LABORATOR 2

Functii sistem -> sectiunea 2 a manualului

*Int main(int argc, char * argv[])*

fd= descriptor

struct stat sb;
int **STAT**(const char *path, &sb); => informatii despre obiectele manipulate (dim, permisiuni de acces)-> ex: sb.st_size => %jd

int **OPEN**(const char*path, int flags, mode_t mode) ;=> descriptorul asociat

flags: O_WRONLY, O_RDONLY, O_RWONLY

* O_CREAT (daca fisierul nu exista) + drepturi de acces: S_I+ R, W +USR

int **CLOSE**(int fd);

ssize_t **READ**(int fd, void * buf, size_t nbytes) ;=> nr de bytes cititi (cand sunt 0 -> final fisier)

ssize_t **WRITE**(int fd, const void * buf, size_t nbytes(*strlen*)); => nr bytes scrisi

ERRNO-> variabila globala cu informatii despre erori

perror(“mesaj”)-> afisare mesaj asociat erorii

*Apelare executabil cu extensiile fisierului ./lab2_2.c foo.txt bar.txt

LABORATOR 3-Adaugarea unei noi functii sistem

1. Compilare kernel:

#cd sys/arch/amd64/compile/GENERIC.MP
#make obj #make config #make (#cp /bsd/bsd.l) **#make install (#reboot)**

2. F. de sistem sunt definite in: **# (nano) /sys/kern/syscalls.master**
   ID+1 STD {tip sys_numef(p);}

3. Regenerare fisiere C: **#cd /sys/kern && make syscalls** (nu trebuie sa dea erori)

(kern/syscalls.c-denumire f, sys/syscallargs.h-structura si functie, sys/syscall.h-id)

4. Definire functie: **# (nano) /sys/kern/sys_generic.c** (aici punem implementarea functiei)

(struct proc *p, void *v, register_t *retval)

Return 0; -> return catre kernel

Retval-> return catre user

Struct sys_numef_args *uap=v -> citire argumente de intrare

Macroul SCARG(uap, msg) –citesc un argument(incarca din registru)

&done-> pune in el cat a reusit sa copieze din msg in kbuf

Mutare in spatiul de adresare a kernelului a bufferului-> functiile de copiere

In kernel NU exista biblioteca C.

5. Creare fisier C: **#cd # nano exn.c (aici apelez syscall)**

Apelare din userland: syscall(p);
Ex pt write: syscall(4,1,"msg",6) -> 4-id write

6. **gcc exn.c –o exn**

./exn -> bad system call => **recompilare kernel**

Copyoutstr(p)->copiaza string kernel-> user

Copyinstr(p)->copiaza string user-> kernel

LABORATOR 4

PROCESE

-> functia de sistem cu care se creaza procese este **fork(**2)=> return process ID of child to the parent, but this fork return 0 to the child itself

!Incepere evaluarea codului imediat dupa fork()

Parintele si copilul incep sa se execute in acelasi timp.

pid_t fork()=> id procees copil

fork()>0 -> Succes (Parent process), Fork()<0-> Error, Fork()=0 -> Child Process

Copilul primeste 0.

Parintele primeste pid-ul copilului.

Pid proces curent: **getpid**();

Pid proces parinte: **getppid**();

Wait(NULL)-> suspendare activitate parinte pana la finalizarea procesului fiu

Wait(NULL)+getpid()=> pid proces copil

Waitpid(pid, NULL, 0)-> asteapta dupa un anumit copil

int **EXECVE**(const char * path, char * const argv[], char * const envp[]);

-> suprascrie procesul apelant cu un nou proces (NU face un alt proces)

-> nu mai revine in programul initial (nu mai re treaba cu ce era dupa el in programul initial)

path-calea absoluta (cea din /bin/...)

argv[]- argumentele programului

envp[]- variabilele sistemului

LABORATOR 5 *la compilare flag-ul –lrt*

Mmemorie partajata

Creare zona -> dimensionare zona -> mapare -> demapare -> stergere zona

Int **SHM_OPEN**(const char *path, int flags, mode_t mode) as OPEN => file descriptor

