# IPC

- Zdieľaná pamäť
- Semafóry
- Mapovaná pamäť

# Zdieľaná pamäť

- Zdieľaná pamäť (ZP) je najrýchlejší spôsob komunikácie medzi procesmi
- Nevyžaduje si systémové volania jadra
- Je treba zabezpečiť synchronizáciu procesov pri prístupe k ZP

# Koncepcia ZP

- Každý proces má vlastnú tabuľku stránok
- Tabuľka stránok zobrazuje virtuálnu pamäť (VP) do fyzickej pamäti ( FP)
- Keď sa proces začne vykonávať má alokovaný pamäťový segment, ktorý obsahuje

  - stack

  - segment kódu programu

  - segment dát
- Každý segment je zložený zo  stránok
- Vytvorenie ZP je vytvorenie a identifikácia zdieľaných stránok pamäte pre daný proces

# Zdieľaná pamäť

- Procesy musia vykonať nasledovné kroky pre získanie ZP

  Niektorý proces musí alokovať pamäťový segment

  Procesy, ktoré chcú mať k alokovanému segmentu prístup si ho musia pripojiť tzn. pridať zdieľané stránky do tabuľky stránok

  Po skončení musia všetky procesy odpojiť segment

  Proces, ktorý segment vytvoril ho musí dealokovať.

# Zdieľaná pamäť

- Alokácia pamäťového segmentu znamená buď jeho vytvorenie alebo ak už existuje, proces získa jeho identifikátor
- Pripojenie procesu k pamäťovému segmentu znamená pridanie položky, ktorá určí zobrazenie virtuálnej pamäti procesu do zdieľaného segmentu
- Odpojenie a dealokácia pamäťového segmentu sú inverzné operácie

# Alokácia pamäte

- Na alokáciu zdieľanej pamäte slúži funkcia:

  int shmget(key_t key, size_t size, int shmflg); (*Shared memory get*)

  Prvý argument je kľúč segmentu (celočíselná hodnota), ktorý sa má vytvoriť. Hodnota kľúča IPC_PRIVATE, zabezpečí tvorbu nového segmentu, ktorý bude určite iný ako s existujúcimi kľúčmi.

  Druhý argument je veľkosť v bytoch zaokrúhlená na násobok veľkosti stránky, ktorá je v Linuxe 4 KB.

  Tretí argument je flag, ktorého hodnoty sú:

  IPC_CREAT- vytvorí sa nový segment alebo sa získa prístup k už existujúcemu segmentu

  IPC_EXCL (používa sa s IPC_CREAT) - vytvorí sa osobitný segment, ak segment s daným kľúčom existuje nastane chyba

# Alokácia pamäte

S_IRUSR, S_IWUSR – R/W prístup vlastníka
S_IROTH, S_IWOTH – R/W prístup pre ostatných

- Príklad

  int segment_id = shmget (shm_key, getpagesize(),

  IPC_CREAT | S_IRUSR | S_IWUSER);

  Vytvorí  nový segment (alebo získa prístup k už existujúcemu s daným kľúčom), ktorý je s prístupom čítania aj zapisovania (R/W) pre vlastníka.

  Návratová hodnota je ID segmentu.

# Pripojenie a odpojenie

- Na pripojenie zdieľaného segmentu slúži funkcia:

  void *shmat(int shmid, const void *shmaddr, int shmflg); kde

  shmid je identifikátor segmentu, ktorý vracia shmget()

  druhý argument je adresa, na ktorú sa mapuje ZP, môže byť aj NULL (adresu vyberie Linux)

  tretí argument môže mať hodnoty:

  SHM_RND – určuje, že adresa v druhom argumente je zaokrúhlená (round down) na násobok veľkosti stránky.

  SHM_RDONLY – segment je len na čítanie.

  0 – segment je pre čítanie aj zápis

  Po úspešnom pripojení funkcia vracia adresu segmentu.

