МЕЂУПРОЦЕСНА КОМУНИКАЦИЈА (*Inter-process communication* – IPC)

Садржај

Међупроцесна комуникација

У оквиру материјала из претходног семестра, размотрен је системски позив ***wait***, помоћу кога један процес-родитељ може добити излазни (***exit***) статус од процеса детета. То је најједноставнија форма комуникације између два процеса, али није и најмоћнија. Механизми ***fork, exec, wait, exit***, не омогућавају ниједан вид комуникације родитеља и детета изузев преко аргумената командне линије и промењивих околине (енг. *environment variables*), а не омогућавају ни било који начин комуникације детета са родитељем изузев преко дететовог излазног статуса. Ниједан од ових механизама не пружа начин или средство за комуникацију са процесом дететом док је оно у стању извршавања, нити ови механизми дозвољавају комуникацију са процесима изван односа родитељ-дете.

У овој вежби су описана средства за међупроцесну комуникацију која заобилазе ова ограничења. Биће представљени различити начини комуникације између родитеља и детета, између „неповезаних" процеса, па чак и између процеса на различитим машинама.

Међурпроцесна комуникација (**IPC**) је **пренос података између процеса**.

На пример, Web претраживач може тражити Web страницу са сервера, који затим шаље HTML податке. Овај пренос података обично користи сокет механизам који је сличан телефонској конекцији. Други пример, жели се одштампати имена датотека у директоријуму користећи команду **ls | lpr**. *Shell* (командни интерпретер) ствара **ls** процес и посебан **lpr** процес, повезујући ова два процеса помоћу ***pipe*** (енг. цев, чита се „пајп") маханизма-датотеке, представљеног симболом "**|**".

Pipe механизам дозвољава једносмерну комуникацију између два сродна процеса. Процес **ls** уписује податке у *pipe*, а процес **lpr** их чита из *pipe*-а.

Радићемо пет типова интерпроцесне комуникације, а то су:

- **Дељива меморија** (*shared memory*) дозвољава процесима да комуницирају простим читањем и уписивањем у одређену меморијску локацију.
- **Мапирана меморија** је слична дељивој меморији, изузев што је повезана са датотеком у систему датотека.
- ***Pipe* датотеке** дозвољавају секвенцијалну комуникацију једног процеса са сродним процесом.
- **FIFO датотеке** су сличне као *pipe*, изузев што и неповезани процеси могу комуницирати јер FIFO датотека има додељено име у систему датотека.
- **Сокети**, они подржавају комуникацију између неповезаних процеса чак и на

различитим рачунарима.

Ови типови IPC разликују се по следећим критеријумима:

- По начину како ограничавају комуникацију:
  - на сродне процесе (процесе са заједничким претком),
  - на неповезане процесе који деле исти систем датотека,
  - на било који рачунар повезан у мрежу.
- По ограничењима комуникационог процеса:
  - да ли је дозвољено и уписивање података,
  - или је дозвољено само читање података.
- По броју процеса којима је дозвољено да комуницирају.
- По томе да ли су комуникациони процеси синхронизовани помоћу IPC – на пример процес читања је у стању чекања све док подаци не постану доступни за читање.

Занемарићемо дискусију о IPC дозволама за комуникацију код којих постоји ограничење колико се пута може комуницирати, нпр. као код комуникације преко излазног статуса детета.

IPC: дељива и мапирана меморија

Дељива меморија

Једна од најједноставнијих метода интерпроцесне комуникације јесте употреба дељиве меморије. Дељива меморија омогућава да два или више процеса приступе истој меморији ако позову ***malloc*** и ако сви враћени показивачи указују на исту меморију. Када један процес промени неку меморијску реч, сви остали процеси виде ту промену.

Брза локална комуникација

Дељива меморија је најбржа форма интерпроцесне комуникације зато што сви процеси деле исти део меморије. Приступ овој дељивој меморији је онолико брз колико је то приступ обичној (недељивој) меморији процеса, и ово не захтева системски позив или приступ кернелу. Такође – избегава се и непотребно копирање података.

С озбиром на то да кернел не синхронизује приступе дељивој меморији, корисник треба да обезбеди сопствену синхронизацију. Нпр. процес не би требало да може читати из меморије док се не заврши упис података у њу, нити би требало да два процеса могу уписивати у исту меморијску локацију истовремено.

Честа стратегија за избегавање ових стања трке јесте **употреба семафора**.

Напомена: У програмима који су дати у примерима, приказан је само случај када један процес приступа меморији. Ово је урађено како би фокус био на самом механизму дељиве меморије, али и да би се избегло компликовање кôда примера синхронизационом логиком.

Меморијски модел

Да би се користио дељиви меморијски сегмент, један од процеса мора алоцирати (***allocate***) сегмент. Затим сваки процес који жели да приступи сегменту мора тај сегмент и прикључити (***attach***). Након завршетка коришћења сегмента, сваки процес мора откачити сегмент (***detach***). На крају, један процес мора и ослободити (***deallocate***) сегмент – како не би тај део меморије остао непотребно заузет.

Разумевање Linux-овог меморијског модела помаже да се објасни алокација и прикључивање процеса. Код Linux-а, свака виртуелна меморија процеса подељена је на странице. Сваки процес мора водити рачуна о мапирању својих меморијских адреса у виртуелне меморијске странице које садрже актуелне податке. Иако сваки процес поседује сопствене адресе, вишеструко процесно мапирање може указивати на исту страницу, тиме дозвољавајући дељење меморије.

Алокација новог дељивог меморијског сегмента узрокује креирање виртуелних меморијских страница. С обзиром на то да сви процеси желе да приступе истом дељивом сегменту, требало би да само један процес алоцира нови дељиви сегмент. Алоцирањем постојећег сегмента се не креирају нове странице, већ се назад добија идентификатор већ постојећих страница.

Да би процес могао да користи дељиви меморијски сегмент, он га прикључује у свој адресни простор, чиме у своју виртуелну меморију додаје мапиране улазе који одговарају страницама дељеног сегмента. Када се заврши са употребом сегмента, ови мапирани улази се уклањају. Када више нема процеса који би користили дељиви меморијски сегмент, тачно један процес треба да делоцира виртуелне меморијске странице сегмента.

Сви дељиви меморијски сегменти алоцирани су као целобројни умношци величине системске странице, која је задата као број бајтова у страници меморије. На Linux системима, али и већини других система, величина странице износи 4KB, али корисник може одредити ову вредност позивајући функцију ***getpagesize***.

Алокација

Процес алоцира дељиви меморијски сегмент коришћењем **shmget** (*SHared Memory GET*) позива.

Параметри **shmget**:

1. Његов први параметар је целобројни кључ који одређује који сегмент се креира. Несродни процеси могу приступити истом дељивом сегменту уколико специфирају исте вредности кључа. На несрећу, други процеси могу такође одабрати исти фиксни кључ, а ово може водити до конфликта. Употребом специјалне константе **IPC_PRIVATE** као вредности кључа, гарантује да ће се креирати потпуно нови меморијски сегмент.
2. Његов други параметар дефинише број бајтова у сегменту. С обзиром на то да су сегменти алоцирани коришћењем страница, број стварно алоцираних бајтова је заокружен на цео број страница.
3. Трећи параметар је један бит или вредност заставице (флега) који одређује опције за *shmget*. Вредности флега укључују следеће:
    - **IPC_CREAT** — Овај флег указује на потребу за креирањем новог сегмента. Ово омогућава креирање новог сегмента на бази вредности кључа.
    - **IPC_EXCL** — Овај флег, који се увек користи са IPC_CREAT, узрокује неуспех у извршењу *shmget* у случају када се зада кључ сегмента који већ постоји. Због тога, он уређује да позвани процес увек добије посебан – *ексклузиван* – односно нов сегмент. У случају да овај флег није задат, а користи се кључ већ постојећег сегмента, *shmget* враћа постојећи сегмент уместо да креира нови.

