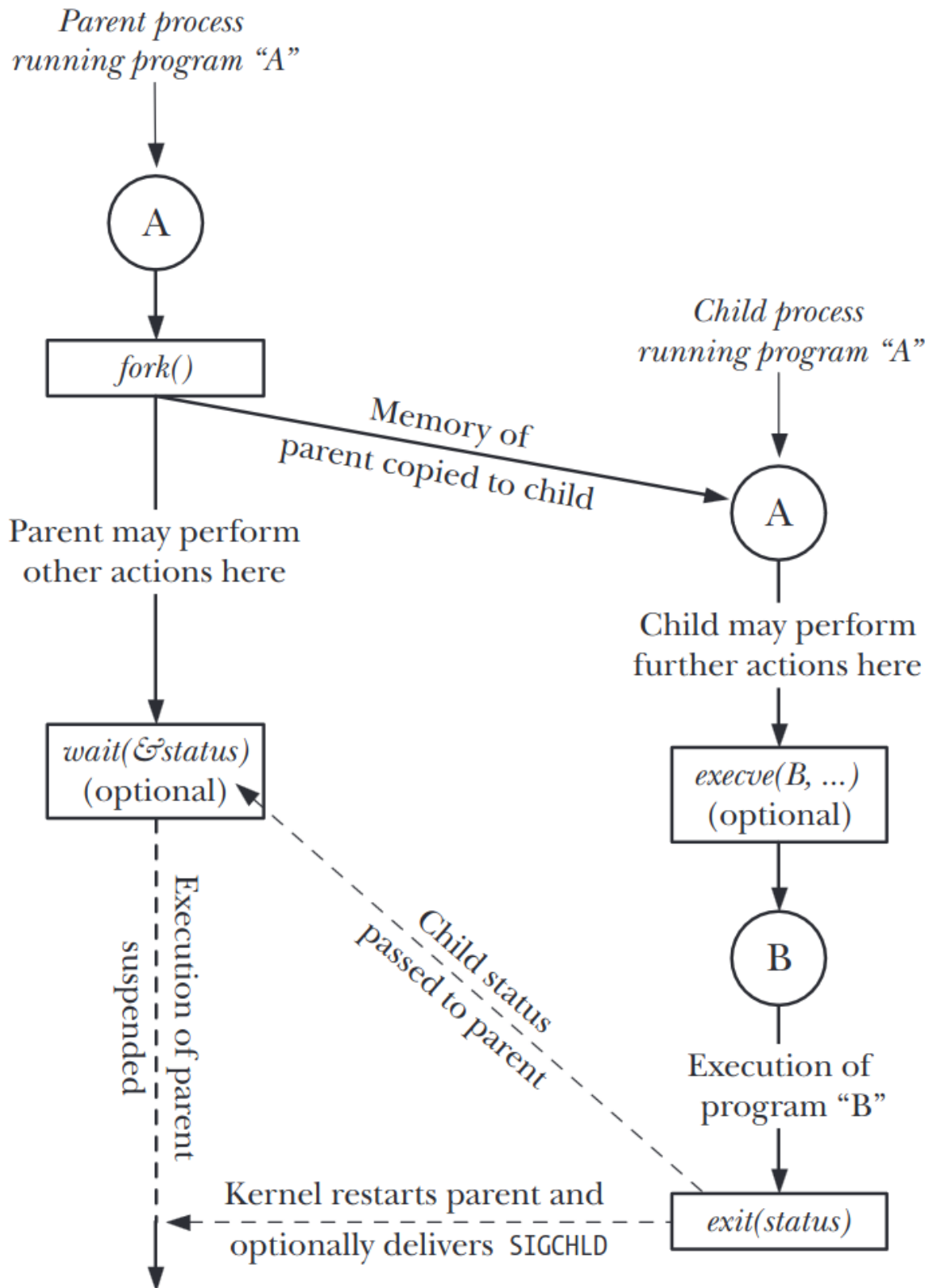


Figure 24-1: Overview of the use of *fork()*, *exit()*, *wait()*, and *execve()*

- Dijagram kreiranja `deteta`, njegovog zivota i unistenja.
- U programu mozemo da uz pomoc funkcije `fork` da kreiramo novi proces kome je nas vec pokrenut program `parent`. Funkcija `fork()` vraca `pid_t` tip podatka odnosno ceo broj koji oznacava ID naseg novog procesa.
- Proces `dete` ima ID o dok je PID `roditelja` uvek veci od nula i tako mozemo da ih i obradjujemo.

```

pid_t childPid = fork();
check_error(childPid != -1, "fork failed");

if (childPid > 0) { // parent branch
    printf("Parent. PPID: %jd, CPID: %jd\n", \
        (intmax_t)getpid(), (intmax_t)childPid);
} else { // child branch
    printf("Child. CPID: %jd, PPID: %jd\n", \
        (intmax_t)getpid(), (intmax_t)getppid());
}

printf("We both execute this\n")