- Na odpojenie slúži funkcia:
  int shmdt(const void *shmaddr)
  kde argumentom je adresa segmentu
- Po volaní execve a exit sú všetky pripojené segmenty odpojené
- Ak je segment odpojený posledným procesom, ktorý ho používa potom je zrušený

# Riadenie zdieľaného segmentu

- Funkcia int shmctl(int shmid, int cmd, struct shmid_ds *buf);
  sa používa na zmenu prístupu a iných charakteristík k ZP pritom musí byť proces vlastníkom segmentu s daným ID.
  cmd argument môže mať hodnoty:
  SHM_LOCK – zabránenie swapovaniu zdieľaného segmentu
  SHM_UNLOCK – umožnenie swapovania zdieľaného segmentu
  IPC_STAT – skopírovanie informácie z jadra do shmid_ds *buf
  IPC_SET – zapísanie niektorých položiek štruktúry shmid_ds danej v buf do dátovej štruktúry jadra vytvorenej k segmentu
  IPC_RMID – označenie zrušenia segmentu ak sa posledný proces odpojí (shm_nattch v štruktúre shmid_ds je 0)
  SHM_INFO – vracia štruktúru shm_info, ktorej položky obsahujú info o systémových zdrojoch používaných zdieľanou pamäťou

- Štruktúry shm_info aj shmid_ds sú definované v <sys/shm.h>
- Štruktúra shmid_ds obsahuje informácie:

  struct ipc_perm shm_perm - vlastníctvo a prístupové práva

  shm_segsz - veľkosť segmentu

  shm_atime – čas posledného pripojenia

  shm_dtime - čas posledného odpojenia

  shm_ctime – čas poslednej zmeny

  shm_cpid – PID tvorcu

  shm_lpid – PID procesu, ktorý sa posledný pri/odpojil

  shm_nattch – počet pripojení

- Štruktúra ipc_perm obsahuje
  key – kľúč segmentu
  uid – UID vlastníka
  gid – GID skupiny vlastníka
  cuid – UID tvorcu
  cgid – GID skupiny tvorcu
  mode – práva
  seq – poradové číslo

# Príklad 1

- Nasledovný program ukazuje použitie funkcií na
- vytvorenie zdieľaného segmentu
- pripojenie  zdieľaného segmentu (adresu určí linux) a zápis do segmentu
- odpojenie zdieľaného segmentu
- znovu pripojenie zdieľaného segmentu (mapovanie na inej adrese) a výpis zo segmentu
- zrušenie zdieľaného segmentu

**Exercise Shared Memory**

```
#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; /*0x as a prefix
    means hexadecimal in C*/
  /* Allocate a shared memory segment. */
  segment_id = shmget (IPC_PRIVATE, shared_segment_size,
  IPC_CREAT | IPC_EXCL | S_IRUSR | S_IWUSR);
```

```
/* Attach the shared memory segment. */
shared_memory = (char*) shmat (segment_id, 0, 0);
printf (“shared memory attached at address %p\n”,
    shared_memory);
/* Determine the segment’s size. */
shmctl (segment_id, IPC_STAT, &shmbuffer);
segment_size = shmbuffer.shm_segsz;
printf (“segment size: %d\n”, segment_size);
/* Write a string to the shared memory segment. */
sprintf (shared_memory, “Hello, world.”);
/* Detach the shared memory segment. */
shmdt (shared_memory);
```

```
/* Reattach the shared memory segment, at a
   different address. */
shared_memory = (char*) shmat (segment_id,
   (void*) 0x5000000, 0);
printf ("shared memory reattached at address
   %p\n", shared_memory);
/* Print out the string from shared memory. */
printf ("%s\n", shared_memory);
/* Detach the shared memory segment. */
shmdt (shared_memory);
/* Deallocate the shared memory segment. */
shmctl (segment_id, IPC_RMID, 0);
return 0;
}
```

# Synchronizácia procesov pri prístupe k ZP

- ZP je kritická oblasť
- Pre zabezpečenie konzistencie údajov môže byť v KO vždy len jeden proces, ktorý do oblasti zapisuje a žiaden iný
- Úloha riadeného prístupu sa nazýva vzájomné vylúčenie procesov (mutual exclusion)
- Na riadenie prístupu do KO sa zriaďujú systémové prostriedky: semafóry

# Semafóry procesov

- Na synchronizáciu procesov používa Linux osobitnú implementáciu semafórov nazývanú aj semafóry procesov alebo Systém V semafóry.
- Podobne ako ZP je potrebné semafór najprv alokovať a inicializovať.
- Tvorca semafóra môže meniť vlastníka a prístupové práva
- Na semafóre sa vykonávajú operácie ako wait a post
- Nakoniec je treba semafór dealokovať alebo zrušiť

# Alokácia

- Na vytvorenie množiny semafórov sa používa funkcia:

  int semget(key_t key, int nsems, int semflg); kde

  prvý argument je identifikátor množiny semafórov (celočíselná hodnota key),

  druhý argument je počet semafórov,

  tretí argument je fleg, ktorý špecifikuje počiatočné prístupové práva semflg: IPC_CREATE | mode.