- ***Mode flags*** — Ова вредност је сачињена од 9 битова који указују на права додељена власнику, групи и остатку света (*others*) за контролу приступа сегменту. Извршни битови се овде занемарују. Једноставан начин да се одреде права је коришћење константи дефинисаних у **<sys/stat.h>.** На пример, **S_IRUSR** и **S_IWUSR** одређују права читања и уписивања за власника дељивог меморијског сегмента, а **S_IROTH** и **S_IWOTH** одређују права читања и уписивања за остале (*others*).

Следи пример, овакво позивање *shmget*, креира нови дељиви меморијски сегмент (или приступ већ постојећем у случају да је *shm_key* већ употребљен) у који власник може уписивати и читати га, а који је недоступан за друге кориснике.

```
int segment_id = shmget (shm_key, getpagesize(),
                                          IPC_CREAT | S_IRUSR | S_IWUSR);
```

У случају да позив успе, *shmget* враћа идентификатор сегмента. Ако дељиви меморијски сегмент већ постоји, врши се верификација права приступа и проверава се да ли је сегмент означен за уништење.

Придруживање и избацивање

Да би дељиви меморијски сегмент учинио доступним, процес мора користити позив **shmat**, тј. *„Shared Memory ATtach”.*

Параметри *shmat*:

1. Овом позиву треба проследити идентификатор дељивог меморијског сегмента који се добија помоћу *shmget*.
2. Други аргумент је показивач, који одређује где у адресном простору процеса корисник жели да мапира дељиву меморију, а ако се проследи NULL, Linux ће сам одабрати расположиву адресу.
3. Трећи аргумент је флег, који може укључивати следеће:
   - **SHM_RND** указује да адреса која је одређена за други параметар треба да буде заокружена на умножак величине странице. Ако се не постави овај флег, потребно је лично подесити други аргумент за *shmat*.
   - **SHM_RDONLY** указује да ће сегмент бити могуће само читати, а не и уписивати.

Ако позив *shmat* успе, он враћа адресу придруженог дељивог сегмента. Деца процеси који се креирају *fork* позивом наслеђују придружене дељиве сегменте, али, ако то желе, она могу и избацити дељиве меморијске сегменте из свог простора.

Када се заврши са дељивим меморијским сегментом, сегмент је потребно избацити

користећи **shmdt** (*Shared Memory DeTach).* За *shmdt* треба користити адресу добијену помоћу *shmat.*
Ако се сегмент ослободи, и ако је то био последњи процес који га је користио, он се уклања. Системски позив *exit*, као и позивање било ког члана *exec* фамилије системских позива, аутоматски избацује сегмент.

Контрола и ослобађање дељиве меморије

Позив **shmctl** (*Shared Memory ConTroL* – контрола дељиве меморије) враћа информацију о дељивом меморијском сегменту и пружа могућност за његову измену.

Параметри *smctl:*

1. Први параметар је идентификатор дељивог меморијског сегмента.
2. Да би добили информацију о дељивом меморијском сегменту потребно је проследити: **IPC_STAT** као други аргумент и показивач на структуру *shmid_ds* као трећи.
3. За уклањање сегмента потребно је проследити: **IPC_RMID** као други аргумент и NULL као трећи аргумент. Сегмент се уклања када га и последњи процес који га је прикључио (*attach*) коначно избаци (*detach)* из свог адресног простора.

За више информација о *shmctl*-у можете погледати *man* страницу, ту је дат опис и осталих операција које се могу обавити над дељивим меморијским сегментима.

Пример употребе дељиве меморије (**shm.c**):

```c
#include <stdio.h>
#include <sys/shm.h>
#include <sys/stat.h>

int main() {
    int segment_id;
    char *shared_memory;
    struct shmid_ds shmbuffer;
    int segment_size;
    const int shared_segment_size = 0x6400;

    // Alociranje deljivog memorijskog segmenta - SHMGET
    segment_id = shmget(IPC_PRIVATE, shared_segment_size,
                            IPC_CREAT | IPC_EXCL | S_IRUSR | S_IWUSR);

    // Pridruzivanje deljivog memorijskog segmenta - SHMAT
    shared_memory = (char *) shmat(segment_id, 0, 0);
    printf("Deljiva memorija pridruzena na adresi %p\n", shared_memory);
```

```c
    // Odredjivanje velicine segmenta - SHMCTL
    shmctl(segment_id, IPC_STAT, &shmbuffer);
    segment_size = shmbuffer.shm_segsz;
    printf("Velicina segmenta: %d\n", segment_size);

    // Upisivanje stringa u deljivi memorijski segment
    sprintf(shared_memory, "Zdravo svete!");
    printf("Upisan string u deljivu memoriju!\n");

    // Odbacivanje deljivog memorijskog segmenta - SHMDT
    shmdt(shared_memory);

    // Ponovno pridruzivanje deljivog segmenta na drugoj adresi
    shared_memory = (char *) shmat(segment_id, (void *)0x5000000, 0);
    printf("Deljiva memorija ponovo pridruzena na adresi %p\n", shared_memory);

    // Ispisivanje stringa iz deljive memorije
    printf("Procitan string iz deljive memorije:\n\t %s \n", shared_memory);

    // Odbacivanje deljivog memorijskog segmenta - SHMDT
    shmdt(shared_memory);

    // Oslobadjanje (dealokacija) deljivog memorijskog segmenta - SHMCTL
    shmctl(segment_id, IPC_RMID, 0);
    return 0;
}
```

Преведите и покрените програм:

```
gcc -o shm shm.c
./shm
```

Потенцијални излаз програма је:

```
d@d:/mnt/c/Users/despot/Desktop/ipc programi$ ./shm
Deljiva memorija pridruzena na adresi 0x7f2feb35b000
Velicina segmenta: 25600
Upisan string u deljivu memoriju!
Deljiva memorija ponovo pridruzena na adresi 0x5000000
String iz deljive memorije:
        Zdravo svete!
```

Отклањање грешака

Команда **IPCS** пружа информације о особинама интерпроцесне комуникације, укључујући и дељиве сегменте. Коришћењем флега **-m** добијају се информације о дељивој меморији.

На пример, следећа слика илуструје да је у употреби један дељиви меморијски сегмент са бројем 229376. Овај излаз је добијен тиме што је у претходном програму наредба ослобађања меморије стављена под коментар.

```
d@d:/mnt/c/Users/despot/Desktop/ipc programi$ ipcs -m

------ Shared Memory Segments --------
key        shmid      owner      perms      bytes      nattch     status
0x00000000 229376     d          600        25600      0
```

Ако је овај меморијски сегмент грешком заостао иза програма, могуће га је уклонити командом **ipcrm**.

```
d@d:/mnt/c/Users/despot/Desktop/ipc programi$ ipcrm shm 229376
resource(s) deleted
d@d:/mnt/c/Users/despot/Desktop/ipc programi$ ipcs -m

------ Shared Memory Segments --------
key        shmid      owner      perms      bytes      nattch     status

d@d:/mnt/c/Users/despot/Desktop/ipc programi$
```

Разлози за и против дељиве меморије

Дељиви меморијски сегменти дозвољавају брзу двосмерну комуникацију између више процеса. Сваки корисник може и читати и уписивати податке, али програм мора успоставити и поштовати неке протоколе како би спречио трку за ресурсима и проблеме које она доноси – као што је то преписивање информације пре читања.

Нажалост, Linux не гарантује стриктно ексклузиван приступ, чак и када се креира нови дељиви сегмент користећи IPC_PRIVATE. Такође, да би више процеса користило исти дељиви сегмент, потребно је усагласити да сви користе исти кључ.

Мапирана меморија

Мапирана меморија дозвољава различитим процесима да комуницирају преко дељене датотеке. Мапирана меморију личи на употребу дељивог меморијског сегмента са именом, али ипак постоје неке техничке разлике. Мапирана меморија се може користити за интерпроцесну комуникацију, али и као једноставан начин за приступ садржају датотеке.