```

- Ovaj kod kreira `dete` proces i iz njega ispisuje "Child". Nakon sto se oba procesa izvrse program ce se nastaviti sa izvorsavanjem `roditelj` i `dete` ce stici do `printf()` funkcije

```

pid_t childPid = fork();
check_error(childPid != -1, "fork failed");

if (childPid > 0) { // parent branch
    printf("Parent. PPID: %jd, CPID: %jd\n", \
        (intmax_t)getpid(), (intmax_t)childPid);
} else { // child branch
    printf("Child. CPID: %jd, PPID: %jd\n", \
        (intmax_t)getpid(), (intmax_t)getppid());
    exit(EXIT_SUCCESS)
}

printf("Only parrent\n")

```

- Kod je isit kao gore samo sto se `dete` ubija i ostaje samo `roditelj`. `roditelji` bez `dece` se zovu zombi procesi, tako da nakon svakog `fork` moramo da imamo i `wait` gde cemo obraditi zombi procese da ne bi zagusivali sistem.
- Isto tako moramo da vodimo i racuna o `sirocicima` ('orphans'). Siroce je `dete` kojem je roditelj završen pre njega.

✓ Obavezne stvari za cist i dobar kod

```

malloc i free
open i close
fork i wait

```

- S obzirom da `dete` dobija kopiju svih podataka koje je imao `roditelj` i da se oni izvorsavaju potpuno izolovano jedan od drugog, svaka promenljiva koju definisemo pre `fork` ce biti dodeljena `detetu` tj. kopije tih promenljivih

```

int var = 5;
pid_t childPid = fork();
check_error(childPid != -1, "fork failed");

if (childPid > 0) { // parent branch
    printf("Parent");

} else { // child branch
    printf("Child:\n");
    var *= 2;

    printf("Var in child: %d\n", var);
    exit(EXIT_SUCCESS);
}

int status;
check_error(wait(&status) != -1, "wait failed");

if (WIFEXITED(status))
    printf("Exit code: %d\n", WEXITSTATUS(status));
else
    printf("Process exited abnormally\n");

printf("Only parent executes this\n");
printf("Var in parent: %d\n", var);
exit(EXIT_SUCCESS);

```

- Ovaj kod ispisuje promenljivu var i proverava na kraju status kako se `dete` završilo.
- Izlaz ovog koda:

```

Hello from parent. My pid is 44274, child pid is 44275
Hello from child. My pid is 44275, parent pid is 44274
Var in child: 10
Process exited normally. Exit code: 0
Only parent executes this
Var in parent: 5

```

Redirekcija ulaza i izlaza procesa koriscenjem pipe-ova

- Pipe se na UNIX sistemima koristi tako da izlaz jednog procesa preusmerimo da bude ulaz drugog procesa.
- Pipe uvek mora da bude jednosmeran, odnosno samo da radi na relaciji izlaz -> ulaz za dva procesa.
- Ovo nam omogućava sistemski poziv `pipe` koji za dati niz celih brojeva od 2 elementa otvara 2 fajl deskriptora. Fajl deskriptor na poziciji nula je uvek za citanje, dok je na

poziciji i uvek za pisanje;

```
int p[2];
pipe(p);
// p[0] - read end - RD_E
// p[1] - write end - WR_E
```