  Funkcia po úspešnom volaní vracia ID množiny semafórov

# Štruktúry vytvorené po alokácii

```
struct semid_ds  {
        struct ipc_perm  * sem_perm;   // pristupove prava
        time_t   sem_otime;    // cas poslednej operacie “semop”
        time_t   sem_ctime;    // cas poslednej operacie “semctl”
        ushort  sem_nsems;         // pocet semaforov v sete
};
```

```
struct  ipc_perm  {
        key_t key;          // kľúč ako v semget
        uid_t   uid;        // UID vlastnika
        gid_t   gid;        // GID skupiny vlastníka
        uid_t   cuid;       // UID tvorcu
        gid_t  cgid;        // GID skupiny tvorcu
        mode_t  mode;       // prístupové práva
        ushort seq;         // poradové číslo
};
```

mode:

- 0400 - read user
- 0200 - write user
- 0040 - read group

- Každý semafór z množiny semafórov má pridružené nasledovné hodnoty

```
struct sem {
        ushort  semval;         // hodnota na semafore
        pid_t   sempid          // pid procesu, ktory posledny
                                // operoval so semaforom
        ushort  semncnt;    // počet procesov čakajúcich
//pokiaľ hodnota na semafore nebude väčšia ako je semval
        ushort  semzcnt;   // pocet procesov čakajucich na
                           // nulovu hodnotu  semafora
};
```

# Zmena charakteristík semafóra

- Nasledovná funkcia umožňuje inicializovať, meniť prístupové práva a iné charakteristiky a dealokovať množinu semafórov:

  int semctl(int semid, int semnum, int cmd, union semun arg); kde

  Prvý argument je platné ID možiny semafórov, druhý argument je index poľa semafórov,

  tretí argument cmd je riadiaci flag, ktorý môže nadobúdať niekoľko hodnôt,

  štvrtý argument je voliteľný ak sa vyžaduje je treba union semun explicitne deklarovať.

ak si cmd vyžaduje argument potom ho treba definovať cez union

```
union  semun {
        int val;        // hodnota pre SETVAL
        struct semid_ds  * buf;     // bufer pre
                                    IPC_STAT, IPC_SET
        ushort  *array;      // pole pre GETALL,SETALL
        struct seminfo *__buf; // bufer pre IPC_INFO
};
```

# Argument cmd

- Argument cmd môže mať nasledovné hodnoty, od niektorých závisí návratová hodnota funkcie semctl:

  SETALL – nastaví pole semafórov na danú hodnotu

  IPC_RMID – zruší pole semafórov

  IPC_INFO – vráti štruktúru seminfo
  GETVAL – vráti hodnotu semafóra semval
  SETVAL – nastaví hodnotu semafóra na semval

# Argument cmd

GETPID – vráti PID procesu, ktorý vykonal poslednú operáciu na množ. semafórov

GETNCNT – vráti počet procesov, ktoré čakajú na zvýšenie hodnoty semafóra

GETZCNT – vráti počet procesov čakajúcich na 0 hodnotu semafóra

GETALL – vráti hodnoty všetkých semafórov do arg.array

IPC_STAT – vráti stavovú informáciu semafórov do smerníka na bufer typu semid_ds

IPC_SET – nastaví užívateľa a skupinu a ich prístupové práva

# Inicializácia semafóra

- Na inicializáciu sa používa funkcia semctl s nasledovnými argumentami:

  semctl (semid, 0, SETALL, arg)

  SETALL nastaví hodnoty všetkých semafórov z množiny danej smerníkom arg.array
- Nasledovný kód programu ukazuje použitie inicializácie ako aj deklaráciu unionu semun

# Inicializácia – príklad

**Initializing a Binary Semaphore**

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
union semun {
  int val;
  struct semid_ds *buf;
  unsigned short int *array;
  struct seminfo *__buf;
};
```

# Inicializácia – príklad

```
/* Initialize a binary semaphore with a value of 1. */
int binary_semaphore_initialize (int semid)
{
  union semun argument;
  unsigned short values[1];
  values[0] = 1;
  argument.array = values;
  return semctl (semid, 0, SETALL, argument);
}
```

# Dealokácia

- Na dealokáciu sa používa funkcia semctl s nasledovnými argumentami:

  semctl (semid, nsems, IPC_RMID, arg)

  IPC_RMID – zruší množinu semafórov

  argument arg je ignorovaný
- Nasledovný kód programu ukazuje alokáciu a dealokáciu binárneho semafóra.

# Príklad

**Allocating and Deallocating a Binary Semaphore**