Мапирана меморија формира везу између датотеке и меморије процеса. Линукс дели датотеке на делове који су величине странице и онда их копира у странице виртуалне меморије како би их учинио доступним у адресном простору процеса. Зато, процеси могу читати садржај датотеке помоћу обичног меморијског приступа. Такође, процес може мењати садржај датотеке уписујући у меморију. Ово омогућава брз приступ датотекама.

На мапирану меморију се може гледати као на алокацију **бафера** који ће да чува читав садржај датотеке, а онда и као на учитавање датотеке у бафер и на крају уписивање бафера назад у датотеку, ако је дошло до неких промена. Linux рукује операцијама читања и уписивања у име корисника.

Мапирање обичне датотеке

Да би се мапирала обична датотека у меморију процеса, треба користити системски позив **mmap** ("Memory MAPed", изговара се "ем-мап").

Аргументи *mmap*:

1. Први аргумент је **адреса** на коју желите да Linux мапира датотеку у вашем процесном адресном простору. Вредност NULL дозвољава Линуксу да одабере неку погодну почетну адресу.
2. Други аргумент је **дужина мапе** у бајтовима.
3. Трећи аргумент одређује **заштиту** мапираног адресног опсега. Ова заштита је или **PROT_NONE** или се састоји од комбинације флагова **PROT_READ**, **PROT_WRITE** и **PR0T_EXEC** над којима се изврши операција логичког ИЛИ на нивоу бита, а које одговарају правима за читање, писање и извршавање респективно.
4. Четврти аргумент је вредност флега која одређује додатне опције.
5. Пети аргумент је **дескриптор датотеке** који описује датотеку која треба да се мапира.
6. Последњи аргумент је **померај** (*offset*) од почетка датотеке, од њега почиње мапирање. Одговарајућим избором почетног помераја и одговарајуће дужине, могуће је да у меморију мапирате целу датотеку или само део датотеке. Вредност флега је комбинација (добијена логичким ИЛИ) од ових константи:
    - **MAP_FIXED** — Ако поставимо овај флег, Linux користи затражену адресу да мапира датотеку уместо да је сам одређује. Ова адреса мора бити странично

подешена (*page-aligned*).

- **MAP_PRIVATE** — Уписи у меморијски опсег се не уписују у мапирану датотеку, него у приватну копију те датотеке. Ниједан други процесс не види ове уписе. Овај мод не би требало користити у комбинацији са MAP_SHARED флегом.
- **MAP_SHARED** — Уписивања се одмах прослеђују у мапирану датотеку, а не само у бафер. Користите овај мод кад користите мапирану меморију за IPC. Овај мод не би требало користити са флегом MAP_PRIVATE.

Ако позив успе, он враћа **показивач** на почетак меморије. При неуспеху, враћа флег **MAP_FAILED**. Када сте завршили са коришћењем мапиране меморије, ослободите је користећи ***munmap***. Покрените команду наводећи почетну адресу и дужину мапираног меморијског простора. Linux аутоматски демапира мапирани простор након завршетка процеса.

Упис у меморијски мапирану датотеку

Да бисмо илустровали коришћење мапираног меморијског простора осмотримо два програма за читање и упис у датотеку.

Први програм, **mmap-write.c**, генерише случајне бројеве и уписује их у меморијски-мапирану датотеку.

Други програм, **mmap-read.c**, чита бројеве, штампа их и замењује их у мапираном меморијски-мапираној датотеци, са њиховом двоструком вредношћу.

Оба програма користе аргументе командне линије као име датотеке која се мапира.

```c
#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <time.h>
#include <unistd.h>
#define FILE_LENGTH 0x100

// Vraca ujednacene slucajne brojeve u opsegu [low, high]
int random_range(unsigned const low, unsigned const high) {
    unsigned const range = high - low + 1;
    return low + (int)(((double)range) * rand() / (RAND_MAX + 1.0));
}

int main(int argc, char *const argv[]) {
    int fd;
    void *file_memory;
```

```
    // Generator slucajnih brojeva
    srand(time(NULL));

    // Priprema datoteku dovoljno veliku za prihvatanje neoznacenog celog broja
    // Ime datoteke se prosledjuje preko standardnog ulaza
    fd = open(argv[1], O_RDWR | O_CREAT, S_IRUSR | S_IWUSR);
    lseek(fd, FILE_LENGTH + 1, SEEK_SET);
    write(fd, "", 1);
    lseek(fd, 0, SEEK_SET);

    // Kreira mapiranu memoriju
    file_memory = mmap(0, FILE_LENGTH, PROT_WRITE, MAP_SHARED, fd, 0);
    close(fd);

    // Upisuje slucajan ceo broj u mapirani memorijski prostor
    sprintf((char *)file_memory, "%d\n", random_range(-100, 100));

    // Oslobadja memoriju (nepotrebno jer se program svakako nakon ovoga zavrsava)
    munmap(file_memory, FILE_LENGTH);
    return 0;
}
```

Преведите и покрените програм:

gcc -o mmap-write mmap-write.c

./mmap-write /tmp/integer-datoteka

Програм mmap-write отвара датотеку, креирајући је – уколико то претходно није урађено. Трећи аргумент за *open* одређује да је датотека отворена за читање и уписивање.

С обзиром на то да не знамо дужину датотеке, користимо *lseek* да бисмо се уверили да је датотека довољно велика да смести цео број и онда се се враћамо на почетну позицију у датотеци (*lseek*).

Програм мапира датотеку у меморију и затим затвара дескриптор датотеке зато што више није потребан. Програм затим уписује случајан цео број у мапирану меморију, а тиме и у датотеку, а затим демапира меморију. Позив *munmap* је непотребан зато што ће Linux аутоматски извршити *unmap* над датотеком, непосредно након завршетка програма.

Читање из меморијски мапиране датотеке

Програм mmap-read.c чита цео број из мапиране меморијске датотеке и удвостручује га.

```c
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <unistd.h>
#define FILE_LENGTH 0x100

int main(int argc, char* const argv[]) {
    int fd;
    void* file_memory;
    int integer;

    // Otvaranje datoteke
    fd =  open(argv[1], O_RDWR, S_IRUSR | S_IWUSR);

    // Stvaranje mapirane datoteke
    file_memory = mmap(0, FILE_LENGTH, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
    close(fd);

    // Citanje celobrojne vrednosti, stampanje i udvorostrucavanje
    sscanf(file_memory, "%d", &integer);
    printf("Vrednost: %d\n", integer);
    sprintf((char*) file_memory, "%d\n", 2*integer);

    // Oslobadjanje memorije (nepotrebno jer se program svakako nakon ovoga zavrsava)
    munmap(file_memory, FILE_LENGTH);

    return 0;
}
```

Преведите програм:

```
gcc -o mmap-read mmap-read.c
```

Програм mmap-read чита број из датотеке и затим уписује у датотеку удвостручену вредност тог броја. Прво, он отвара датотеку и мапира је у меморију за читање и уписивање. С обзиром на то да можемо претпоставити да је датотека довољно велика да прихвати неозначени цео број (интеџер), није потребно да користимо lseek, као што смо радили у претходном програму. Програм чита вредности из меморије тј. датотеке користећи sscanf наредбу, а затим их форматира и уписује двоструке вредности користећи sprintf.

Ово је пример извршења овог програма. Он мапира датотеку integer-datoteka.

```
./mmap-write integer-datoteka
cat integer-datoteka
  42
./mmap-read integer-datoteka
  Vrednost: 42
cat integer-datoteka
  84
```

Приметите да је текст 42 записан на диск без позивања системског позива *write*, и поново је прочитан без системског позива *read*. Напомена: овај програм-пример уписује и чита цео број као стринг (користећи *sprint* и *sscanf*) у демонстративну сврху - није неопходно да садржај меморијски мапиране датотеке буде текст. Нпр. можете сместити и поново узети произвољне бинарне цифре из меморијски мапиране датотеке.