- Implementacija pipe-a za dete i roditelja:
 1. Kreiramo dva niza celih brojeva sa po dva elementa
 2. sistemskim pozivom `pipe` kreiramo pipeove za relacije `deteRoditelj` i `roditeljDete`
 3. Kreiramo `dete` sa `fork`
 4. Zatvarmo nepotrebne fajl deskriptore da bi omogućili jednosmeran protok informacija, ovo zatvaranje fajlova radimo zato sto oba procesa dobijaju kopije podataka
 1. U `roditelju` zatvarmo `deteRoditelj[WR_E]` i `roditeljDete[RD_E]`
 2. U `detetu` zatvarom `deteRoditelj[RD_E]` i `roditeljDete[WR_E]`
 5. U parentu kreiramo poruku i pisemo u fajl deskriptor `roditeljDete[WR_E]`
 6. Zatim u `detetu` citamo iz `roditeljDete[RD_E]` i obradjujemo dalje poruku

```
int childToParent[2];
int parentToChild[2];

check_error(pipe(childToParent) != -1, "pipe");
check_error(pipe(parentToChild) != -1, "pipe");

pid_t childPid = fork();
check_error(childPid != -1, "fork");

if (childPid > 0) { // parent

    close(parentToChild[RD_END]);
    close(childToParent[WR_END]);

    char buffer[BUFF_SIZE];
    sprintf(buffer, "%s", "Hello child\n");

    check_error(write(parentToChild[WR_END], buffer, strlen(buffer))
!= -1, "write parent");

    // recive message from child
    char message[BUFF_SIZE];
    int readBytes = read(childToParent[RD_END], message, BUFF_SIZE);

    check_error(readBytes != -1, "read parent");
    message[readBytes] = 0;
```



```

    printf("Child message: %s\n", message);

    // close remaining fd's
    close(parentToChild[WR_END]);
    close(childToParent[RD_END]);
} else { // child

    close(childToParent[RD_END]);
    close(parentToChild[WR_END]);

    char message[BUFF_SIZE];
    int readBytes = read(parentToChild[RD_END], message, BUFF_SIZE);
    check_error(readBytes != -1, "read");
    message[readBytes] = 0;

    char buffer[BUFF_SIZE];
    sprintf(buffer, "Parent message: %s", message);

    printf("%s\n", buffer);

    // send message to parent
    sprintf(buffer, "%s", "Hello father\n");

    check_error(write(childToParent[WR_END], buffer, strlen(buffer))
!= -1, "write child");

    close(childToParent[WR_END]);
    close(parentToChild[RD_END]);

    exit(EXIT_SUCCESS);
}

check_error(wait(NULL) != -1, "wait");
exit(EXIT_SUCCESS);

```

- Ovaj kod predstavlja implementaciju gore navedenog procesa stvaranja pipe-a

Exec

- Varijacije funkcije `exec` nam omogućavaju da iz našeg programa pokrenemo druge programe, za razliku od `fork` koji vrši kopiranje adresnog prostora i dodeljuje ga `child` procesu, `exec` vrši zamenu adresnog prostora i naš program više ne postoji kao proces već je njegov prostor dodeljen novom procesu
- `exec` porodica funkcija se deli na:
 - I. `execl`:
 - I. `execvp`

- 2. `execle`
- 2. `execv`:
 - 1. `execvp`
 - 2. `execvpe`

`execle` koriste niz NULL-terminiranih nizova karaktera kao argumente

`execv` koriste `char **arguments` kao argumente funkcije koje pozivaju, prvi element ovog niza `arguments` se uvek koristi kao ime fajla kojeg pozivamo.

```
int main(int argc, char** argv) {  
  
    check_error(execle("ls", "ls", "-l", NULL) != -1, "exec failed");  
  
    printf("Ovaj deo koda se ne izvrsava\n");  
    exit(EXIT_SUCCESS);  
}
```

- Ovaj kod poziva funkciju `ls` sa argumentima `-l` koji vrši ispisivanje sadržaja trenutnog direktorijuma. Ovde za poziv `ls -l` koristimo funkciju `execle`

```
char** arguments = malloc(argc * sizeof(char*));  
check_error(arguments != NULL, "");  
  
for(int i = 1; i < argc; i++){  
    arguments[i-1] = malloc(strlen(argv[i]) + 1);  
    check_error(arguments[i-1] != NULL, "");  
    strcpy(arguments[i-1], argv[i]);  
}  
  
arguments[argc - 1] = NULL;  
pid_t childPid = fork();  
check_error(childPid != -1, "");  
  
if(childPid == 0){  
    check_error(execvp(arguments[0], arguments) != -1, "");  
    exit(EXIT_SUCCESS);  
}  
  
int status = 0;  
check_error(wait(&status) != -1, "");  
if(WIFEXITED(status) && (WEXITSTATUS(status) == EXIT_SUCCESS)){  
    printf("success\n");  
} else {  
    printf("fail\n");  
}  
  
exit(EXIT_SUCCESS);
```

- Ovaj kod implementira `execvp` i pokazuje način na koji se on implementira i pritom koristi `dete` proces da se adresni prostor koji koristi `roditelj` ne bi obrisao i što omogućava dalje izvršavanje programa nakon što `dete` pozove `exec` i završi se.
- Sa `wait` se brinemo o tome da se ne stvaraju zombiji, i tu završavamo i izvršavanje roditelja

🔗 TLPI poglavlja

- 24-24.2
 - 26.1-26.1.3, 26.2
 - 27-27.4
 - 44-44.4
-