```
#include <sys/ipc.h>
#include <sys/sem.h>
#include <sys/types.h>
/* We must define union semun ourselves. */
union semun {
  int val;
  struct semid_ds *buf;
  unsigned short int *array;
  struct seminfo *__buf;
};
```

# Príklad

```
/* Obtain a binary semaphore's ID, allocating if necessary.*/
int binary_semaphore_allocation (key_t key, int sem_flags)
{
  return semget (key, 1, sem_flags);
}
  /* Deallocate a binary semaphore. All users must have
     finished their use. Returns -1 on failure. */
int binary_semaphore_deallocate (int semid)
{
  union semun ignored_argument;
  return semctl (semid, 1, IPC_RMID, ignored_argument);
}
```

# Operácie na semafóre

- Každý semafór  má nezáporné hodnoty a podporuje **wait** a **post** operácie.
- Funkcia semop implementuje obe tieto operácie na množine semafórov. Jej prototyp je:

  int semop(int semid, struct sembuf *sops, size_t nsops);
  kde

  semid je semafór ID získané volaním semget(), sops je smerník na pole štruktúr obsahujúcich informácie:

```
struct sembuf {
  ushort_t sem_num;   /* semaphore number */
  short sem_op;       /* semaphore operation */
  short sem_flg;      /* operation flags */
};
```

  Posledný argument nsops špecifikuje počet operácií na semafóre

# Operácie na semafóre

- Operácie v štruktúre sembuf sú určené nasledovne, ak je sem_op:

  **celé kladné číslo** – znamená inkrementovanie semafóra o danú hodnotu

  **záporné celé číslo** – znamená dekrementovanie hodnoty semafóra. Ak by mala byť hodnota semafóra < 0 spôsobí to chybu (nevykonanie operácie) alebo blokovanie (pokiaľ hodnota semafóra nebude ≥ ako absolútna hodnota sem_op a pritom sa dekrementuje semncnt) v závislosti či je sem_flg nastavený na IPC_NOWAIT alebo nie.

  **nula** – ak IPC_NOWAIT je nie nastavené znamená blokovanie operácie pokiaľ bude hodnota semafóra 0 a súčasne sa dekrementuje semzcnt v opačnom prípade sa operácia na semafóre sa nevykoná.

# Operácie na semafóre

- sem_flg v štruktúre sembuf môže mať hodnoty:

  IPC_NOWAIT – zabráni blokovaniu operácie. V prípade ak by mala byť  hodnota semafóra záporná alebo v prípade ak sa testuje či je nenulová hodnota semafóra rovná nule volanie semop spôsobí chybu a operácie sa nevykonajú.

  SEM_UNDO – spôsobí nevykonanie operácií pri exite procesu
- Proces bude blokovaný (flag nie je nastavený na IPC_NOWAIT) pokiaľ:

  sú nie všetky operácie úspešne ukončené

  proces dostane signál

  množina semafórov je zrušená

# Operácie na semafóre

- Súčasne môže na semafóre vykonávať operácie len jeden proces.
- Súčasné požiadavky viacerých procesov o vykonanie operácie budú vykonané v ľubovolnom poradí.
- Ak je proces s výlučným prístupom k semafóru ukončený abnormálne operácia sa nevykoná a semafór sa neuvolní a zostáva v pamäti uzamknutý v takom stave ako ho proces zanechal.
- Aby sa tomu zabránilo slúži flag SEM_UNDO, ktorý vráti semafór do jeho predchádzajúceho stavu – množina semafórov nezostane v nekonzistentnom stave.

# Príklad

**Wait and Post Operations for a Binary Semaphore**

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
/* Wait on a binary semaphore. Block until the semaphore value is
    positive, then decrement it by 1. */
int binary_semaphore_wait (int semid)
  {
  struct sembuf operations[1];
  /* Use the first (and only) semaphore. */
  operations[0].sem_num = 0;
  /* Decrement by 1. */
  operations[0].sem_op = -1;
  /* Permit undo'ing. */
  operations[0].sem_flg = SEM_UNDO;
  return semop (semid, operations, 1);
  }
```

```
  /* Post to a binary semaphore: increment its value by 1. This returns
     immediately. */
int binary_semaphore_post (int semid)
{
  struct sembuf operations[1];
  /* Use the first (and only) semaphore. */
  operations[0].sem_num = 0;
  /* Increment by 1. */
  operations[0].sem_op = 1;
  /* Permit undo'ing. */
  operations[0].sem_flg = SEM_UNDO;
  return semop (semid, operations, 1);
}
```