Дељиви приступи датотекама

Различити процеси могу комуницирати, користећи мапиране меморијске области повезане истом датотеком. Поставите флег MAP_SHARED, тако да било који упис у ове меморијске регионе буде одмах пренет у мапирану датотеку и буде видљив другим процесима. Ако не поставите овај флег, Linux може кеширати (илити бафеоровати) уписе (*cache*), пре него што их пренесе у датотеку.

Алтернативно, можете принудити Linux да пребаци бафероване уписе у датотеку на диску позивајући ***msync***. Његова прва два параметра одређују мапирани меморијски простор, као и код **munmap**. Трећи параметар је флег са следећим вредностима:

- **MS_ASYNC** — Ажурирање је предвиђено, али не мора безусловно да се изврши пре краја позива.
- **MS_SYNC** — Ажурирање је тренутно; позив *msync* се блокира док се потпуно не изврши. Ово није могуће заједно користити MS_SYNC и MS_ASYNC.
- **MS_INVALIDATE** — Сва друга мапирања датотеке се поништавају, како би и та мапирања могла видети ажуриране вредности,

На пример: да бисмо ажурирали дељиву датотеку, мапирану на адреси mem_addr и дужине mem_length бајтова, позовимо следеће:

```
msync(mem_addr, mem_length, MS_SYNC | MS_INVALIDATE);
```

Као и код дељивих меморијских сегмената, корисници мапираних меморијских области морају успоставити и придржавати се протокола за избегавање стања трке. Нпр. семафор се може користити да спречи да у једном тренутку једној мапираној меморији приступа више од једног процеса. Алтернативно, можете користити *fcntl* да поставите

забрану читања или уписивања датотека.

Приватно мапирање

Постављањем **MAP_PRIVATE** код позива *mmap*, креира се област са особином "копирај при упису" (***copy-on-write***). Било који упис у ову област се одражава само на ову меморију процеса, а други процеси који мапирају исту датотеку, неће видети промене.

Уместо да уписује директно на страницу која је дељена са свим процесима, процес уписује у приватну копију своје странице. Свако наредно читање и писање овог процеса, користи ову копирану страницу.

Друге употребе mmap

Поред интерпроцесне комуникације, позив ***mmap*** се може користити и у друге сврхе.

Једна уобичајена употреба је замена за системске позиве *read* и *write*. На пример, уместо да експлицитно прочита садржај датотеке у меморији, програм ће радије да мапира датотеку и скенира је користећи меморијско читање. За неке програме ово је погодније и такође се може извршити брже него експлицитне У/И операције датотека.

Једна напредна и моћна техника коју користе неки програми јесте креирање структуре података (нпр. обичне структуре) у меморијски мапираној датотеци. При каснијем позиву, програм поново мапира ту датотеку у меморију, а структуре података се враћају у своје претходно стање. Приметите то да ће показивачи на ове структуре података бити погрешни уколико сви не показују у оквиру граница истог меморијски мапираног простора и уколико се програм не побрине да се датотека поново мапира у исти адресни простор који је раније заузимала.

Друга погодна техника је мапирање специјалне **/dev/zero** датотеке у меморију. Ова датотека се понаша као бесконачно дуга датотека попуњена нулама. Програм коме је потребан извор нула може користити mmap на датотеку /dev/zero. Уписивања у /dev/zero се одбацују, тако да се мапирана меморија може користити у било коју сврху. Уобичајени меморијски алокатори често мапирају /dev/zero да би добили делове преиницијализоване меморије попуњене нулама.

IPC : pipe и FIFO

Pipe је комуникациони уређај који дозвољава једносмерну комуникацију. Податак који је уписан на уписни крај *pipe* датотеке *("**write end**")* се чита са краја за читање pipe датотеке *("**read end**")*.

Pipe датотеке су серијски уређаји, подаци се увек читају из pipe датотеке у истом редоследу како су уписани. Обично се pipe користи за комуникацију између две нити у једном процесу или између процеса родитеља и процеса детета.

Неименовани пипе

У **shell** програму, симбол „|“ креира пипе. На пример, ова *shell* команда узрокује да *shell* створи два процеса детета, једна за ls и један за less:

ls | less

Shell такође ствара pipe датотеку, спајајући стандардни излаз *ls* процеса са стандардним улазом *less* процеса. Имена датотека излистана помоћу *ls* се шаљу ка *less* у истом оном поретку у којем би били послати директно ка терминалу.

Капацитет података pipe датотеке је **ограничен**. Ако процес који уписује пише брже него што процес који чита може да прочита податке, и ако pipe не може да сачува вишак података, процес писац се блокира док већи капацитет не постане доступан. Ако процес читалац покуша да прочита, а не постоје расположиви подаци, он се блокира све док подаци не постану доступни. Дакле, pipe **аутоматски синхронизује два процеса**.

Креирање pipe датотека

Да би се креирао *pipe*, треба користити системски позив *pipe*. Системски позив pipe захтева на улазу целобројни низ величине **2**. Позив pipe смешта дескриптор читања датотеке на позицију **0** у низу, а дескриптор уписа у датотеку на позицију **1**.

На пример, осмотримо следећи код:

```
int pipe_fd[2];
int read_fd;
int write_fd;
pipe(pipe_fd);
read_fd = pipe_fds[0];
write_fd = pipe_fds[1];
```

Подаци уписани преко *write* дескриптора write_fd се могу прочитати помоћу read

дескриптора read_fd.

Комуникација између процеса родитеља и процеса детета

Позив pipe креира **два дескриптора** датотеке, а који су валидни само за тај процес и за његову децу. Дескриптори датотека процеса не могу се делити између два неповезана процеса. Када процес позове *fork*, дескриптори датотека се копирају и у нови процес дете. Дакле, pipe може спајати једино повезане процесе.

У програму **pipe.c** *fork* ствара процес дете. Дете наслеђује дескрипторе *pipe* датотеке. Родитељ уписује стринг у *pipe*, а дете процес га чита. Једноставан програм конвертује ове дескрипторе датотека у **FILE* stream** структуре, користећи *fdopen*. С обзиром на то да радије користимо *stream* структуре него дескрипторе датотека, можемо користити улазно/излазне функције вишег реда из стандардне Ц библиотеке као што су *fprintf* и *fgetc*.

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>

/*  Upisuje "count" broj puta poruku u tok "stream",
    izmedju upisivanja pravi pauzu jednu sekundu. */
void writer(const char *message, int count, FILE *stream) {
    for ( ; count > 0; --count) {
        // Upisuje poruku u niz, i odmah ih salje dalje...
        fprintf(stream, "%s\n", message);
        fflush(stream);
        sleep(1); // Ceka jednu sekundu
    }
}

// Cita proizvoljne stringove iz niza sto je duze moguce
void reader(FILE *stream) {
    char buffer[1024];
    /* Cita sve dok ne stigne do kraja niza. fgets cita do nove linije ili do kraja. */
    while (!feof(stream) && !fer-
ror(stream) && fgets(buffer, sizeof(buffer), stream) != NULL)
        fputs(buffer, stdout);
}

int main() {
    int fds[2];
    pid_t pid;
    // Kreira pipe. Fajl deskriptori oba kraja se smestaju u fds
```

```
    pipe(fds);

    // Stvara se proces dete
    pid = fork();
    if (pid == (pid_t)0) {
        FILE *stream;
        /* Ovo je proces dete. Zatvara kopiju kraja za upis fajl deskriptora - nepo-
trebna je. */
        close(fds[1]);
        /* Konvertuje file-deskriptor za citanje u ob-
jekat tipa FILE i cita iz njega. */
        stream = fdopen(fds[0], "r");
        reader(stream);
        close(fds[0]);
    } else {
        // Ovo je proces roditelj
        FILE *stream;
        // Zatvara nasu kopiju kraja za citanje datoteka
        close(fds[0]);
        /* Konvertuje file-descriptor upisa u datoteka object i upisuje u njega */
        stream = fdopen(fds[1], "w");
        writer("Zdravo, svete!", 5, stream);
        close(fds[1]);
    }
    return 0;
}
```

Преведите и извршите програм pipe.c и објасните излазне резултате.

gcc -o pipe pipe.c

./pipe