* in caz de eroare => stergem obiectele create cu **SHM_UNLINK**(int fd);

Int **FTRUNCATE**(**int fd,** size_t size) -> definire dimensiune descriptor

Void* **MMAP**(void *adr, size_t len, int prot, int flags, **int fd**, off_t offset) => pointer catre adresa din spatiul procesului la care a fost incarcat obiectul

->incarcare memorie partajata

adr – adresa (de obicei 0 ca sa decida kernelul)

len-dim memorie de incarcat

prot-drepturi de acces (PROT_READ, PROT_WRITE)

flags- tip de memorie (MAP_SHARED)

fd-descriptor

offset- locul in obiectul de memorie partajata DE LA CARE sa fie incarcat in spatiul procesului

ex: mmap(0, 100, PROT_WRITE, MAP_SHARED, shm_fd, 500) –indica catre o parte de 100 bytes care incepe de la byte-ul 500 din zona de memorie aferenta descriptorului ce va fi doar scrisa (PROT_WRITE) si impartita cu restul proceselor (MAP_SHARED)

Atentie! Dimensiunea este multiplu de pagini: n* **getpagesize**();

Int **MUNMAP**(void *adr, size_t len)

adr-pointerul catre zona de memorie incarcata in spatiul procesului

len-dimensiunea / o parte = cat a fost incarcata

LABORATOR 6 *compilare cu flagul –pthread*

Fire de executie

-> orice modificare facuta in spatiul procesului de un fir este instantaneu vizibila tuturor celorlalte fara a apela la un mecanism exterior

Diferenta: threadurile nu duplica memoria, doar registrii si se continua stiva (fiecare thread cu stiva lui)

Int **PTHREAD_CREATE** (pthread_t *thread, const pthread_attr_at *attr, void * (* start_routine)(void*), void *arg);

-> initializeaza thread cu noul fir de executie lansat prin apelarea functiei start_routine

attr – implicit valoarea NULL pt atributele setate de sistemul de operare

arg – parametrii functiei start_routine

Int **PTHREAD_JOIN**(pthread_t thread, void **value_ptr);

-> pentru asteptarea explicita (nu ca wait) a finalizarii executiei unui thread

-> daca value_ptr !=NULL atunci pune rezultatul functiei start_routine

LABORATOR 7 *compilare cu flagul –pthread*

Sincronizare -> Programare paralela

-zona in care are voie un singur proces/thread =critical section

-accesul la aceasta zona se face prin obiectul mutex (mutual exclusive)

Int **PTHREAD_MUTEX_INIT**(pthread_mutex_t *mutex, const pthread_mutexattr_t *attr); -> creaza si initializeaza mutex cu atributele attr (aici NULL-atributele implicite)

-> vaiabilele mutex se pun in memoria globala/ structura accesibila tuturor firelor de executie, dupa care se initializeaza.

-> stari mutex: locked(threadul detine dreptul exclusiv de executie asupra zonei critice),unlocked

Functii folosite: **pthread_mutex_lock**(&mtx)-> blocheaza firul de executie

pthread_mutex_unlock(&mtx);

pthread_mutex_destroy(&mtx);

Semafor: sem_t sem;

!Un thread intra in zona critica cand un semafor are orice valoarea diferita de 0.

Semaforul poate da mai multe valori deci poate lasa mai multe thread-uri in zona critica.

int **SEM_INIT**(sem_t *sem, int pshared, unsigned int value);-> initializare semafor

-seteaza valoarea lui sem cu value

-pshared semnaleaza daca vrem sa folosim semaforul in mai multe procese (aici 0)

Int **SEM_WAIT**(&sem);-> scade valoarea semaforului cu o unitate

Int **SEM_POST**(&sem);-> creste valoarea semaforului cu o unitate

-> se elibereaza thread-ul care asteapta cel mai mult in coada

Int **SEM_DESTROY**(&sem);->eliberare semafor