# Príklad 2 – komunikácia dvoch procesov cez ZP

- Nasledovný program demonštruje komunikáciu procesov klient/server cez zdieľanu pamäť
- Proces server do pamäti zapisuje
- Proces klient z pamäti číta
- Pri prístupe k ZP sa procesy synchronizujú semafórom
- Na zrušení pamäti a semafóra sa procesy synchronizujú zápisom dohodnutého charakteru klientom na prvú pozíciu v ZP (že skončil čítanie) a server môže pamäť a semafór zrušiť
- Program sa spušťa v dvoch oknách v jednom server v druhom klient v uvedenom poradí

**shm_server.c**

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/sem.h>
#include <stdio.h>
#define SHMSZ 27

union semun {
    int val;
    struct semid_ds *buf;
    unsigned short int *array;
} ;
int alloc_semaphore();
int init_semaphore();
int lock_semaphore();
int unlock_semaphore();
void free_semaphore();
```

```
main()
{
  char c;
  int shmid, sem;
  key_t shm_key, SEM_KEY;
  char *shm, *s;
  SEM_KEY = 4545;
  /* We'll name our shared memory segment "5678".*/
  shm_key = 5678;
  /* Create the segment.*/
  if ((shmid = shmget(shm_key, SHMSZ, IPC_CREAT | 0666)) < 0) {
  perror("shmget");
  exit(1);
  }
/* Now we attach the segment to our data space.*/
  if ((shm = shmat(shmid, NULL, 0)) == (char *) -1) {
  perror("shmat");
  exit(1);
  }
```

```
  alloc_semaphore();
  init_semaphore();
  /* Now put some things into the memory for the other process to
     read.*/
  s = shm;
  lock_semaphore();
  for (c = 'a'; c <= 'z'; c++){
  *s++ = c;
  *s = NULL;
   sleep(1);
  }
  unlock_semaphore();
/*Finally, we wait until the other process changes the first character of
     our memory to '*', indicating that it has read what we put there.*/
  while (*shm != '*');
  shmdt(shm);
  shmctl(shmid, IPC_RMID, 0);
  free_semaphore();
  exit(0);
}
```

```
int alloc_semaphore()
{                                                          //alokovanie semafora
        sem = semget(SEM_KEY, 1, 0664 | IPC_CREAT);
        if (sem == -1) {
                perror("semget() failed");
                exit(1);
        }
        return 0;
}

int init_semaphore()
{                                                          //inicializacia semafora
   semun semunion;
   ushort values[1];
   values[0] = 1;
   semunion.array = values;
   if (semctl(sem,0,SETALL, semunion) == -1){
        perror("semctl() failed");
        exit(1);
   }
   return 0;
}
```

```
int lock_semaphore()
{                                                    //uzamknutie semafora
        struct sembuf semaphore[1];
        semaphore[0].sem_num = 0;
        semaphore[0].sem_op = -1;
        semaphore[0].sem_flg = SEM_UNDO;
        if (semop(sem, semaphore, 1) == -1) {
                perror("can't lock semaphore");
                exit(1);
        }
        return 0;
}
```

```
int unlock_semaphore()
{                                                    //odomknutie semafora
        struct sembuf semaphore[1];
        semaphore[0].sem_num = 0;
        semaphore[0].sem_op = +1;
        semaphore[0].sem_flg = SEM_UNDO;
        if (semop(sem, semaphore, 1) == -1) {
                perror("can't unlock semaphore");
                exit(1);
        }
        return 0;
}
```

```
void free_semaphore()
{                                                    //uvolnime semafor
        if (semctl(sem, 1, IPC_RMID, 0) == -1) {
                perror("can't remove semaphore");
        }
}
```

**shm_client.c**

```
/* shm-client - client program to demonstrate shared memory.*/
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/sem.h>
#include <stdio.h>
#define SHMSZ 27
union semun {
    int val;
    struct semid_ds *buf;
    unsigned short int *array;
} ;
int sem;
key_t SEM_KEY = 4545;
int open_semaphore();
int lock_semaphore();
int unlock_semaphore();
void free_semaphore();
static void signal_handler( int );
```