```
d@d:/mnt/c/Users/despot/Desktop/ipc programi$ gcc pipe.c -o pipe
d@d:/mnt/c/Users/despot/Desktop/ipc programi$ ./pipe
Zdravo, svete!
Zdravo, svete!
Zdravo, svete!
^C
```

На почетку *main* функције, fds се декларише као целобројни низ величине **2**. *Pipe* позив креира pipe датотеку и смешта дескрипторе за упис и читање у тај низ. Програм тада преко позива *fork* креира процес дете.

Након затварања краја за читање pipe датотеке, процес родитељ почиње да уписује податке у pipe датотеку. Након затварања краја за упис pipe, дете чита податке из pipe датотеке.

Запазите да након уписивања у writer-функцији, родитељ обавља **flush** функцију како би сравнио стање из кеш меморије у pipe датотеку. У супротном, може се десити да стринг не буде послат одмах кроз *pipe*.

Када позивате команду ls | less, дешавају се два позива fork: један за ls дете процес и један less дете процес. Оба ова процеса наслеђују pipe *file*-дескрипторе како би могли комуницирати користећи pipe датотеку.

Уколико се жели остваривање комуникације између неповезаних процеса, треба користити FIFO, као што је описано у наредном одељку – „FIFO".

Преусмеравање стандардног улаза, излаза и излаза за грешке

Често ћете желети да креирате процес дете и да поставите један крај pipe као његов стандардни улаз или стандардни излаз. Коришћењем позива ***dup2***, можете изједначити један фајл-дескриптор са другим. На пример, да би се преусмерио стандардни улаз процеса на фајл-дескриптор fd, треба користити следећу линију:

```
dup2(fd, STDIN_FILENO);
```

Симболичка константа **STDIN_FILENO** представља фајл-дескриптор за стандардни улаз, који има вредност 0 (нула). Позив затвара стандардни улаз и поново га отвара као дупликат fd-а како би се могао користити за размењивање. Изједначени фајл-дескриптори деле исту позицију датотека и исте фајл-статус флегове. Зато се карактери прочитани помоћу fd не читају поново (*reread*) кроз стандардни улаз.

Пример преусмеравања излаза

Програм **dup2.c** користи системски позив ***dup2***, да пошаље садржај из pipe датотеке ка команди за сортирање. Након креирања *pipe*, програм обавља *fork*. Процес родитељ исписује податке кроз *pipe* датотеку. Процес дете придружује фајл-дескриптор за читање *pipe* на свој стандардни улаз користећи *dup2*; тада извршава програм за сортирање.

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

int main() {
    int fds[2];
    pid_t pid;
```

```
    pipe(fds);    // Kreira se pipe
    pid = fork(); // Stvara se proces dete.

    if (pid == (pid_t) 0) {
        // Ovo je proces dete.
        // Zatvara se nasa kopija kraja za upis file-deskriptora
        close(fds[1]);
        // Povezuje se kraj za citanje pipe na standardni ulaz
        dup2(fds[0], STDIN_FILENO);
        // Zamenjuje se proces dete programom za sortiranje
        execlp("sort", "sort", (char *) 0);
    } else {
        // Ovo je proces roditelj.
        FILE *stream;
        // Zatvara se nasa kopija kraja za citanje file-deskriptora
        close(fds[0]);

        /* Kovertuje se file deskriptor za upis i vrsi se upisivanje u njega. */
        stream = fdopen(fds[1], "w");
        fprintf(stream, "Ovo je test.\n");
        fprintf(stream, "Zdravo, svete.\n");
        fprintf(stream, "Ovaj program nije dosadan!\n");
        fprintf(stream, "Necemo pauzu.\n");
        fprintf(stream, "Hocemo domaci!\n");

        fflush(stream);
        close(fds[1]);

        // Ceka se da proces dete zavrsi
        waitpid(pid, NULL, 0);
    }
    return 0;
}
```

Преведите и извршите програм пипе.ц и објасните излазне резултате.

```
gcc -o dup2 dup2.c
./dup2
```

Процес дете сортира садржај ріре датотеке коју пуни процес родитељ стринговима:

```
d@d:/mnt/c/Users/despot/Desktop/ipc programi$ gcc dup2.c -o dup2
d@d:/mnt/c/Users/despot/Desktop/ipc programi$ ./dup2
Hocemo domaci!
Necemo pauzu.
Ovaj program nije dosadan!
Ovo je test.
Zdravo, svete.
```

popen and pclose

Уобичајена употреба pipe датотеке је слање или примање података од програма који се извршава у другом процесу. Функције ***popen*** и ***pclose*** олакшавају ову употребу (парадигму) елиминишући потребу за употребом pipe, fork, dup2, exec и fdopen. Упоредите програм **popen.c** који користи *popen* и *pclose* са претходним примером dup2.c.

```c
#include <stdio.h>
#include <unistd.h>

int main () {
    FILE* stream = popen("sort", "w");
    fprintf (stream, "Ovo je test.\n");
    fprintf (stream, "Zdravo, svete.\n");
    fprintf (stream, "Ovaj program nije dosadan!\n");
    fprintf (stream, "Necemo pauzu.\n");
    fprintf (stream, "Hocemo domaci!\n");

    return pclose(stream);
}
```

Позив ***popen*** креира процес дете које извршава команду за сортирање замењујући позиве pipe, fork, dup2, и execlp. Други аргумент "**w**", указује да овај процес хоће да уписује у процес дете.

Повратна вредност позива *popen* је један крај pipe датотеке, а други крај је повезан на стандардни улаз процеса детета. Након завршетка уписивања, *pclose* затвара *stream* структуру процеса детета, чека да се процес заврши и враћа његову статусну вредност.

Први аргумент позива popen се извршава као shell команда у процесу кога покреће /bin/sh. Shell користи **PATH** варијаблу на уобичајени начин како би пронашао програме да би их извршио.

Ако је други аргумент "**r**", функција враћа процесу детету стандардни *stream* излаз, како би родитељ могао прочитати тај излаз. Ако је други аргумент "**w**", функција враћа процесу стандардни *stream* улаз како би родитељ могао да пошаље податке. Ако се догоди грешка, *popen* враћа **null** (нулти) показивач.

Позовите *pclose* да би се затворила stream структура која је враћена помоћу *popen*. Након

затварања одређене стреам структуре, *pclose* чека на завршетак процеса детета.

FIFO

FIFO датотека је pipe који **има име** у систему датотека. Било који процес може отворити или затворити FIFO, процеси не морају бити сродни један са другим. FIFO се другачије зове и „**именовани пипе**“.

Можете креирати FIFO користећи ***mkfifo*** команду. Одредите путању до FIFO датотеке на командној линији. На пример, креирајте FIFO у /tmp/fifo користећи следеће:

```
mkfifo /tmp/fifo
ls -l /tmp/fifo
  prw-rw -rw - 1 d users 0 Dec 16 14:04 /tmp/fifo
```

Први карактер излаза из ls је **p**, који указује да је датотека заправо FIFO (именовани pipe).

Отворите два терминала. У једном прозору читајте из FIFO користећи:

```
cat < /tmp/fifo
```

У другом прозору, уписујте у FIFO користећи:

```
cat > /tmp/fifo
```

Затим укуцајте неколико линија текста. Сваки пут када притиснете Ентер, линија текста се шаље кроз FIFO и појављује се у првом прозору. Затворите FIFO притискањем Ctrl + D у другом прозору. Уклоните FIFO командом:

```
rm /tmp/fifo
```

Креирање FIFO

Креирајте FIFO програмски користећи ***mkfifo*** функцију.

Први аргумент је путања до новокреиране FIFO датотеке. Други параметар одређује права приступа за pipe датотеку (за власника, групу, остале). С обзиром на то да pipe мора поседовати и читача (*reader*) и уписивача (*writer*), дозволе морају укључивати и право читања и право уписивања.