```
main()
{
  int shmid, sem;
  key_t shm_key, SEM_KEY;
  char *shm, *s;
  SEM_KEY = 4545;
  /*We need to get the segment named "5678", created by the server.*/
  shm_key = 5678;
  /* Locate the segment.*/
  if ((shmid = shmget(shm_key, SHMSZ, 0666)) < 0) {
    perror("shmget");
    exit(1);
  }
  /* Now we attach the segment to our data space.*/
  if ((shm = shmat(shmid, NULL, 0)) == (char *) -1) {
    perror("shmat");
    exit(1);
  }
```

```
   open_semaphore();
   /* Now read what the server put in the memory.*/
    lock_semaphore();
    for (s = shm; *s != NULL; s++){
      putchar(*s);
    }
    unlock_semaphore();
    putchar('\n');
    /* Finally, change the first character of the segment to '*',
    indicating we have read the segment.*/
    *shm = '*';
    sleep(1);
    /* nie je nutne po exite sa vykona odpojenie*/
    shmdt(shm);
    exit(0);
}
```

```
int open_semaphore()
{                                                    //alokovanie semafora
  sem = semget(SEM_KEY, 1, 0664);
  if (sem == -1) {
            perror("semget() failed");
            exit(1);
  }
  return 0;
}
```

# Mapovaná pamäť

- Umožňuje komunikáciu procesov cez zdieľané súbory ako aj rýchly prístup procesov k súborom.
- Mapovaná pamäť (MP) tvorí asociáciu medzi súbormi a pamäťou procesu. Linux rozdelí súbor do stránok a skopíruje ich do stránok virtuálnej pamäti tak, že sú dostupné v adresnom priestore procesu. Proces môže potom pristupovať k obsahu súboru ako s prístupom do pamäte.
- Stránky pamäti sa zapíšu do súboru iba ak sa zmení ich obsah. Takže čítanie a zapisovanie do súborov je plne automatické a vysoko efektívne.
- K súborom môže takto pristupovať aj viac procesov, čo poskytuje veľmi účinné zdieľanie pamäte.

# Mapovaná pamäť

- Mapovanie vykonáva funkcia mmap()

  caddr_t mmap(caddr_t addr, size_t len, int prot, int flags, int fd, off_t off); kde

  - prvý argument je adresa v adresnom priestore procesu, od ktorej sa má súbor mapovať, ak je NULL Linux vyberie vhodnú adresu

  - druhý argument je veľkosť v bytoch

  - tretí argument obsahuje prístupové práva PROT_READ, PROT_WRITE a PROT_EXEC, PROT_NONE (môžu sa kombinovať)

  - štvrý argument je flag

  - piaty argument je deskriptor mapovaného súboru

  - posledný argument je offset od začiatku súbora, od ktorého chceme mapovať.

# Mapovaná pamäť

- Hodnoty flagu môžu byť:

  MAP_FIXED – používa sa špecifikovaná adresa pre mapovanie

  MAP_PRIVATE – zápis do pamäti sa neodpamätá do súboru ale bude v privátnej kópii súboru. Ďalšie procesy tento zápis nevidia.

  MAP_SHARED – zápis sa okamžite prenesie do súboru, používa sa na komunikáciu procesov. Nemôže sa použiť s MAP_PRIVATE.
- Funkcia vracia smerník na začiatok pamäti v prípade neúspechu vracia MAP_FAILED.

# Mapovaná pamäť

- MP sa uvolní funkciou:

  munmap(caddr_t adr, size_t len), kde

  - adr je adresa mapovanej pamäti

  - len je jej veľkosť.

  Funkciu nie je nutné volať, pretože Linux automaticky uvolní pamäť po ukončení procesu.
- Nasledovné programy dokumentujú použitie MP

# Príklad 1

- prvý program generuje náhodné číslo a zapisuje ho do mapovanej pamäte súboru
- druhý program  číslo prečíta a vytlačí a jeho dvojnásobok zapíše do mapovanej pamäte súboru

# Program1

**Write a Random Number to a Memory-Mapped File (mmap-write.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
/* Return a uniformly random number in the range [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;
  /* Seed the random number generator. */
  srand (time (NULL));
  /* Prepare a file large enough to hold an unsigned integer. */
  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);
  /* Create the memory mapping. */
  file_memory = mmap (0, FILE_LENGTH, PROT_WRITE, MAP_SHARED, fd, 0);
  close (fd);
  /* Write a random integer to memory-mapped area. */
  sprintf((char*) file_memory, “%d\n”, random_range (-100, 100));
  /* Release the memory (unnecessary because the program exits). */
  munmap (file_memory, FILE_LENGTH);
  return 0;
}
```

# Program2

**Read an Integer from a Memory-Mapped File, and Double It (mmap-read.c)**

```
#include <stdlib.h>
#include <stdio.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;
  /* Open the file. */
  fd = open (argv[1], O_RDWR, S_IRUSR | S_IWUSR);
```

```
/* Create the memory mapping. */
  file_memory = mmap (0, FILE_LENGTH, PROT_READ |
    PROT_WRITE, MAP_SHARED, fd, 0);
  close (fd);
  /* Read the integer, print it out, and double it. */
  sscanf (file_memory, “%d”, &integer);
  printf (“value: %d\n”, integer);
  sprintf ((char*) file_memory, “%d\n”, 2 * integer);
  /* Release the memory (unnecessary because the program exits)*/
  munmap (file_memory, FILE_LENGTH);
  return 0;
}
```

# Výstup

- Nasleduje ukážka použitia programov, ktorá mapuje súbor  /tmp/integer-file.

  % ./mmap-write /tmp/integer-file

  % cat /tmp/integer-file

  42

  % ./mmap-read /tmp/integer-file

  value: 42

  % cat /tmp/integer-file

  84

  – Programy použili funkcie sscanf a sprintf ako pre prácu s textom len pre demonštráciu.

# Zdieľaný prístup k súboru

- Rôzne procesy môžu komunikovať cez mapovanú pamäť asociovanú k tomu istému súboru
- Hodnota flagu MAP_SHARED spôsobí, že každý zápis do pamäti sa okamžite prenesie do súboru a je viditeľný inými procesmi.

# Zdieľaný prístup k súboru

- Ak sa nešpecifikuje flag ako MAP_SHARED Linux zapisuje zmeny súboru do bufra v pamäti.
- Alternatívne okamžitý zápis bufra do súboru na disku spôsobí volanie funkcie msync (caddr_t adr, size_t len, int flag)

  - prvé dva argumenty sú ako vo funkcii munmap

  - flag môže mať hodnoty:

  MS_ASYNC – zápis sa nemusí uskutočniť nevyhnutne po návrate z funkcii

  MS_SYNC – zápis sa vykoná hneď, volanie je blokované na funkcii pokiaľ sa zápis nevykoná

  MS_INVALIDATE – všetky ostatné mapovania súboru sú neplatné, takže zmeny v súbore sú zverejnené

# Zdieľaný prístup k súboru

- Príklad volania:

  msync (mem_addr, mem_length, MS_SYNC | MS_INVALIDATE);

  spôsobí, že zápis do súboru sa okamžite vykoná a zmeny sú viditeľné
- MP musí byť chránená semafórom proti súčasnému prístupu viacerých procesov podobne ako zdieľaná pamäť.
- Alternatívne, možno použiť funkciu fcntl na uzamknutie súboru pred čítaním alebo zápisom.

# Príklad 2

- Programy ukazujú synchronizovaný prístup (vzájomné vylúčenie) dvoch procesov pri prístupe k súboru ako mapovanej pamäti
- Jeden proces bude do súboru zapisovať druhý bude zo súboru čítať
- Vzájomné vylúčenie znamená, že oba procesy nesmú k súboru pristupovať súčasne
- Na synchronizáciu procesov použijeme semafór