Ако pipe не може бити креиран (на пример, ако датотека са тим именом већ постоји), *mkfifo* враћа **-1**.

Укључите **<sys/types.h>** и **<sys/stat.h>** ако користите mkfifo.

Приступ FIFO датотеци

Приступ FIFO датотеци је исти као приступ обичној датотеци. За комуникацију кроз FIFO, потребно је да га један програм отвори за упис, а други да га отвори за читање.
Користе се У/И функције нижег реда (*open, write, read, close*) или У/И функције из Ц библиотеке (*fopen, fprintf, fscanf, fclose,* итд.).

На пример, за упис бафера података у FIFO датотеку користећи У/И рутине нижег реда, можете користити следећи код:

```
int fd = open(fifo_path, O_WRONLY);
write(fd, data, data_length);
close(fd);
```

На пример, за читање података из FIFO датотеке, користећи У/И функције Ц библиотеке, можете користити овај код:

```
FILE* fifo = fopen(fifo_path, "r");
fscanf(fifo, "%s", buffer);
fclose(fifo);
```

FIFO може имати више читача (*readers*) или уписивача (*writers*). Бајтови са сваког уписивача се атомично (недељиво) уписују до максималне величине **PIPE_BUF** (4KB на Linux-у). Делови са симултаних уписивача се могу преплитати. Слична правила се могу применити и на симултане читаче.

Разлике у односу на Windows именоване pipe датотеке

Pipe датотеке у Win32 оперативним системима су веома сличне Linux pipe датотекама (погледати Win32 документацију за техничке детаље). Главна разлика код именованих pipe датотека је у томе што код Win32 система они више функционишу као **сокети**. Win32 именовани pipe могу повезати процесе на различитим рачунарима повезаним преко мреже. На Linux-у се у ову сврху користе сокети. Такође, Win32 дозвољава више конекција типа читач-уписивач на именованој pipe датотеци без преплитања, а *pipe* датотеке се могу користити за двосмерну комуникацију.

IPC : сокети

Сокет (прикључница) је **бидирекциони уређај** који се може користити за комуникацију са другим процесом на истој машини или са процесом на другим машинама. Сокети су једина интерпроцесна комуникација коју ћемо посматрати, а која дозвољава комуникацију између процеса на различитим рачунарима. Интернет програми као што су Telnet, rlogin, FTP, talk, и WWW користе сокете.

На пример, можете добити WWW страницу са Web сервера, користећи Telnet програм, зато што оба процеса користе сокете за мрежну комуникацију. Да бисте отворили конекцију ка WWW серверу на www.mumbojumbo.com, користите наредбу `Telnet www.mumbojumbo.com 80`. Број порта 80 одређује конекцију ка Web серверу и програмско покретање www.mumbojumbo.com уместо неког другог процеса. Покушајте да укуцате GET / након што је конекција успостављена. Ова команда шаље поруку кроз сокет ка Web серверу, који одговара шаљући HTML код за *home page* (почетне стране) и затим затвара конекцију, на пример:

```
d@d:~$ telnet www.google.com 80
Trying 216.58.212.36...
Connected to www.google.com.
Escape character is '^]'.
GET /index.html
HTTP/1.0 200 OK
Date: Wed, 23 Dec 2020 09:33:45 GMT
Expires: -1
Cache-Control: private, max-age=0
Content-Type: text/html; charset=ISO-8859-1
P3P: CP="This is not a P3P policy! See g.co/p3phelp for more info."
Server: gws
X-XSS-Protection: 0
X-Frame-Options: SAMEORIGIN
Set-Cookie: 1P_JAR=2020-12-23-09; expires=Fri, 22-Jan-2021 09:33:45 GMT; path=/; domain=.google.com; Secure
```

Обично ћете користити Telnet да повежете Telnet сервер са удаљеним приступима (***logins***). Али Telnet можете користити такође за конекцију са сервером друге врсте и онда укуцавати коментаре директно на њега.

Концепт сокета

Када креирате сокет, морате одредити три параметра:

- **стил комуникације**
- ***namespace*** (простор имена)
- **протокол**.

Тип или стил комуникације контролише како сокети третирају послате податке и

одређује број партнера за комуникацију. Када је податак послат кроз сокет, он се групише у делове који се називају **пакети**. Стил комуникације одређује како се поступа са овим пакетима и како се они адресирају од пошиљаоца до примаоца. Постоје два стила комуникације:

- **Конекцијски стил** гарантује испоруку свих пакета у поретку како су и послани. Ако су пакети изгубљени или је промењен њихов редослед услед проблема у мрежи, прималац аутоматски захтева од пошиљаоца њихово поновно слање. Код конекцијског стила, сокет је као телефонски позив: адресе пошиљаоца и примаоца су фиксне на почетку комуникације и током конекције.

- **Датаграм стил** не гарантује поредак пакета за испоруку или пријем. Пакети могу бити изгубљени или преорганизовани током транспорта кроз мрежу у зависности од мрежних грешака или других услова. Сваки пакет мора бити обележен заједно са својом дестинацијом и није гарантовано да ће бити испоручен. Систем гарантује само "најбољи труд" (*best-effort*), тако да пакети могу нестати или бити преорганизовани током транспорта. Сокети са датаграм стилом се понашају више као поштанска писма. Пошиљалац специфира адресу примаоца за сваку поруку појединачно.

Namespace сокета одређује како се пише адреса сокета. Адреса сокета идентификује један крај сокет конекције. На пример, сокет адресе у "локалном простору имена" су обична имена датотека. У "Интернет простору имена" сокет адресе се састављају од Интернет адресе (такође познате као Интернет протокол адресе - *Internet Protocol address* или ИП адресе - *IP Address*) хоста прикаченог на мрежу и броја порта. Порт бројеви су различити за сваки сокет на истом хосту.

Протокол одређује како се преносе подаци. Неки протоколи су:

- TCP/IP, примарни протокол који се користи на Интернету
- AppleTalk мрежни протоцол.
- UNIX локални комуникациони протокол

Нису подржане све комбинације стилова, имена простора и протокола.

Системски позиви

Сокети су флексибилнији од претходно осмотрених комуникационих техника. Ово су системски позиви који су везани за сокете:

- ***socket*** - креира сокет
- ***close*** - уништава сокет
- ***connect*** - креира конекцију између два сокета
- ***bind*** - обележава серверски сокет адресом

- ***listen*** - конфигурише сокет да прихвати услове
- ***accept*** - прихвата конекцију и креира нови сокет за конекцију

Сокети се представљају помоћу *file*-дескриптора.

Креирање и уништавање сокета

Функције ***socket*** и ***close*** креирају и уништавају сокете, респективно. Када креирате сокете специфирајте три сокет параметра: *namespace*, комуникациони стил и протокол.

За *namespace* параметар користите константе које почињу са **PF_** (скраћеница *protocol family,* тј. "фамилија протокола"). На пример,

- PF_LOCAL или PF_UNIX одређује локални *namespace*
- PF_INET одређује Интернет *namespace*.

За комуникациони стил, користите константе које почињу са **SOCK_**. Користите:

- SOCK_STREAM - сокет са конекцијским стилом
- SOCK_DGRAM - сокет са датаграм стилом

Трећи параметар, **протокол**, одређује механизам нижег реда за слање и примање података. Сваки протокол је одговарајући за одређену комбинацију *namespace* - стил. С обзиром на то да обично постоји један најбољи протокол за сваки такав пар, постављањем **0** обично се добија добар протокол.

Ако је сокет позив био успешан, он враћа фајл-дескриптор за сокет. Можете читати, уписивати у сокет исто као и код других фајл-дескриптора. Када сте завшрили са сокетом, покрените **close** како бисте га уклонили.

Системски позив **connect**

Да би се креирала конекција између два сокета, клијент сокет позива **connect**, притом назначавајући адресу серверског сокета на који се повезује. Клијент је процес који покреће конекцију, а сервер је процес који чека да прихвати конекције.

Клијентски процес позива *connect* како би отпочео конекцију од локалног сокета до сокета сервера, назначеног другим аргументом. Трећи аргумент је дужина у бајтовима адресне структуре на коју се указује другим аргументом. Формати адреса сокета се разликују у зависности од *namespace* сокета.

Слање информација

Било која техника за упис у дескриптор датотеке може се користити да би се уписивало у сокет. Погледајте дискусију о Linux-овим У/И функцијама нижег реда и о неким последицама њихове употребе.

Функција **send**, која је посебна за сокет дескрипторе датотека, омогућава алтернативан упис са неколико додатних могућности, а за информације о команди *send*, погледајте *man* страну.

Сервери

Животни циклус сервера састоји се од креирања сокета конекцијског стила, додељивања адресе сокету, постављање позива за ослушкивање (***listen***), који омогућава конекцију са сокетима, постављање позива за прихватање долазне конекције; и на крају затварање сокета.

Подаци се не читају или уписују директно кроз сервер сокет, већ сваки пут када програм прихвати нову конекцију, Linux креира посебан сокет који се користи за пренос података преко те конекције. У овом одељку, упознаћемо се са **bind**, **listen** и **accept** позивима.

Системски позив **bind**

Адреса мора бити додељена серверском сокету, коришћењем **bind**. Први аргумент је сокет *file*-дескриптор. Други аргумент је показивач на сокет адресну структуру, а формат зависи од фамилије сокет адресе. Трећи аргумент је дужина адресне структуре у бајтима.

Системски позив **listen**

Када је адреса додељена сокету са конекцијским стилом, он мора покренути ослушкивање како би указивао да је сервер. Први аргумент је сокет *file*-дескриптор. Други аргумент одређује колико нерешених конекција стоји у **реду за чекање**. Ако је ред за чекање пун, додатне конекције ће бити одбијене. Ово не ограничава укупан број конекција које сервер може да опслужи; ово ограничава само број конекција које покушавају да се конектују, а које још нису прихваћене.

Системски позив **accept**

Сервер прихвата конекцију од клијента покретањем позива **accept**. Први аргумент је сокет фајл-дескриптор. Други аргумент указује на сокет адресну структуру која се попуњава сокет адресом клијента. Трећи аргумент је дужина у бајтима сокет адресне структуре. Сервер може користити адресу клијента да би одлучио да ли стварно жели да комуницира са клијентом. Позив *accept*, креира нови сокет за комуникацију са клијентом и враћа се нови, одговарајући file-дескриптор. Оригинални сервер сокет наставља да прихвата нове клијентске конекције.

Да бисте прочитали податак са сокета, али без уклањања података из реда за чекање, користите позив **recv**. Он користи исте аргументе као read, уз додатни FLAG аргумент. Флег MSG_PEEK проузрокује да податак буде прочитан, али не и уклоњен са улазног реда

за чекање (*queue*).

Локални сокети

Сокети који спајају процесе на истом рачунару могу користити локални *namespace* представљен синонимима **PF__LOCAL** и **PF_UNIX**. Они се зову локални сокети или **UNIX-domain** сокети. Њихове сокет адресе, одређене именом датотека, се користе само када се креирају конекције. Сокет име је одређено у структури **sockaddr_un**. Потребно је поставити **sun_family** поље на **AF_LOCAL**, указујући тиме да је ово локални *namespace*. Поље **sun_path** одређује име датотеке која се користи, а може бити дуго највише 108 бајтова. Стварна дужина структуре sockaddr_un требало би да се обрачуна коришћењем макроа **SUN_LEN**.

Може се користити било које име датотека, али процес мора имати права уписа над директоријумом, јер то му дозвољава додавање датотека у директоријум. Да би се конектовао на сокет, процес мора имати право читања сокет датотеке. И поред тога што различити рачунари могу делити исти систем датотека, само процеси који се извршавају на истом рачунару могу комуницирати преко локалних *namespace* сокета.

Једини протокол који је допуштен за локални namespace је **0**. Пошто се налази у систему датотека, локални сокет је приказан као датотека. На пример, запазите иницијал **s**:

```
ls -l /tmp/socket
```

```
srwxrwx--x  1  user  group 0  Dec 18 19:18  /tmp/socket
```

Позовите **unlink** за уклањање локалног сокета када сте завршили са њим.

Пример коришћења локалног *namespace* сокета

Илустроваћемо сокете са два програма: **socket-server.c** и **socket-client.c**. Сервер програм у **socket-server.c** креира локални *namespace* сокет и чека на конекцију преко њега. Када прими конекцију, он чита текстуалне поруке са конекције и исписује их све до затварања конекције. Ако је једна од ових порука "*quit*" серверски програм уклања сокет и завршава се. Серверски сокет програм узима путању до сокета као аргумент са командне линије.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <unistd.h>

/* Cita poruke sa soketa i prikazuje ih. To radi sve do zatvaranja soketa.
   Vraca nenultu vrednost ako klijent posalje "quit", a nulu u drugom slucaju. */
```

```c
int server(int client_socket) {
    while(1) {
        int length;
        char *text;

        /* Prvo cita duzinu tekstualne poruke sa soketa.
           Ako citanje vrati nulu, klijent zatvara konekciju. */
        if (read(client_socket, &length, sizeof(length)) == 0)
            return 0;

        // Alocira se bafer da bi prihvatio poruku.
        text = (char *) malloc(length);

        // Cita sam tekst i prikazuje ga
        read(client_socket, text, length);
        printf("%s\n", text);

        // Oslobadja bafer
        free(text);

        // Ako klijent posalje poruku "quit," sve je zavrseno
        if (!strcmp(text, "quit"))
            return 1;
    }
}

int main(int argc, char *const argv[]) {
    const char *const socket_name = argv[1];
    int socket_fd;
    struct sockaddr_un name;
    int client_sent_quit_message;

    // Kreira soket
    socket_fd = socket(PF_LOCAL, SOCK_STREAM, 0);

    // Ukazuje na to da je ovo server
    name.sun_family = AF_LOCAL;

    strcpy(name.sun_path, socket_name);
    bind(socket_fd, (struct sockaddr *)&name, SUN_LEN(&name));

    // Ceka - osluskuje konekcije
    listen(socket_fd, 5);
```

```
    /* Ponavlja prihvatanje konekcija, cineci da jedan server radi sa svakim
       klijentom. Nastavlja dok klijent ne posalje "quit" poruku. */

    do {
        printf("Ulazi u while\n");
        struct sockaddr_un client_name;
        socklen_t client_name_len;
        int client_socket_fd;

        // Prihvata konekciju
        client_socket_fd = accept(socket_fd, (struct sockaddr *)&client_name, &cli-
ent_name_len);

        // Rukuje konekcijom
        client_sent_quit_message = server(client_socket_fd);

        // Zatvara nas kraj konekcije
        close(client_socket_fd);
    } while (!client_sent_quit_message);

    // Uklanja soket datoteka
    close(socket_fd);
    unlink(socket_name);

    return 0;
}
```

Клијентски програм socket-client.c, конектује се на *local-namespace* сокет и шаље поруку. Путања до сокета и порука су одређени на командној линији.

```
#include <stdio.h>
#include <string.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <unistd.h>

// Upisuje u soket tekst dat putem deskriptora SOCKET_FD
void write_text(int socket_fd, const char *text) {
    // Upisuje broj bajtova u string, ukljucujuci NULL zavrsetak
    int length = strlen(text) + 1;
    write(socket_fd, &length, sizeof(length));
```

```
    // Upisuje string
    write(socket_fd, text, length);
}

int main(int argc, char *const argv[]) {
    const char *const socket_name = argv[1];
    const char *const message = argv[2];
    int socket_fd;
    struct sockaddr_un name;

    // Kreira soket
    socket_fd = socket(PF_LOCAL, SOCK_STREAM, 0);

    // Cuva ime servera u adresi soketa
    name.sun_family = AF_LOCAL;
    strcpy(name.sun_path, socket_name);

    // Konektuje soket
    connect(socket_fd, (struct sockaddr *) &name, SUN_LEN (&name));

    // Upisuje tekst sa komandne linije u soket
    write_text(socket_fd, message);

    close(socket_fd);
    return 0;
}
```

Пре него што клијент пошаље текст поруке, он шаље дужину текста у бајтовима (варијабла **length**). Такође, сервер прво чита дужину текста поруке из сокета. Ово омогућава серверу да алоцира одговарајућу величину бафера за прихватање текста поруке, а пре читања са сокета.

Преведите и извршите програме и објасните излазне резултате. Да бисте пробали овај пример, покрените сервер програм у посебном прозору. Одредите путању до сокета - на пример, socket.