**Program writer.c**

```
#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <sys/sem.h>
#include <unistd.h>
#define FILE_LENGTH 0x100
#define SEM_RESOURCE_MAX 1 /* Initial value of all semaphores */
#define SEMMSL 10
union semun{
        int val;
        struct semid_ds *buf;
        unsigned short int *array;
};
void createsem(int *sid, key_t key, int members);
void locksem(int sid, int member);
void unlocksem(int sid, int member);
void removesem(int sid);
int getval(int sid, int member);
void dispval(int sid, int member);
```

```
int main (int argc, char* const argv[])
{
  key_t key;
  int semset_id;
  char c;
  int fd;
  char* file_memory, *s;
  key = ftok(".", 's');
  /* Prepare a file large enough to hold an unsigned integer. */
  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);
  /* Create the memory mapping. */
  file_memory = mmap (0, FILE_LENGTH, PROT_WRITE, MAP_SHARED,
  fd, 0);
  close (fd);
  createsem(&semset_id, key, 1);
```

```
/* Now put some things into the map file for the other process to read */
  s = file_memory;
  locksem(semset_id, 0);
  for(c = 'a'; c<= 'z'; c++){
    *s++ = c;
    *s = '\n';
    sleep(1);
  }
  unlocksem(semset_id, 0);
  sleep(1);
  /* Release the memory (unnecessary because the program exits). */
  munmap (file_memory, FILE_LENGTH);
  removesem(semset_id);
  return 0;
}
```

```
void createsem(int *sid, key_t key, int members)
{
  int cntr;
  struct semid_ds mysemds;
  union semun semopts;
  if(members > SEMMSL) {
            printf("Sorry, max number of semaphores in a set is %d\n",SEMMSL);
  exit(1);
  }
  printf("Attempting to create new semaphore set with %d members\n",members);
  if((*sid = semget(key, members, IPC_CREAT|IPC_EXCL|0666))== -1)
  {
            fprintf(stderr, "Semaphore set already exists!\n");
            exit(1);
  }
  semopts.val = SEM_RESOURCE_MAX;
  /* Initialize all members (could be done with SETALL) */
  for(cntr=0; cntr<members; cntr++)
  semctl(*sid, cntr, SETVAL, semopts);
}
```

```
void locksem(int sid, int member)
{
  struct sembuf sem_lock={ member, -1, SEM_UNDO};

  if((semop(sid, &sem_lock, 1)) == -1)
  {
          fprintf(stderr, "Lock failed\n");
          exit(1);
  }
  else
          printf("Semaphore resources decremented by one (locked)\n");
  dispval(sid, member);
}
```

```
void unlocksem(int sid, int member)
{
  struct sembuf sem_unlock={ member, 1, SEM_UNDO};
  int semval;

  /* Is the semaphore set locked? */
  semval = getval(sid, member);
  if(semval == SEM_RESOURCE_MAX) {
            fprintf(stderr, "Semaphore not locked!\n");
            exit(1);
  }
  /* Attempt to lock the semaphore set */
  if((semop(sid, &sem_unlock, 1)) == -1)
  {
            fprintf(stderr, "Unlock failed\n");
            exit(1);
  }
  else
            printf("Semaphore resources incremented by one (unlocked)\n");
            dispval(sid, member);
}
```

```
void removesem(int sid)
{
  semctl(sid, 0, IPC_RMID, 0);
  printf("Semaphore removed\n");
}

int getval(int sid, int member)
{
  int semval;
  semval = semctl(sid, member, GETVAL, 0);
  return(semval);
}

void dispval(int sid, int member)
{
  int semval;
  semval = semctl(sid, member, GETVAL, 0);
  printf("semval for member %d is %d\n", member, semval);
}
```

**Program reader.c**

```
#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <sys/sem.h>
#include <unistd.h>
#define FILE_LENGTH 0x100
#define SEM_RESOURCE_MAX 1 /* Initial value of all semaphores */
#define SEMMSL 10
union semun{
        int val;
        struct semid_ds *buf;
        unsigned short int *array;
};
void opensem(int *sid, key_t key);
void locksem(int sid, int member);
void unlocksem(int sid, int member);
void removesem(int sid);
int getval(int sid, int member);
void dispval(int sid, int member);
```

```
int main (int argc, char* const argv[]){
  key_t key;
  int semset_id;
  int fd;
  char* file_memory, *s;
  key = ftok(".", 's');
  /* Open the file. */
  fd = open(argv[1], O_RDWR, S_IRUSR | S_IWUSR);
  /* Create the memory mapping. */
  file_memory = mmap(0, FILE_LENGTH, PROT_READ | PROT_WRITE,
  MAP_SHARED, fd, 0);
  close (fd);
  opensem(&semset_id, key);
  /* Now read what the writer put into the file */
  locksem(semset_id, 0);
  for(s = file_memory; *s != '\n'; s++){
    putchar(*s);
  }
  putchar('\n');
  unlocksem(semset_id, 0);
  sleep(1);
  munmap(file_memory, FILE_LENGTH);
  return 0;
}
```

```
void opensem(int *sid, key_t key)
{
  /* Open the semaphore set - do not create! */
  if((*sid = semget(key, 0, 0666)) == -1)
  {
          printf("Semaphore set does not exist!\n");
          exit(1);
  }
}
```

Poznámka

Programy možno spustiť v dvoch oknách v poradí „writer“ „reader“ s argumentom meno súboru