```
./socket-server socket
```

У другом прозору, покрените клијента неколико пута, одређујући исту сокет путању, и поруке да их пошаљете клијенту:

```
./socket-client socket "Hej, hej, hej."
```

```
./socket-client socket "Ovo je test."
```

Програм сервер прихвата и исписује ове поруке. Да бисте затворили сервер, пошаљите поруку "quit" са клијента:

```
./socket-client socket "quit"
```

Програм сервер се затвара.

Интернет-domain сокети

UNIX-domain сокети могу се користити само за комуникацију између два процеса на истом рачунару. За разлику од претходно наведених, **Интернет-домен** сокети се могу користити за повезивање процеса на различитим машинама, повезаних мрежом.

Сокети који повезују процесе преко интернета, користе Интернет *namespace*, који се представља помоћу **PF_INET**. Најчешћи протоколи за њега су TCP/IP. Интернет протокол (**ИП**), протокол нижег реда, преноси пакете кроз Интернет, делећи их и поново их спајајући ако је то потребно. Он гарантује само "**best-effort**" (најбољи труд) испоруку, тако да пакети могу нестати или бити реорганизовани током транспорта. Сваки рачунар учесник је одређен коришћењем ИП броја. Transmission Control Protocol (**ТЦП**-протокол контроле преноса), ослањајући се на ИП, пружа поуздан и уредан транспорт. Он дозвољава успоставу телефонске конекције између два рачунара и осигурава да ће пренос података бити сигуран и уредан.

ДНС имена

С обзиром не то да је лакше запамтити имена него бројеве, Domain Name Service (**ДНС**) повезује имена као што је www.mumbojumbo.com са јединственом ИП адресом. ДНС је имплементиран помоћу светске хијерархије имена сервера, с тим што ви не морате да разумете ДНС протоцол да бисте користили хост имена у својим програмима.

Интернет сокет адресе садрже два дела**: број машине-ИП** и **број порта**. Ове информације се чувају у структури **sockaddr_in** променљиве. Поставите **sin_family** поље на **AF_INET** да бисте указали да је ово Интернет *namespace* адреса. Поље **sin_addr**, чува Интернет адресу жељене машине у облику ИП броја (у битима). Порт помаже да се разликују различити сокети који су додељени рачунару. С обзиром на то да различите машине Чувају вишебајтне вредности у различитом бајт-поретку, користите **htons** да би конвертовали порт број у мрежни бајт поредак ***(network byte order)***. Погледајте *ip man* страну за више информација.

Да бисте конвертовали људима читљива хост имена, или бројеве у стандардном дот-означавању (као што је 10.0.0.1), или ДНС имена (као што је www.mumbojunibo.com) у 32-битне IP бројеве, можете користити функцију **gethostbyname**, која враћа показивач на структуру **hostent**; поље **h_addr** садржи host IP број.

Пример употребе Интернет-домен сокета

Програм **socket-inet.c** илуструје употребу Интернет-домаин сокета. Програм добија *home page* (почетну страну) са Web сервера чије је хост име одређено на командној линији.

```c
#include <stdlib.h>
#include <stdio.h>
#include <netinet/in.h>
#include <netdb.h>
#include <sys/socket.h>
#include <unistd.h>
#include <string.h>

/* Prikazuje sadrzaj home page-a sa server soketa. Vraca indikator uspeha. */
void get_home_page(int socket_fd) {
    char buffer[10000];
    ssize_t number_characters_read;

    // Salje HTTP GET komandu za home page
    sprintf(buffer, "GET /\n");
    write(socket_fd, buffer, strlen(buffer));

    /* Cita sa soketa. Poziv za citanje mozda ne moze vratiti sve podatke
        u jednom trenutku, tako da nastavlja da pokusava dok ga ne prekinemo. */
    while (1) {
        number_characters_read = read(socket_fd, buffer, 10000);
        if (number_characters_read == 0)
            return;
        // Ispisuje podatke na standardni izlaz
        fwrite(buffer, sizeof(char), number_characters_read, stdout);
    }
}

int main(int argc, char *const argv[]) {
    int socket_fd;
    struct sockaddr_in name;
    struct hostent *hostinfo;

    // Kreira soket.
    socket_fd = socket(PF_INET, SOCK_STREAM, 0);

    // Cuva ime servera u adresu soketa.
    name.sin_family = AF_INET;
```

```
    // Konvertuje iz stringa u brojeve.
    hostinfo = gethostbyname(argv[1]);
    if (hostinfo == NULL)
        return 1;
    else
        name.sin_addr = *((struct in_addr *)hostinfo -> h_addr);

    // Web serveri koriste port 80.
    name.sin_port = htons(80);

    // Konektuje se na Web server
    if (connect(socket_fd, (struct sockaddr *)&name, sizeof(struct sockaddr_in)) == -
1) {
        perror("connect");
        return 1;
    }

    // Trazi server home page (pocetnu stranu)
    get_home_page(socket_fd);

    return 0;
}
```

Овај програм узима *hostname* Web сервера са командне линије (не URL, тј. без "http://"). Он позива **gethostbyname** да би превео име хоста у нумеричку ИП адресу и да би затим конектовао *stream* (ТЦП) сокет на порт број 80 за тај рачунар. Web сервери користе Hypertext Transport Protocol (HTTP), тако да програм издаје HTTP GET команду и сервер одговара шаљући текст своје *home* странице.

Преведите и извршите програм соцкет-инет.ц и објасните излазне резултате.

```
d@d:/mnt/c/Users/despot/Desktop/ipc programi$ ./socket-inet www.ftn.pr.ac.rs
<html>
        <head>
                <title>Nikola Tesla</title>
        </head>
        <body>
                <h1>Nikola Tesla</h1>
                <p>This is the default web page for this server.</p>
                <p>Server name: Nikola Tesla.</p>
        </body>
</html>
```

Стандардни порт бројеви

По договору, Web сервери чекају на конекције на порту 80. Већина Интернет мрежних сервиса је повезана помоћу стандардних порт бројева. На пример, сигурни Web сервери

који користе SSL чекају на конекцију на порту 443, а e-mail сервери (који користе SMTP) користе порт 25.

На GNU/Linux системима, веза између имена протокола/сервиса и стандардних порт бројева је наведена у датотеци **/etc/services**. Прва колона је име протокола или име сервиса. Друга колона наводи број порта и врсту конекције: *tcp* за конекцијски орјентисану, или *udp* за датаграм. Ако имплементирате уобичајене мрежне сервисе користећи Интернет-домен сокете, користите портове веће од 1024.

Сокет парови

Као што смо видели раније, *pipe* функција креира два дескриптора датотека за почетак и крај пајпа. Они су ограничени зато што дескрипторе датотека морају користити само сродни процеси и зато што је комуникација унидирекциона (једносмерна). Функција **socketpair**, креира два дескриптора датотеке за два повезана сокета на истом рачунару. Ови филе-дескриптори дозвољавају двосмерну комуникацију између сродних процеса. Његова прва три параметра су иста као и она код сокет позива: они одређују домен (*domain),* стил конекције, и протокол. Последњи параметар је низ са два целобројна члана који је попуњен дескрипторима датотеке за два сокета, слично као код пипе. Када позивате *socketpair*, морате специфирати **PF_LOCAL** као домен.